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 and external DRAMs. Current HLS tools allow exploiting 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 v
List of Tables vi
List of Acronyms vii

1 Background 1
1.1 Cache .................................................. 1
   1.1.1 Structure ........................................... 1
   1.1.2 Policies ............................................ 2
   1.1.3 Benefits .......................................... 3
1.2 Field-Programmable Gate Array .............................. 4
   1.2.1 Memory system ...................................... 4
1.3 High-Level Synthesis .................................... 4
   1.3.1 Workflow .......................................... 4
   1.3.2 Optimization techniques ......................... 5

2 Motivation 8
2.1 Ma’s cache ............................................. 8
2.2 Proposed solution ..................................... 9

3 Basic cache 10
3.1 Architecture ......................................... 10
   3.1.1 Functionality ...................................... 10
   3.1.2 Characteristics ................................... 11
   3.1.3 Single-process Basic cache ....................... 11
   3.1.4 Multi-processes Basic cache ...................... 12
3.2 Implementation ....................................... 12
   3.2.1 Internals ......................................... 14
   3.2.2 Interface ......................................... 15

4 Multi-levels cache 18
4.1 Architecture ......................................... 18
4.2 Implementation ...................................... 18
# List of Figures

1.1 Cache logic structure. ......................................................... 2
1.2 Set associative policy address bits meaning. ......................... 3
1.3 Pipelining example. .......................................................... 5
1.4 Loop unrolling example. ..................................................... 6
1.5 Array partitioning examples. ............................................... 7
1.6 Burst read and write example. ............................................ 7

3.1 *Single-process Basic cache* architecture. ............................... 10
3.2 *Multi-processes Basic cache* architecture. ............................. 12
3.3 Stalling schedule of request writing and response reading. .......... 16
3.4 Optimal schedule of request writing and response reading. .......... 17
3.5 Static schedules in case of multiple accesses per iteration. ........ 17

4.1 *Multi-levels cache* architecture. ......................................... 19

5.1 *Multi-ports cache* architecture. ......................................... 21
5.2 Static schedules in case of 2-ports cache. ............................. 22

6.1 Design space of *Matrix multiplication 16x16* (single-level). ........ 27
6.2 Request and response waveforms for *Matrix multiplication 16x16* single-level and single-port. ............................................. 28
6.3 Design space of *Matrix multiplication 16x16* (multi-levels). ........ 29
6.4 Design space of *Matrix multiplication 32x32* (single-level). ......... 31
6.5 Pareto curve of *Matrix multiplication 32x32*. ......................... 34
6.6 Pareto curve of *Bitonic sorting 128*. .................................. 38
6.7 Pareto curve of *2D convolution*. ....................................... 42
List of Tables

3.1 Data exchanged through Port. .................................................. 13
6.1 Simulation environment configuration. ................................. 23
6.2 Single-level cache configuration for Matrix multiplication 16x16. ...... 27
6.3 Performance and resource usage of Matrix multiplication 16x16 (single-level). 28
6.4 Multi-levels cache configuration for Matrix multiplication 16x16. .......... 29
6.5 Performance and resource usage of Matrix multiplication 16x16 (multi-levels). 30
6.6 Performance and resource usage of Matrix multiplication 16x16. .......... 30
6.7 Single-level cache configuration for Matrix multiplication 32x32. ............. 31
6.8 Performance and resource usage of Matrix multiplication 32x32 (single-level). 32
6.9 Multi-levels cache configuration for Matrix multiplication 32x32. .......... 32
6.10 Multi-levels cache Matrix multiplication 32x32 hit ratios. ............ 32
6.11 Performance and resource usage of Matrix multiplication 32x32 (multi-levels). 33
6.12 Performance and resource usage of Matrix multiplication 32x32. ............. 33
6.13 Single-level cache configuration for Bitonic sorting 128. ............... 36
6.15 Multi-levels cache configuration for Bitonic sorting 128. ............... 37
6.16 Performance and resource usage of Bitonic sorting 128 (multi-levels). ..... 37
6.17 Performance and resource usage of Bitonic sorting 128. ............... 38
6.18 kernel and B caches configuration for 2D convolution. .................. 40
6.19 Performance and resource usage of 2D convolution (single-level). ......... 40
6.20 Performance and resource usage of 2D convolution (multi-levels). ........ 41
6.21 Performance and resource usage of Convolution. ..................... 41
List of Acronyms

API  Application Programming Interface
AXI  Advanced eXtensible Interface
BRAM  Block RAM
CC  Clock Cycle
DRAM  Dynamic RAM
DSP  Digital Signal Processor
FF  Flip-Flop
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
LUT  Lookup Table
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 a 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 belong 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 belong 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.
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.
1. Background

• **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:

\[
H := \frac{n_{hit}}{n_{tot}}
\]  

(1.1)
1. Background

The average access time and energy are defined as:

\[
\begin{align*}
\bar{t}(H) &:= H t_{L1} + (1 - H) t_{L2} \\
\bar{E}(H) &:= H E_{L1} + (1 - H) E_{L2}
\end{align*}
\] (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 2020.2 [2] and 2021.1 [3], 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.](image-url)
1. Background

**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 \( L_{loop,\text{rolled}} := N \cdot L_{it} \).

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: \( L_{loop,\text{unrolled}}(f) := \frac{N}{f} \cdot L_{it} \).

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

![Figure 1.4: Loop unrolling example.](image)

**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:**
1. Background

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 Motivation

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.

The proposed cache module has the dual purpose of:

- Reducing the number of DRAM accesses: misses only need 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 Ma’s cache


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).
2. Motivation

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.

2.2 Proposed solution

The primary goal of this thesis work 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.
3 Basic cache

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

3.1 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 3.1): this isolation makes the cache always perform in the same manner, independently of the algorithm accessing it, while keeping the application logic as clean as possible, since application only has to write requests to cache and read responses, instead of integrating the whole cache logic.

![Single-process Basic cache architecture.](image)

3.1.1 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*.

### 3.1.2 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).

### 3.1.3 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.

- 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 cache process II, reducing cache performance.
3.1.4 Multi-processes Basic cache

The Multi-processes Basic cache splits cache into two processes (Figure 3.2):

- **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-only 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 write-only caches too.

In the case of caches which are accessed both in read and in write mode, it has not been possible to achieve an II of 1, due to dependencies on the cache memory. Given that a read-write cache implies at least one read access and one write access,

![Figure 3.2: Multi-processes Basic cache architecture.](image)

3.2 Implementation

The Basic cache is implemented in the form of a C++14 [4] 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:
3. Basic cache

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

**Internals** and **Interface** communicate with each other through a **Port** (Table 3.1), in a **Master/Slave** fashion:

- **Interface** sends to **Internals** a **request** (operation, address and write data).
- **Internals** sends to **Interface** a **response** (read data), after executing the requested operation.

<table>
<thead>
<tr>
<th>Content</th>
<th>Description</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Read/Write</td>
<td><strong>Internals</strong> → <strong>Interface</strong></td>
</tr>
<tr>
<td>Address</td>
<td>Index to be accessed</td>
<td><strong>Internals</strong> → <strong>Interface</strong></td>
</tr>
<tr>
<td>Write data</td>
<td>Data to be written to memory</td>
<td><strong>Internals</strong> → <strong>Interface</strong></td>
</tr>
<tr>
<td>Read data</td>
<td>Data read from memory</td>
<td><strong>Internals</strong> ← <strong>Interface</strong></td>
</tr>
</tbody>
</table>

Table 3.1: Data exchanged through Port.

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

The proposed model follows the **Master/Slave** paradigm:

1. **Master** sends a request to **Slave**.
2. **Slave** executes the requested operation and optionally sends a response to **Master**.

**Slave** must be modeled as an infinite loop which waits for requests from **Master** before executing its functionality, while **Master** can be modeled as standard sequential code (or it can be in turn a **Slave** of another **Master**).

The parallelism between **Master** and **Slave** 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.

The communication between the two processes is performed through a **port**, which contains data flowing from **Master** to **Slave** (request) and from **Slave** to **Master** (response). Request and response are mapped to one or more FIFOs which are written from the transmitter and read from the receiver. `hls::stream` class by Vitis™ HLS can be used as FIFO implementation.
3.2.1 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.

**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.

**AXI optimizations**

To exploit the Vitis™ HLS support to automatic port widening and burst accesses to AXI interface, every access to external DRAM accesses a whole cache line. The accessed
addresses Least Significant Bits (LSBs) are explicitly set to 0 so that synthesizer can infer that they are aligned to the line size.

