POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

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

Supervisor
Prof. Luciano Lavagno

Candidate
Giovanni Brignone
ID: 274148

Academic year 2020-2021

Abstract

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

High-Level Synthesis is aimed at reducing design efforts, shrinking the gap between hardware and software design.

FPGAs allow to reduce deployment costs, making possible to implement special-purpose hardware modules on general-purpose underlying architecture.

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

Contents

Chapter 1

Background

1.1 Cache memory

Memory devices are usually the performance bottleneck in the execution of memory-bound 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 setup a memory hierarchy in which fast but small memories are paired with large but slow memories, which allows to get good performance on average while containing costs.

This hierarchy can be managed by two main approaches:

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

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

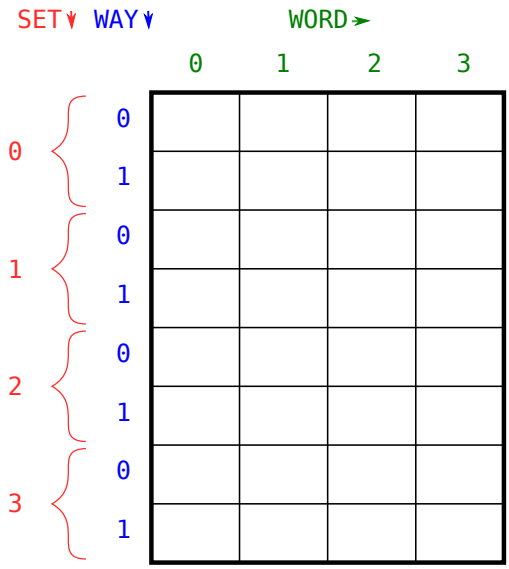

Figure 1.1: Cache logic structure.

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.

Policies

Mapping policy

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

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

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

Special cases of this policy are:

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

Figure 1.2: Set associative policy address bits meaning.

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

Replacement policy

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

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

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

Consistency policy

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

The most common solutions to this problem are:

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

1.2 High-Level Synthesis

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

HLS allows to design more complex systems in less time, compared to HDL design, moreover makes the hardware and software co-design much easier, at the cost of less expressiveness.

Workflow

The typical HLS workflow consists in:

1. software 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. software 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. hardware synthesis: the synthesizer generates an RTL description of the top level entity. It is possible to generate different architectures by setting up some parameters through dedicated directives.
4. hardware verification: the RTL description is simulated, to make sure that SW and HW outputs match.

Optimization techniques

Typical optimization techniques used by HLS for improving performance include:

- pipelining: loops and functions logic can be pipelined so that successive iterations/calls can start while previous ones are still running. The introduced parallelism allows to increase the throughput at a limited additional area cost (only pipeline registers and an FSM are required).
- dataflow: different functions composing the design are called in a pipelined fashion (similarly to pipelining, but at task level, instead of instruction level).
- loop unrolling: the loop logic is instantiated multiple times, to execute multiple loop iterations in parallel, reducing latency and improving throughput.
- memory optimizations
  - bursting: multiple memory accesses are aggregated to reduce overall latency and improving throughput.
  - interface widening: multiple data elements are packed into a single bigger word, to perform multiple accesses at the same time.

1.3 FPGAs

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

Memory system

An FPGA memory system is typically made up of:

- registers: the fastest but most expensive memories, therefore there are only a few.
- block RAMs: on chip RAMs accessible through simple and fast interface.
- external DRAM: off chip RAMs through complex and slow interface (e.g. AXI).

1.4 Previous work

Chapter 2

Architecture

2.1 Basic architecture

The fundamental idea behind the proposed architecture is to keep application and cache logic into separate processes, in order to simplify synthesizer job and possibly obtain a better performing circuit, as shown in Figure 2.1.

When application needs to access memory:

1. application writes the request to the request FIFO.
2. cache reads the request FIFO and checks if it causes a miss
3. in case of miss, cache issues a request to the AXI interface to prepare its own memory for fulfilling the requested access.
4. cache performs the access to its own memory and writes the outcome to the response FIFO (in case of a read request).

Single-process solution

The straightforward solution is to design the cache as a single process, but the synthesizer would generate a pipeline which includes the AXI interfacing and any request, no matter if it is an hit, has to pass through it, cancelling any performance advantage from the cache.

Multi-process solution

The desirable cache architecture should be made up of a short pipeline which runs at full speed in case of hit and stalls in case of miss. To make the synthesizer generate such an architecture, the cache have been split into two processes (Figure 2.2):

- core: it is in charge of managing communication with application and keeping cache data structures up to date.