If the cache line is at most equal to the maximum AXI interface width, it is accessed in a single request, otherwise it is accessed in multiple burst requests.

**Read After Write dependencies**

In case of read-write caches, the *Core* process II increases to 3 due to RAW dependencies on the cache BRAM.

To mitigate this issue 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 with distance 1 on the cache memory (by setting to *false* the RAW inter-iteration dependencies and to *true* the RAW inter-iteration dependencies with distance 1).

This solution allows to schedule the cache process with an II equal to 2. The *RAW cache* could be extended to a fully-associative cache complying with the FIFO replacement policy, allowing to falsify the RAW dependency with distance 2 and achieving an II of 1.

A read-write cache implies that it is accessed at least two times per iteration (once in read mode, once in write mode), therefore, due to the issues discussed in Subsection 3.2.2 it is not possible to fully exploit the cache pipelining. In this case the cache II does not have a relevant impact on effective performance: *RAW cache* could not provide real advantages and it has not been included in the final design, to keep it simpler.

### 3.2.2 Interface

*Interface* provides APIs for managing requests and responses between application and 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)`.)
3. Basic cache

**Deadlock prevention**

The HLS scheduler is not able to infer the dependency between the request writing \((W)\) and the response reading \((R)\) in the \texttt{get} function (i.e. it is not aware that first the request has to be written, then it is necessary to wait for the cache latency and finally the response has to be read).

For that reason the scheduler optimizes the logic by inserting both \(W\) and \(R\) into the same pipeline stage. This leads to a deadlock: \(R\) is blocked since it reads from an empty FIFO (it cannot contain the response yet) and it blocks the whole stage, including \(W\), making \(R\) wait for the response to a request which cannot be sent.

The deadlock has been fixed by inserting a clock operation between \(W\) and \(R\) (calling \texttt{ap\_wait}), which forces \(W\) and \(R\) to separate pipeline stages.

**Cache pipeline exploiting**

At the steady state, in case of hit, the cache can process one request per cycle, thanks to its optimal pipelining (i.e. \(II\) equal to 1).

HLS is not aware of the dependency and latency between request writing \((W)\) and response read \((R)\), so it schedules \(R\) just after \(W\) (Figure 3.3a): at runtime \(R_i\), which should be executed in the cycle following \(W_i\), stalls, since the cache response has a latency (and \(W_{i+1}\) stalls too, by consequence).

\(W_{i+1}\) is executed after waiting for the full latency of the cache (Figure 3.3b) and the final result is that cache never receives multiple requests in consecutive cycles, it never reaches the steady state and its throughput is the same as if it were not pipelined.

\[
\begin{array}{c|c|c|c|c}
W_0 & \text{STALL} & \text{STALL} & \text{STALL} & R_0 \\
W_1 & \text{STALL} & \text{STALL} & \text{STALL} & R_1 \\
W_2 & & & & \\
\end{array}
\]

(a) Static schedule. \hspace{2cm} (b) Runtime.

Figure 3.3: Stalling schedule of request writing and response reading.

To mitigate this issue the \texttt{ap\_wait} between request write and response read has been replaced with \texttt{ap\_wait\_n(LATENCY)}, where \texttt{LATENCY} is an integer value set through a template parameter. This forces the scheduler to insert \texttt{LATENCY} clock cycles between \(W\) and \(R\) (Figure 3.4a), so that at runtime stalls are avoided (in case of hit) and one request per cycle is sent to cache (Figure 3.4b).

\texttt{LATENCY} is not set to a constant because its optimal value highly depends on memory access pattern and cache configuration, and can be determined by means of design exploration.

This is a partial solution: the \texttt{ap\_wait} forces all the subsequent operations to wait: when there are multiple calls to \texttt{get} per iteration (e.g. \(A\) and \(B\)), \(W_B\) has to wait...
3. Basic cache

![Diagram](image)

(a) Static schedule.    (b) Runtime.

Figure 3.4: Optimal schedule of request writing and response reading.

**LATENCY** cycles after $W_A$ before being scheduled (Figure 3.5a). This situation makes the application loop II to increase, since it must guarantee the order of accesses to FIFOs (i.e. $W_{A,i+1}$ cannot be executed before $W_{B,i}$).

To actually fix this problem (with the schedule shown in Figure 3.5b), a mechanism for informing the scheduler about dependencies and latency between specific operations is probably needed, but this is not available in *Vitis™ HLS 2021.1*.

![Diagram](image)

(a) Achieved static schedule.    (b) Optimal static schedule.

Figure 3.5: Static schedules in case of multiple accesses per iteration.
4 Multi-levels cache

The Multi-levels cache is aimed at improving performance by making the memory hierarchy deeper, adding a faster L1 cache memory on top of it. This alternative approach has been proposed to overcome the difficulties, to fully exploit the optimal pipeline of the Basic cache, due to the scheduler unawareness about the latency between request writing and response reading (as explained in Section 3.2).

4.1 Architecture

The Multi-levels cache introduces a L1 cache inlined in the application logic (Figure 4.1): the scheduler exactly knows the latency of each L1 cache operation and can build an application pipeline which stalls in case of L1 miss only.

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 more frequent than writes, and they suffer the most from the scheduling issues which lead to the introduction of the L1 cache, this has been considered the best trade-off.

4.2 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. When it is set to 0, the resulting architecture is equivalent to the Basic cache.

4.2.1 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).
4.2.2 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 forwards the request to the L2 cache.
5 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.

5.1 Architecture

The Multi-ports cache is characterized by multiple ports accessed in parallel (Figure 5.1).

Each port has dedicated logic for communicating with the shared L2 cache and an independent L1 cache.

Multiple independent ports allow removing dependencies between different accesses to the cache. This brings the advantage of achieving better performance, making it possible to schedule multiple requests at the same time, without increasing the application loop II, but it also brings the disadvantage of not guaranteeing the expected ordering between different accesses. To guarantee the correct functionality the Multi-ports architecture is compatible with read-only accesses.

5.2 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. When it is set to 1, the resulting architecture is equivalent to the Multi-levels cache.

5.2.1 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 guarantees all the dependencies on cache data structures, and the resulting II of the Core process is equal to PORTS.

5.2.2 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) is automatically associated with a specific port by means of a member variable holding the port index and is updated after each access.

FIFOs accesses scheduling

Ideally the request write (W) and the response read (R) should be scheduled in parallel in the same cycle (Figure 5.2b). Due to the scheduler limitations (described in Subsection 3.2) it is not possible to achieve such a schedule, since there is a forced clock cycle between W and R, which delays all the subsequent operations.

The resulting schedule (Figure 5.2a) is almost equivalent to the one achieved with the Basic cache in case of multiple accesses per iteration (Figure 3.5a), with the difference that request and response FIFOs are distinct, since they belong to separate ports, therefore the scheduler does not have to ensure dependencies between subsequent reads and writes and application loop II does not increase.

At the steady state, in case of hit, one W and R are executed per cycle, allowing to fully exploit the L2 cache pipeline.
5. Multi-ports cache

(a) Achieved static schedule.  
(b) Parallel static schedule.

Figure 5.2: Static schedules in case of 2-ports cache.

5.3 Limitations

In some particular situations (e.g. when cache is explicitly accessed multiple times per iteration) the simulation of the generated circuit 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.
6 Results

The proposed cache architecture has been embedded in multiple *Vitis HLS* kernels implementing different algorithms, to evaluate both the performance gain and the resource usage of different cache configurations.

Each algorithm has been selected for its memory intensiveness and for its specific memory access patterns.

6.1 Simulation environment

Kernels have been synthesized by the C Synthesis in *Vitis™ HLS 2021.1*, targeting the `xcvu9p-flgb2104-2-e` part, running at a clock frequency of 250 MHz.

*Vitis™ 2021.1* provides two main kind of simulation:

- Hardware Emulation: accurate, but slow.
- C/RTL Co-Simulation: fast, but not very accurate (especially for what concerns the AXI interface model).

HW Emulation has been used for determining the delay of the AXI interface (which is around 4 clock cycles). The AXI latency has been accordingly set to 3, so that the synthesizer can better optimize the circuit and Co-Simulation results match HW emulation as much as possible.

<table>
<thead>
<tr>
<th>Synthesizer</th>
<th>C Synthesis in <em>Vitis™ HLS 2021.1</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator</td>
<td>C/RTL Co-Simulation in <em>Vitis™ HLS 2021.1</em></td>
</tr>
<tr>
<td>Flow target</td>
<td><code>vitis</code></td>
</tr>
<tr>
<td>Part codename</td>
<td><code>xcvu9p-flgb2104-2-e</code></td>
</tr>
<tr>
<td>Clock period</td>
<td><code>4ns</code></td>
</tr>
<tr>
<td>AXI latency</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.1: Simulation environment configuration.
6. Results

6.1.1 Reference memory models

The results have been compared with the output of synthesis and simulation of same algorithms implemented with different data access mechanisms: global memory (performance lower bound) and local memory (performance higher bound).

Global memory

The algorithms access data directly from external DRAM through AXI interface: this is the straightforward but slowest solution, therefore it determines the performance lower bound.

Local memory

All the data required by algorithms is stored to local BRAMs: it determines the performance higher bound, but it is unfeasible in general, due to the limited amount of BRAMs.

   With this solution the kernel:

   1. Moves all the input data from DRAM to BRAMs.
   2. Performs all the computations accessing data to and from BRAMs.
   3. Moves all the output data from BRAMs to DRAM.

The execution time of DRAM accesses is not of interest, therefore it has been subtracted from reported results.

Ma’s cache

Ma’s cache was designed for Vivado™ HLS 2016.2: with some minor changes it is possible to synthesize it with Vitis HLS 2021.1, but it would need some more optimizations to achieve the original performance in the new environment.

   The results reported in Ma’s paper “Acceleration by Inline Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis” and PhD thesis “Low power and high performance heterogeneous computing on FPGAs” are not comparable with the ones obtained in Vitis HLS 2021.1: the execution times of same algorithms with same configurations (i.e. global and local memory) differ up to one order of magnitude (most probably due to different AXI latency values), therefore also cache execution times would not be reliable for making comparisons.

   The lack of comparable results and the impossibility to generate new ones prevented from directly comparing the proposed cache results with the Ma’s ones.
6. Results

6.1.2 Collected data

The most relevant collected data concerns:

- Performance: evaluated in terms of execution time (i.e. the time at which the simulation of the algorithm terminates).
- Resource usage: evaluated in terms of number of used BRAMs, Digital Signal Processors (DSPs), Lookup Tables (LUTs) and Flip-Flops (FFs).

These values are approximate, since they come from C/RTL Co-Simulation and from the estimations performed by the C Synthesis, but in any case they can be meaningful for identifying some trends.
6. Results

6.2 Matrix multiplication

The standard row-by-column Matrix multiplication algorithm (Algorithm 1) includes two memory access patterns: by rows (A and C) and by columns (B).

Each row of A matrix is accessed $P$ times and then it is not accessed anymore: the most convenient A cache is composed of a single line which fits a matrix row, which is filled each time a new row is accessed and it hits until the next row is accessed.

Each column of B matrix is accessed $P$ times: the B cache, to get a hit ratio greater than 0 needs to contain at least $M$ lines and comply with the fully-associative mapping policy. The results reported by Ma used a direct-mapped cache with $M$ lines each one containing $P$ elements (so that it is as big as the B matrix).

$C$ elements are accessed sequentially and only once: any single-line cache with $n$ words per line would have a hit ratio of $\frac{n-1}{n}$.

The implementation used during the tests applies both pipelining and unrolling (with factor equal to the number of ports) to the innermost loop.

Algorithm 1 Matrix multiplication algorithm.

Require: $A \in \mathbb{R}^{N \times M}$, $B \in \mathbb{R}^{M \times P}$, $C \in \mathbb{R}^{N \times P}$

Ensure: $C = A \times B$

procedure Multiply($A, B, C$)

for $i = 0, \ldots, N - 1$ do

for $j = 0, \ldots, P - 1$ do

    $tmp \leftarrow 0$

    for $k = 0, \ldots, M - 1$ do

        $tmp \leftarrow tmp + A[i][k] \cdot B[k][j]$

    end for

    $C[i][j] \leftarrow tmp$

end for

end procedure

6.2.1 16x16 matrices

In the case of Matrix multiplication 16x16, matrices $A$, $B$ and $C$ are sized $16 \times 16$ ($N = 16, M = 16, P = 16$).

This problem has been explored in two configurations: Single-level cache configuration (L2 caches only), and Multi-level cache configuration (L2 and L1 caches).

Single-level cache configuration

The cache sizes have been fixed with the values shown in Table 6.2. The get latency and the number of ports have been determined through design space exploration.
6. Results

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Sets</th>
<th>Ways</th>
<th>Words per line</th>
<th>L1 lines</th>
<th>Hit ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>99.6 %</td>
</tr>
<tr>
<td>B</td>
<td>16</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>99.6 %</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>93.8 %</td>
</tr>
</tbody>
</table>

Table 6.2: Single-level cache configuration for *Matrix multiplication 16x16*.

Figure 6.1 shows the execution time with respect to the *get* latency, for different numbers of ports.

It is worth noting that the *get* latency has a big impact on effective performance, especially in the single-port case (one order of magnitude). This makes clear that the cache process itself can run at high speed and the bottleneck is the scheduling of the FIFO accesses.

Increasing the number of ports can provide significant advantages when the *get* latency is not optimal, because multi-port allow to schedule some cache requests in consecutive clock cycles.

![Figure 6.1: Design space of Matrix multiplication 16x16 (single-level).](image)

The best performance is achieved by the single-port, since in this case the caches core process has an II of 1: with a *get* latency of 1 it is not possible to take full advantage of the core pipelining (as explained in Subsection 3.2.1), therefore the design keeps stalling even at the steady state (Figure 6.2a: a new request is written every multiple cycles) but the optimal *get* latency allows to fully exploit the pipelining and at every cycle one request is written and a new response is read (Figure 6.2b).
6. Results

(a) Sub-optimal get latency of 1.

(b) Optimal get latency of 15.

Figure 6.2: Request and response waveforms for Matrix multiplication 16x16 single-level and single-port.

Table 6.3 reports the data for the single-level cache configuration of different port numbers, each one set to its optimal get latency value (15 for the 1-port, 3 for the 2-ports and 2 for the 4-ports).

The single-port configuration fully exploits the underlying single L2 cache, therefore adding more ports not only increase the resource usage, but it also reduces performance since the cache II increases.

It is not clear why the estimated required BRAMs in the 2-ports case is much higher than the other cases.

<table>
<thead>
<tr>
<th>Name</th>
<th>1-port</th>
<th>2-ports</th>
<th>4-ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ns]</td>
<td>17438</td>
<td>38570</td>
<td>49694</td>
</tr>
<tr>
<td>BRAM</td>
<td>90</td>
<td>165</td>
<td>90</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>LUT</td>
<td>57653</td>
<td>87437</td>
<td>118434</td>
</tr>
<tr>
<td>FF</td>
<td>26597</td>
<td>37686</td>
<td>39352</td>
</tr>
</tbody>
</table>

Table 6.3: Performance and resource usage of Matrix multiplication 16x16 (single-level).

Multi-levels cache configuration

The cache sizes have been fixed with the values shown in Table 6.4. The get latency and the number of ports have been determined through design space exploration.
6. Results

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Sets</th>
<th>Ways</th>
<th>Words per line</th>
<th>L1 lines</th>
<th>Hit ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>99.6 %</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>16</td>
<td>99.6 %</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>93.8 %</td>
</tr>
</tbody>
</table>

Table 6.4: Multi-levels cache configuration for Matrix multiplication 16x16.

From Figure 6.3 it is clear that the get latency is not relevant in this case, since all the cache hits are on the L1 cache.

Increasing the number of ports allows to significantly improve performance, since the multiple L1 caches can run effectively in parallel. The higher is the number of ports, the lower is the hit ratio of each L1 cache: indefinitely increasing the number of ports is not always convenient; in this case 4 ports is the optimal configuration in terms of performance. More details about some simulated configurations are reported in Table 6.5.

![Design space of Matrix multiplication 16x16 (multi-levels)](image_url)

Figure 6.3: Design space of Matrix multiplication 16x16 (multi-levels).
6. Results

<table>
<thead>
<tr>
<th></th>
<th>1-port</th>
<th>2-ports</th>
<th>4-ports</th>
<th>8-ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>get latency [μs]</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Execution time [ns]</td>
<td>18986</td>
<td>10166</td>
<td>6458</td>
<td>7358</td>
</tr>
<tr>
<td>BRAM</td>
<td>129</td>
<td>165</td>
<td>237</td>
<td>381</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>LUT</td>
<td>58138</td>
<td>81779</td>
<td>118794</td>
<td>198678</td>
</tr>
<tr>
<td>FF</td>
<td>41961</td>
<td>101315</td>
<td>238374</td>
<td>581204</td>
</tr>
</tbody>
</table>

Table 6.5: Performance and resource usage of Matrix multiplication 16x16 (multi-levels).

Summary

Table 6.6 reports most relevant figures about performance and resource usage of Matrix multiplication 16x16.

The Proposed cache data is referred to the most performant cases of the single-level variant (with single port and get latency equal to 15) and of the multi-levels variant (with 4 ports and get latency equal to 7).

The cost in terms of resource usage of the proposed cache is significant in terms of resource usage, particularly the number of LUTs and FFs is one order of magnitude more for the single-level and single-port configuration and two orders of magnitude for the multi-levels and multi-ports configuration, with respect to global memory. The single-level configuration allows reaching performance on par with the local memory and the multi-level configuration is more than two times faster, thanks to the unrolling.

<table>
<thead>
<tr>
<th></th>
<th>Global memory</th>
<th>Local memory</th>
<th>Proposed cache (single-level)</th>
<th>Proposed cache (multi-levels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ns]</td>
<td>30182</td>
<td>16916</td>
<td>17438</td>
<td>6458</td>
</tr>
<tr>
<td>BRAM</td>
<td>34</td>
<td>90</td>
<td>90</td>
<td>237</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>LUT</td>
<td>4421</td>
<td>26403</td>
<td>57653</td>
<td>118794</td>
</tr>
<tr>
<td>FF</td>
<td>4736</td>
<td>8829</td>
<td>26597</td>
<td>238374</td>
</tr>
</tbody>
</table>

Table 6.6: Performance and resource usage of Matrix multiplication 16x16.
6. Results

6.2.2 32x32 matrices

To check whether the results scale with the problem size, in the case of Matrix multiplication 32x32, matrices A, B and C have been sized 32 × 32 (N = 32, M = 32, P = 32).

Single-level cache configuration

The cache sizes have been fixed with the values shown in Table 6.7. The get latency and the number of ports have been determined through design space exploration.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Sets</th>
<th>Ways</th>
<th>Words per line</th>
<th>L1 lines</th>
<th>Hit ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>99.9</td>
</tr>
<tr>
<td>B</td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>99.9</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>96.9</td>
</tr>
</tbody>
</table>

Table 6.7: Single-level cache configuration for Matrix multiplication 32x32.

From Figure 6.4 it is possible to infer that the shapes of the Execution time - get latency plot of Matrix multiplication 32x32 are equivalent to the ones obtained in the 16x16 case.

![Design space of Matrix multiplication 32x32 (single-level).](image)

Figure 6.4: Design space of Matrix multiplication 32x32 (single-level).

Results reported by Table 6.8 have been obtained by setting the get latency to value which provided the best performance (12 for the single-port cache and 3 for the dual-
port cache). The performance gain obtained by doubling the number of ports is not very significant.

<table>
<thead>
<tr>
<th></th>
<th>1-port</th>
<th>2-ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time  [ns]</td>
<td>265226</td>
<td>229190</td>
</tr>
<tr>
<td>BRAM</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>LUT</td>
<td>126411</td>
<td>175727</td>
</tr>
<tr>
<td>FF</td>
<td>39871</td>
<td>47648</td>
</tr>
</tbody>
</table>

Table 6.8: Performance and resource usage of *Matrix multiplication 32x32* (single-level).

**Multi-levels cache configuration**

The cache sizes have been fixed with the values shown in Table 6.9. Since most of the read accesses are performed to L1 caches (Table 6.10), the *get* latency value does not have a big impact on the performance, therefore it has been fixed to 1.

The number of ports has been kept free for design space exploration.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Sets</th>
<th>Ways</th>
<th>Words per line</th>
<th>L1 lines</th>
<th>get latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.9: Multi-levels cache configuration for *Matrix multiplication 32x32*.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>L2 hit ratio [%]</th>
<th>L1 hit ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.7</td>
<td>99.2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>99.9</td>
</tr>
<tr>
<td>C</td>
<td>96.9</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.10: Multi-levels cache *Matrix multiplication 32x32* hit ratios.
6. Results

<table>
<thead>
<tr>
<th></th>
<th>1-port</th>
<th>2-ports</th>
<th>4-ports</th>
<th>8-ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ns]</td>
<td>138974</td>
<td>71618</td>
<td>39398</td>
<td>30362</td>
</tr>
<tr>
<td>BRAM</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>LUT</td>
<td>113957</td>
<td>140117</td>
<td>183947</td>
<td>291738</td>
</tr>
<tr>
<td>FF</td>
<td>51004</td>
<td>75734</td>
<td>140269</td>
<td>336003</td>
</tr>
</tbody>
</table>

Table 6.11: Performance and resource usage of Matrix multiplication 32x32 (multi-levels).

Summary

Table 6.12 compares the results obtained by different memory access mechanisms. The column Cache is referred to the most performant tested configuration: the multi-levels version with 8 ports, therefore the Matrix multiplication inner loop is unrolled by a factor 8. To make the higher bound comparable, there are two versions of the Local memory report: the first is referred to the case in which the Matrix multiplication inner loop is kept rolled, while in the second case the loop is unrolled with a factor 8 so that it is comparable with cache results.

<table>
<thead>
<tr>
<th></th>
<th>Global memory</th>
<th>Local memory</th>
<th>Local memory (unrolled)</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ns]</td>
<td>389498</td>
<td>131580</td>
<td>16920</td>
<td>30362</td>
</tr>
<tr>
<td>BRAM</td>
<td>34</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>3</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>LUT</td>
<td>4106</td>
<td>30589</td>
<td>121971</td>
<td>291738</td>
</tr>
<tr>
<td>FF</td>
<td>4699</td>
<td>11417</td>
<td>27866</td>
<td>336003</td>
</tr>
</tbody>
</table>

Table 6.12: Performance and resource usage of Matrix multiplication 32x32.

The cache can provide a great performance gain: it is one order of magnitude faster than the global memory and it is almost on par with the ideal case of the unrolled local memory. From Figure 6.5 it is possible to recognize the typical Pareto curve shape, which makes easy to find the desired trade-off: the more resources are employed, the more performance is delivered.
6. Results

Figure 6.5: Pareto curve of Matrix multiplication 32x32.
6.3 Bitonic sorting

Bitonic sorting (Algorithm 2) is a sorting algorithm characterized by a high degree of parallelism, therefore it is suitable for Hardware implementation.

Algorithm 2 Bitonic sorting algorithm.

Require: \( a \in \mathbb{R}^N, N = 2^n; \) \( \text{dir} \): sorting direction

Ensure: \( a[i] \geq a[j], \forall i \geq j \land \text{dir} = \text{true} \lor a[i] \leq a[j], \forall i \geq j \land \text{dir} = \text{false} \)

procedure Sort\((a, \text{dir})\)
  for \( b = 1, \ldots, n \) do
    for \( s = i - 1, \ldots, 0 \) do
      for \( i = 0, \ldots, N/2 - 1 \) do
        \( \text{dir}_0 \leftarrow (i/2^{b-1}) \& 1 \)
        \( \text{dir}_0 \leftarrow \text{dir}_0 | \text{dir} \)
        \( \text{step} \leftarrow 2^s \)
        \( \text{pos} \leftarrow 2i - (i \& (s - 1)) \)
        \( a_0 \leftarrow a[\text{pos}] \)
        \( a_1 \leftarrow a[\text{pos} + \text{step}] \)
        if \( a_0 > a_1 \neq \text{dir}_0 \) then
          \( \text{tmp} \leftarrow a_0 \)
          \( a_0 \leftarrow a_1 \)
          \( a_1 \leftarrow \text{tmp} \)
        end if
        \( a[\text{pos}] \leftarrow a_0 \)
        \( a[\text{pos} + \text{step}] \leftarrow a_1 \)
      end for
    end for
  end for
end procedure

From the memory accesses point of view, each inner loop iteration:

1. \( a[\text{pos}] \) is read.
2. \( a[\text{pos} + \text{step}] \) is read.
3. \( a[\text{pos}] \) is written.
4. \( a[\text{pos} + \text{step}] \) is written.

The cache associated with \( a \) array should be set-associative with at least 2 sets, so that the interleaved accesses to \( \text{pos} \) and \( \text{pos} + \text{step} \) do not overwrite the related cache lines.

In the design under test the inner loop have been pipelined, but, due to the data dependencies on \( a \) array, the pipeline performance is limited (i.e. \( II \) is greater than 1).

Due to the multiple accesses per iteration, it is not possible to exploit: 35
6. Results

- Multiple ports: setting the number of ports to a number greater than one would result in a deadlock in C/RTL Co-Simulation, due to scheduling issues).

- get latency tuning: due to dependencies on $a$, each request to L1 cache is written only when the previous request is read.

For these reasons the get latency has been fixed to 2 (not 1, because it would lead the synthesizer to hang for unknown reasons) and the number of ports to 1.

6.3.1 128 elements

In the case of Bitonic sorting 128, array $a$ contains 128 elements to be sorted ($n = 7$). The Single-level cache configuration is aimed at checking the performance of the L2 cache alone, while the Multi-levels cache configuration has the purpose to evaluate the boost given by adding a L1 cache on top of the L2.

Single-level cache configuration

For the Single-level cache configuration the cache associated with $a$ is a fully-associative cache with two sets, so that the accesses to $pos + step$ index do not interfere with accesses to the $pos$ index. The number of words per cache line has been kept free for design space exploration.

Table 6.13 summarizes the cache configuration.

<table>
<thead>
<tr>
<th>Sets</th>
<th>Ways</th>
<th>L1 lines</th>
<th>get latency</th>
<th>Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.13: Single-level cache configuration for Bitonic sorting 128.

Table 6.14 shows the achieved results, with different number of words per cache line. Doubling the number of words approximately doubles the resource usage, while the performance grows at a much lower rate.

<table>
<thead>
<tr>
<th></th>
<th>8 words</th>
<th>16 words</th>
<th>32 words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit ratio [%]</td>
<td>93.8</td>
<td>96.9</td>
<td>98.4</td>
</tr>
<tr>
<td>Execution time [µs]</td>
<td>232</td>
<td>202</td>
<td>188</td>
</tr>
<tr>
<td>BRAM</td>
<td>16</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>LUT</td>
<td>20294</td>
<td>45481</td>
<td>91656</td>
</tr>
<tr>
<td>FF</td>
<td>6766</td>
<td>14403</td>
<td>24851</td>
</tr>
</tbody>
</table>

Table 6.14: Performance and resource usage of Bitonic sorting 128 (single-level).
Multi-levels cache configuration

The Multi-levels cache configuration (Table 6.15) matches the Single-level cache configuration, with the only difference that a single-line L1 cache has been added on top of the memory hierarchy.

<table>
<thead>
<tr>
<th>Sets</th>
<th>Ways</th>
<th>L1 lines</th>
<th>get latency</th>
<th>Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.15: Multi-levels cache configuration for Bitonic sorting 128.

Table 6.16 shows some results obtained with the Multi-levels cache configuration. Comparing these data with the Single-level cache configuration (Table 6.14) ones it is clear that the insertion of the L1 cache gives a further boost to performance, without increasing much the resource usage.

Execution time of the 16 words case is not reported because the Co-Simulation deadlocks with that specific configuration; its value would be probably between 150 $\mu$s and 170 $\mu$s.

<table>
<thead>
<tr>
<th></th>
<th>8 words</th>
<th>16 words</th>
<th>32 words</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 hit ratio [%]</td>
<td>16.1</td>
<td>19.6</td>
<td>22.3</td>
</tr>
<tr>
<td>L2 hit ratio [%]</td>
<td>77.7</td>
<td>77.2</td>
<td>76.1</td>
</tr>
<tr>
<td>Execution time [\mu s]</td>
<td>194</td>
<td>-</td>
<td>130</td>
</tr>
<tr>
<td>BRAM</td>
<td>16</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>LUT</td>
<td>20745</td>
<td>46246</td>
<td>92740</td>
</tr>
<tr>
<td>FF</td>
<td>8082</td>
<td>18025</td>
<td>33066</td>
</tr>
</tbody>
</table>

Table 6.16: Performance and resource usage of Bitonic sorting 128 (multi-levels).

Summary

Table 6.17 compares the results achieved with the most performant cache configuration (i.e. multi-levels with 32 words per line), with the higher and lower bounds for performance.

Cache is almost two times faster than the global memory, but it’s still one order of magnitude slower than the local memory, and the resource usage is not negligible.
6. Results

<table>
<thead>
<tr>
<th></th>
<th>Global memory</th>
<th>Local memory</th>
<th>Proposed cache</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution time [µs]</strong></td>
<td>244</td>
<td>23</td>
<td>130</td>
</tr>
<tr>
<td><strong>BRAM</strong></td>
<td>2</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td><strong>LUT</strong></td>
<td>1743</td>
<td>3540</td>
<td>92740</td>
</tr>
<tr>
<td><strong>FF</strong></td>
<td>1088</td>
<td>3464</td>
<td>33066</td>
</tr>
</tbody>
</table>

Table 6.17: Performance and resource usage of Bitonic sorting 128.

From Figure 6.6 it is easy to conclude that the best trade-offs (between area and performance) belonging to the Pareto curve are the ones which include the L1 cache.

![Figure 6.6: Pareto curve of Bitonic sorting 128.](image)
6. Results

6.4 2D convolution

Algorithm 3 is a possible implementation of 2D convolution [1].

The algorithm accesses three matrices:

- \( A \): read according to a sliding window pattern.
- \( \text{kernel} \): read sequentially.
- \( B \): written sequentially.

Accesses to \( \text{kernel} \) and \( B \) can be optimized by single-line caches, while \( A \) cache requires a more complex cache to get a sufficiently high hit ratio.

In the Hardware implementation the innermost loop have been pipelined, after moving the assignment to \( B \) in its latest iteration, to make the loop nest perfect (i.e. a loop contains only another loop without any external logic, reducing the HLS effort for pipelining).

**Algorithm 3 2D convolution algorithm.**

**Require:** \( A \in \mathbb{R}^{N \times M}, \text{kernel} \in \mathbb{R}^{P \times Q} \)

**Ensure:** \( B \in \mathbb{R}^{N \times M}; B = A \ast \text{kernel} \)

**procedure** CONV(\( A, \text{kernel} \))

\[
\text{for } i = 0, \ldots, N - 1 \text{ do} \\
\quad \text{for } j = 0, \ldots, M - 1 \text{ do} \\
\quad \quad \text{tmp} \leftarrow 0 \\
\quad \quad \text{for } m = 0, \ldots, P - 1 \text{ do} \\
\quad \quad \quad \text{for } n = 0, \ldots, Q - 1 \text{ do} \\
\quad \quad \quad \quad ii \leftarrow i + (Q/2 - m) \\
\quad \quad \quad \quad jj \leftarrow j + (P/2 - n) \\
\quad \quad \quad \quad \text{if } ii \geq 0 \& ii < N \& jj \geq 0 \& jj < M \text{ then} \\
\quad \quad \quad \quad \text{tmp} \leftarrow \text{tmp} + A[ii][jj] \cdot \text{kernel}[m][n] \\
\quad \quad \quad \text{end if} \\
\quad \quad \text{end for} \\
\quad \quad B[ii][jj] \leftarrow \text{tmp} \\
\quad \text{end for} \\
\text{end for} \\
\text{end procedure}
\]

6.4.1 32x32 matrix and 9x9 kernel

The design under test has been set such that \( A, B \in \mathbb{N}^{32 \times 32} \) and the \( \text{kernel} \in \mathbb{N}^{3 \times 3} \).

The cache associated with \( A \) matrix is fully associative with 4 sets, to ensure that consecutive cache accesses do not overwrite previously loaded line. The other parameters have been left free for design space exploration.
6. Results

_kernel_ cache has been set to work as a line buffer: it contains a single line which fits the whole kernel, so that the first request load the kernel to cache and all the subsequent accesses are L1 cache hits.

_B_ cache has the purpose of gathering multiple write requests.

The full configuration of caches is shown in Table 6.18.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Sets</th>
<th>Ways</th>
<th>Words per line</th>
<th>L1 lines</th>
<th>Hit ratio [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>4</td>
<td>Free</td>
<td>Free</td>
<td>?</td>
</tr>
<tr>
<td>kernel</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>0</td>
<td>96.9</td>
</tr>
</tbody>
</table>

Table 6.18: kernel and B caches configuration for 2D convolution.

Single-level cache configuration

With the _Single-level cache configuration_ the A cache is configured to have no L1 cache, while the number of words per line is kept free. The _get_ latency has been set to 9 for both A and kernel caches.

Table 6.19 reports the collected results. The execution time with 32 words is missing since this configuration causes a deadlock in Co-Simulation.

<table>
<thead>
<tr>
<th></th>
<th>8 words</th>
<th>16 words</th>
<th>32 words</th>
</tr>
</thead>
<tbody>
<tr>
<td>A hit ratio [%]</td>
<td>89.6</td>
<td>95.8</td>
<td>99.6</td>
</tr>
<tr>
<td>Execution time [µs]</td>
<td>45</td>
<td>40</td>
<td>-</td>
</tr>
<tr>
<td>BRAM</td>
<td>146</td>
<td>174</td>
<td>174</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>LUT</td>
<td>85574</td>
<td>91190</td>
<td>102887</td>
</tr>
<tr>
<td>FF</td>
<td>33803</td>
<td>37779</td>
<td>44578</td>
</tr>
</tbody>
</table>

Table 6.19: Performance and resource usage of 2D convolution (single-level).

Multi-levels cache configuration

The _Multi-levels cache configuration_ is equivalent to the _Single-level_, but a single-line L1 cache has been added.

The performance advantage with respect to the _Single-level configuration_ is negligible, therefore it is not convenient to invest in more resources in this case.
### 6. Results

<table>
<thead>
<tr>
<th></th>
<th>8 words</th>
<th>16 words</th>
<th>32 words</th>
</tr>
</thead>
<tbody>
<tr>
<td>A L1 hit ratio [%]</td>
<td>59.6</td>
<td>63.8</td>
<td>66.0</td>
</tr>
<tr>
<td>A L2 hit ratio [%]</td>
<td>30.1</td>
<td>32.0</td>
<td>33.7</td>
</tr>
<tr>
<td>Execution time [µs]</td>
<td>43</td>
<td>40</td>
<td>39</td>
</tr>
<tr>
<td>BRAM</td>
<td>146</td>
<td>174</td>
<td>174</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>LUT</td>
<td>85640</td>
<td>91274</td>
<td>102970</td>
</tr>
<tr>
<td>FF</td>
<td>40288</td>
<td>50146</td>
<td>68720</td>
</tr>
</tbody>
</table>

Table 6.20: Performance and resource usage of 2D convolution (multi-levels).

**Summary**

Table 6.21 compares the most performant cache configuration (i.e. multi-levels with 32 words per line) with the performance bounds: the achieved performance is almost equal to the performance higher bound.

<table>
<thead>
<tr>
<th></th>
<th>Global memory</th>
<th>Local memory</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [µs]</td>
<td>66</td>
<td>37</td>
<td>39</td>
</tr>
<tr>
<td>BRAM</td>
<td>12</td>
<td>178</td>
<td>174</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>LUT</td>
<td>3504</td>
<td>30897</td>
<td>102970</td>
</tr>
<tr>
<td>FF</td>
<td>3257</td>
<td>10672</td>
<td>68720</td>
</tr>
</tbody>
</table>

Table 6.21: Performance and resource usage of Convolution.

The plot shown in Figure 6.7 suggests that the single-level 16 words per line configuration may be the most convenient in this case, since it offers almost the same performance and requires almost half of the FFs with respect to the multi-levels 32 words configuration.
6. Results

Figure 6.7: Pareto curve of 2D convolution.
7 Conclusions

This thesis work proven the possibility to implement multi-process designs in HLS. The cache process itself can provide high performance thanks to its optimal pipelining (i.e. II of 1), but some issues arise from the proposed Inter-Process Communication protocol: *Vitis HLS 2021.1* do not provide any mechanism to inform the scheduler about the dependencies and latency between the request writing to a FIFO and the response reading from another FIFO, therefore the accesses to these data structures are not optimally scheduled by HLS, resulting in performance loss and in some cases even deadlocks.

The proposed workaround of forcing clock cycles to increase the latency between the FIFO write and the FIFO read allows to achieve optimal results in some specific situations, but it can not always work.

The collected results prove that in general the resource usage required by the proposed cache module is paid off in terms of performance, without requiring high design efforts for optimizing memory interfacing: it is enough to include the cache in the kernel and setup the parameters according to the desired performance and resource usage. In some cases it is possible to achieve very high performance gain (e.g. *Matrix multiplication* achieved a reduction of execution time of more than one order of magnitude).

To overcome the limitations posed by the HLS tool it may be worth implementing the interface with the cache or the whole architecture at RTL, which provides full control on the operations scheduling.

The cache performance could be further improved by implementing a pre-fetching mechanism which analyses the memory access patterns and tries to load in advance the data, before it is requested.
A Source code

The source code produced during this thesis work is fully open source and can be found in the git repository hosted to https://github.com/brigio345/hls_cache.

The src directory contains the cache class and the related dependencies.

The test directory contains some test programs, including those used to collect the reported results; each of them can be synthesized and simulated in Vitis HLS 2021.1 by executing the related script stored to scripts.

A.1 Cache

Listing A.1: Source code of src/cache.h

```c
#ifndef CACHE_H
#define CACHE_H

/**
 * \file cache.h
 * \brief Cache module compatible with Vitis HLS 2021.1.
 *
 * Cache module whose characteristics are:
 * - address mapping: set-associative;
 * - replacement policy: least-recently-used or last-in first-out;
 * - write policy: write-back.
 *
 * Advanced features:
 * - Multi-levels: L1 cache (direct-mapped, write-through).
 * - Multi-ports (read-only).
 */

#include "address.h"
#include "replacer.h"
#include "l1_cache.h"
#define HLS_STREAM_THREAD_SAFE
#include "hls_stream.h"
#include "stream_cond.h"
#include "ap_utils.h"
#include "ap_int.h"
```
# include "utils.h"
#ifdef __SYNTHESIS__
#include "hls_vector.h"
#else
#include <thread>
#include <array>
#include <cassert>
#endif /* __SYNTHESIS__ */
#define MAX_AXI_BITWIDTH 512

template <typename T, bool RD_ENABLED, bool WR_ENABLED, size_t PORTS,
size_t MAIN_SIZE, size_t N_SETS, size_t N_WAYS,
size_t N_WORDS_PER_LINE, bool LRU,
size_t L1_CACHE_LINES, size_t LATENCY>
class cache {
private:
  static const bool MEM_IF_PROCESS = (WR_ENABLED || (PORTS > 1) ||
    ((sizeof(T) * N_WORDS_PER_LINE * 8) > MAX_AXI_BITWIDTH));
  static const bool L1_CACHE = (L1_CACHE_LINES > 0);
  static const size_t ADDR_SIZE = utils::log2_ceil(MAIN_SIZE);
  static const size_t SET_SIZE = utils::log2_ceil(N_SETS);
  static const size_t OFF_SIZE = utils::log2_ceil(N_WORDS_PER_LINE);
  static const size_t TAG_SIZE = (ADDR_SIZE - (SET_SIZE + OFF_SIZE));
  static const size_t WAY_SIZE = utils::log2_ceil(N_WAYS);
  static_assert((RD_ENABLED || WR_ENABLED),
    "RD_ENABLED and/or WR_ENABLED must be true");
  static_assert((PORTS > 0), "PORTS must be greater than 0");
  static_assert(((WR_ENABLED && (PORTS > 1)),
    "PORTS must be equal to 1 when WR_ENABLED is true");
  static_assert((MAIN_SIZE > 0) && ((1 << ADDR_SIZE) == MAIN_SIZE),
    "MAIN_SIZE must be a power of 2 greater than 0");
  static_assert((N_SETS > 0) && ((1 << SET_SIZE) == N_SETS),
    "N_SETS must be a power of 2 greater than 0");
  static_assert((N_WAYS > 0) && ((1 << WAY_SIZE) == N_WAYS),
    "N_WAYS must be a power of 2 greater than 0");
  static_assert((N_WORDS_PER_LINE > 0) &&
    ((1 << OFF_SIZE) == N_WORDS_PER_LINE),
    "N_WORDS_PER_LINE must be a power of 2 greater than 0");
  static_assert((MAIN_SIZE >= (N_SETS * N_WAYS * N_WORDS_PER_LINE)),
    "N_SETS and/or N_WAYS and/or N_WORDS_PER_LINE are too big \n    for the specified MAIN_SIZE");
#else
  template <typename TYPE, size_t SIZE>
  using array_type = hls::vector<TYPE, SIZE>;
#else
  template <typename TYPE, size_t SIZE>
  using array_type = std::array<TYPE, SIZE>;
#endif /* __SYNTHESIS__ */

typedef address<ADDR_SIZE, TAG_SIZE, SET_SIZE, WAY_SIZE>
A. Source code

```c
address_type;
typedef array_type<T, N_WORDS_PER_LINE> line_type;
typedef l1_cache<T, MAIN_SIZE, L1_CACHE_LINES, N_WORDS_PER_LINE> l1_cache_type;
typedef replacer<LRU, address_type, N_SETS, N_WAYS, N_WORDS_PER_LINE> replacer_type;

typedef enum {
  READ_OP,
  WRITE_OP,
  READ_WRITE_OP,
  STOP_OP,
  NOP_OP
} op_type;

#if ( defined(PROFILE) && (! defined(__SYNTHESIS__)))
typedef enum {
  MISS,
  HIT,
  L1_HIT
} hit_status_type;
#endif /* ( defined(PROFILE) && (! defined(__SYNTHESIS__))) */

typedef struct {
  op_type op;
  ap_uint<ADDR_SIZE> load_addr;
  ap_uint<ADDR_SIZE> write_back_addr;
  line_type line;
} mem_req_type;

ap_uint<( TAG_SIZE > 0) ? TAG_SIZE : 1> m_tag[N_SETS * N_WAYS];
bool m_valid[N_SETS * N_WAYS];
bool m_dirty[N_SETS * N_WAYS];
T m_cache_mem[N_SETS * N_WAYS * N_WORDS_PER_LINE];
hls::stream<op_type, 4> m_core_req_op[PORTS];
hls::stream<ap_uint<ADDR_SIZE>, 4> m_core_req_addr[PORTS];
hls::stream<T, 4> m_core_req_data[PORTS];
hls::stream<line_type, 4> m_core_resp[PORTS];
stream_cond<mem_req_type, 2, MEM_IF_PROCESS> m_mem_req;
stream_cond<line_type, 2, MEM_IF_PROCESS> m_mem_resp;
l1_cache_type m_l1_cache_get[PORTS];
replacer_type m_replacer;
unsigned int m_core_port;

#pragma HLS array_partition variable=m_tag complete dim=1
```

46
```c
#pragma HLS array_partition variable = m_valid complete dim = 1
#pragma HLS array_partition variable = m_dirty complete dim = 1
#pragma HLS array_partition variable = m_cache_mem cyclic factor = N_WORDS_PER_LINE dim = 1
#pragma HLS array_partition variable = m_core_req_op complete
#pragma HLS array_partition variable = m_core_req_addr complete
#pragma HLS array_partition variable = m_core_req_data complete
#pragma HLS array_partition variable = m_core_resp complete
#pragma HLS array_partition variable = m_l1_cache_get complete

/**
 * brief Initialize the cache.
 * note Must be called before calling \ref run.
 */
void init() {
    m_core_port = 0;
    if (L1_CACHE) {
        for (auto port = 0; port < PORTS; port++)
            m_l1_cache_get[port].init();
    }
}

/**
 * brief Start cache internal processes.
 * param[in] main_mem The pointer to the main memory.
 * note In case of synthesis this must be called in a dataflow region with disable_start_propagation option,
 * together with the function in which cache is accessed.
 * note In case of C simulation this must be executed by a thread separated from the thread in which cache is accessed.
 */
void run(T * const main_mem) {
#pragma HLS inline
#ifdef __SYNTHESIS__
    run_core(main_mem);
    if (MEM_IF_PROCESS)
        run_mem_if(main_mem);
#else
    std::thread core_thd([&]{ run_core(main_mem);});
    if (MEM_IF_PROCESS) {
        std::thread mem_if_thd([&] { run_mem_if(main_mem); });
    }
    mem_if_thd.join();
#endif
```
} 

core_thd.join();
#endif /* __SYNTHESIS__ */
}

/** 
 * \brief Stop cache internal processes.
 * 
 * \note Must be called after the function in which cache 
 * is accessed has completed.
 */
void stop() {
  for (auto port = 0; port < PORTS; port++)
    m_core_req_op[port].write(STOP_OP);
}

/** 
 * \brief Request to read a whole cache line.
 * 
 * \param [in] addr_main The address in main memory belonging to 
 * the cache line to be read.
 * \param [out] line The buffer to store the read line.
 */
void get_line(const ap_uint<ADDR_SIZE> addr_main, line_type &line) {
  #pragma HLS inline
  #ifndef __SYNTHESIS__
  assert(addr_main < MAIN_SIZE);
  #endif /* __SYNTHESIS__ */

  const auto port = m_core_port;
  m_core_port = ((m_core_port + 1) % PORTS);

  // try to get line from L1 cache
  const auto l1_hit = (L1_CACHE &&
    m_l1_cache_get[port].hit(addr_main));

  if (l1_hit) {
    m_l1_cache_get[port].get_line(addr_main, line);
  } else {
    // send read request to cache
    m_core_req_op[port].write(READ_OP);
    m_core_req_addr[port].write(addr_main);
    // force FIFO write and FIFO read to separate 
    // pipeline stages to avoid deadlock due to
    // the blocking read
    ap_wait_n(LATENCY);
    // read response from cache
    m_core_resp[port].read(line);
  }
}
if (L1_CACHE) {
    // store line to L1 cache
    m_l1_cache_get[port].set_line(addr_main, line);
}

#if (defined(PROFILE) && (!defined(__SYNTHESIS__)))
update_profiling(l1_hit ? L1_HIT : m_hit_status.read);
#endif /* (defined(PROFILE) && (!defined(__SYNTHESIS__))) */

/**
 * \brief Request to read a data element.
 *
 * \param[in] addr_main The address in main memory referring to
 * the data element to be read.
 * \return The read data element.
 */
T get(const ap_uint<ADDR_SIZE> addr_main) {
    line_type line;

    // get the whole cache line
    get_line(addr_main, line);

    // extract information from address
    address_type addr(addr_main);

    return line[addr.m_off];
}

/**
 * \brief Request to write a data element.
 *
 * \param[in] addr_main The address in main memory referring to
 * the data element to be written.
 * \param[in] data The data to be written.
 */
void set(const ap_uint<ADDR_SIZE> addr_main, const T data) {
    #ifdef __SYNTHESIS__
    assert(addr_main < MAIN_SIZE);
    #endif /* __SYNTHESIS__ */

    if (L1_CACHE) {
        // inform L1 caches about the writing
        m_l1_cache_get[0].notify_write(addr_main);
    }

    // send write request to cache
    m_core_req_op[0].write(WRITE_OP);
    m_core_req_addr[0].write(addr_main);
m_core_req_data[0].write(data);

#if (defined(PROFILE) && (!defined(__SYNTHESIS__)))
  update_profiling(m_hit_status.read());
#endif /* (defined(PROFILE) && (!defined(__SYNTHESIS__))) */

#else
  int get_n_reqs() const {
    return m_n_reqs;
  }
  int get_n_hits() const {
    return m_n_hits;
  }
  int get_n_l1_hits() const {
    return m_n_l1_hits;
  }
  double get_hit_ratio() const {
    if (m_n_reqs > 0)
      return ((m_n_hits + m_n_l1_hits) /
                static_cast<double>(m_n_reqs));
    return 0;
  }
#endif /* (defined(PROFILE) && (!defined(__SYNTHESIS__))) */

private:
  /**
   * \brief Infinite loop managing the cache access
   * requests (sent from the outside).
   *
   * \param [in] main_mem The pointer to the main memory (ignored
   * if \ref MEM_IF_PROCESS is \c true).
   *
   * \note The infinite loop must be stopped by
   * calling \ref stop (from the outside)
   * when all the accesses have been completed.
   */
  #pragma HLS inline off
  void run_core(T * const main_mem) {
    // invalidate all cache lines
    for (auto line = 0; line < (N_SETS * N_WAYS); line++)
      m_valid[line] = false;
    m_replacer.init();

    #pragma HLS pipeline II=PORTS
    INNER_CORE_LOOP: for (auto port = 0; port < PORTS; port++) {
      op_type op;
      #ifndef __SYNTHESIS__
// get request and
// make pipeline flushable (to avoid deadlock)
if (m_core_req_op[port].read_nb(op)) {
  #else
    // get request
    m_core_req_op[port].read(op);
  #endif /* __SYNTHESIS__ */
  // exit the loop if request is "end-of-request"
  if (op == STOP_OP)
    goto core_end;
  #ifndef __SYNTHESIS__
  if (op == NOP_OP)
    continue;
  #endif /* __SYNTHESIS__ */
  // check the request type
  const auto read = ((RD_ENABLED && (op == READ_OP)) || (!WR_ENABLED));
  // in case of write request, read data to be written
  const auto addr_main = m_core_req_addr[port].read();
  T data;
  if (!read)
    data = m_core_req_data[port].read();
  // extract information from address
  address_type addr(addr_main);
  auto way = hit(addr);
  const auto is_hit = (way != -1);
  if (!is_hit)
    way = m_replacer.get_way(addr);
  addr.set_way(way);
  m_replacer.notify_use(addr);
  line_type line;
  if (is_hit) {
    // read from cache memory
    get_line(m_cache_mem, addr.m_addr_cache, line);
  } else {
    // read from main memory
    auto op = READ_OP;
    // build write-back address
    address_type write_back_addr(m_tag[addr.m_addr_line],
      addr.m_set,
      0, addr.m_way);
    // check if write back is necessary

if (WR_ENABLED && m_valid[addr.m_addr_line] &&
    m_dirty[addr.m_addr_line]) {
    // get the line to be written back
    get_line(m_cache_mem,
             write_back_addr.m_addr_cache,
             line);

    op = READ_WRITE_OP;
}

const mem_req_type req = {op, addr.m_addr_main,
                          write_back_addr.m_addr_main, line};

if (MEM_IF_PROCESS) {
    // send read request to
    // memory interface and
    // write request if
    // write-back is necessary
    m_mem_req.write(req);

    // force FIFO write and
    // FIFO read to separate
    // pipeline stages to
    // avoid deadlock due to
    // the blocking read
    ap_wait();

    // read response from
    // memory interface
    m_mem_resp.read(line);
} else {
    execute_mem_if_req(main_mem,
                        req, line);
}

m_tag[addr.m_addr_line] = addr.m_tag;
m_valid[addr.m_addr_line] = true;
m_dirty[addr.m_addr_line] = false;

m_replacer.notify_insertion(addr);

if (read) {
    // store loaded line to cache
    set_line(m_cache_mem,
             addr.m_addr_cache,
             line);
}

if (read) {
    // send the response to the read request
    m_core_resp[port].write(line);
} else {
// modify the line
line[addr.m_off] = data;

// store the modified line to cache
set_line(m_cache_mem,
         addr.m_addr_cache,
         line);

    m_dirty[addr.m_addr_line] = true;
}

#if (defined(PROFILE) && (!defined(__SYNTHESIS__)))
    m_hit_status.write(is_hit ? HIT : MISS);
#endif /* (defined(PROFILE) && (!defined(__SYNTHESIS__))) */
#endif /* __SYNTHESIS__ */

core_end:
    // synchronize main memory with cache memory
if (WR_ENABLED)
    flush();

    // make sure that flush has completed before stopping
    // memory interface
ap_wait();

    if (MEM_IF_PROCESS) {
        // stop memory interface
        line_type dummy;
        m_mem_req.write({STOP_OP, 0, 0, dummy});
    }

/**
 * brief Infinite loop managing main memory access requests (sent from \ref run_core).
 *
 * \param[in] main_mem The pointer to the main memory.
 *
 * \note \p main_mem must be associated with a dedicated AXI port.
 *
 * \note The infinite loop is stopped by \ref run_core when it is in turn stopped from the outside.
 */
void run_mem_if(T * const main_mem) {
    #pragma HLS inline off
}
MEM_IF_LOOP: while (1) {

#pragma HLS pipeline off
mem_req_type req;
   // get request
m_mem_req.read(req);

   // exit the loop if request is "end-of-request"
if (req.op == STOP_OP)
   break;

line_type line;
execute_mem_if_req(main_mem, req, line);

if ((req.op == READ_OP) || (req.op == READ_WRITE_OP)) {
   // send the response to the read request
   m_mem_resp.write(line);
}

} /*
*brief   Execute memory access(es) specified in
*p
*    
*    
*    
*param[in] main_mem   The pointer to the main memory.
*param[in] req The request to be executed.
*param[out] line The buffer to store the read line.
*note    \p main_mem must be associated with
*        
*        
*/
void execute_mem_if_req(T * const main_mem,
            const mem_req_type &req, line_type & line) {
#pragma HLS inline
if ((req.op == READ_OP) || (req.op == READ_WRITE_OP)) {
    // read line from main memory
    get_line(main_mem, req.load_addr, line);
}

if (WR_ENABLED && ((req.op == WRITE_OP) ||
   (req.op == READ_WRITE_OP))) {
    // write line to main memory
    set_line(main_mem, req.write_back_addr, req.line);
}

} /*
*brief   Check if \p addr causes an HIT or a MISS.
*    
*    
*    
*param[in] addr The address to be checked.
*    
*    
*/
return  hitting way on HIT.
inline int hit(const address_type &addr) const {
    auto addr_tmp = addr;
    auto hit_way = -1;
    for (auto way = 0; way < N_WAYS; way++) {
        addr_tmp.set_way(way);
        if (m_valid[addr_tmp.m_addr_line] &&
            (addr_tmp.m_tag == m_tag[addr_tmp.m_addr_line])) {
            hit_way = way;
        }
    }
    return hit_way;
}

/**
 * \brief Write back all valid dirty cache lines to main memory.
 */
void flush() {
    for (auto set = 0; set < N_SETS; set++) {
        for (auto way = 0; way < N_WAYS; way++) {
            const address_type addr(
                m_tag[set * N_WAYS + way],
                set, 0, way);
            if (m_valid[addr.m_addr_line] &&
                m_dirty[addr.m_addr_line]) {
                // write line back
                line_type line;

                // send write request to memory
                // interface
                m_mem_req.write({WRITE_OP, 0,
                    addr.m_addr_main,
                    line});
                m_dirty[addr.m_addr_line] = false;
            }
        }
    }
}

void get_line(const T * const mem,
    const ap_uint<(ADDR_SIZE > 0) ? ADDR_SIZE : 1> addr,
    line_type &line) {
#pragma HLS inline
const T * const mem_line = &( mem[addr & (-1U << OFF_SIZE)]);

for (auto off = 0; off < N_WORDS_PER_LINE; off++) {
#pragma HLS unroll
    line[off] = mem_line[off];
}

void set_line(T * const mem,
             const ap_uint<(ADDR_SIZE > 0) ? ADDR_SIZE : 1> addr,
             const line_type & line) {
#pragma HLS inline
    T * const mem_line = &(mem[addr & (-1U << OFF_SIZE)]);
    for (auto off = 0; off < N_WORDS_PER_LINE; off++) {
#pragma HLS unroll
        mem_line[off] = line[off];
    }
}

#if (defined(PROFILE) && (!defined(__SYNTHESIS__)))
    void update_profiling(const hit_status_type status) {
        m_n_reqs ++;
        if (status == HIT)
            m_n_hits ++;
        else if (status == L1_HIT)
            m_n_l1_hits ++;
    }
#endif /* (defined(PROFILE) && (!defined(__SYNTHESIS__))) */

class square_bracket_proxy {
    private:
        cache *m_cache;
        const ap_uint<ADDR_SIZE> m_addr_main;
    public:
        square_bracket_proxy(cache *c,
                              const ap_uint<ADDR_SIZE> addr_main):
            m_cache(c), m_addr_main(addr_main) {
#pragma HLS inline
        }

    #pragma HLS inline
    operator T() const {
#pragma HLS inline
        return get();
    }

    square_bracket_proxy & operator=(const T data) {
#pragma HLS inline
        set(data);
        return *this;
    }
A. Source code

```c++

square_bracket_proxy & operator=(
    const square_bracket_proxy &proxy) {
    #pragma HLS inline
    set(proxy.get());
    return *this;
}

private:
    T get() const {
    #pragma HLS inline
    return m_cache->get(m_addr_main);
    }

    void set(const T data) {
    #pragma HLS inline
    m_cache->set(m_addr_main, data);
    }

public:
    square_bracket_proxy operator[](const ap_uint<ADDR_SIZE> addr_main) {
    #pragma HLS inline
    return square_bracket_proxy(this, addr_main);
    }
}

#define /* CACHE_H */
```
Bibliography


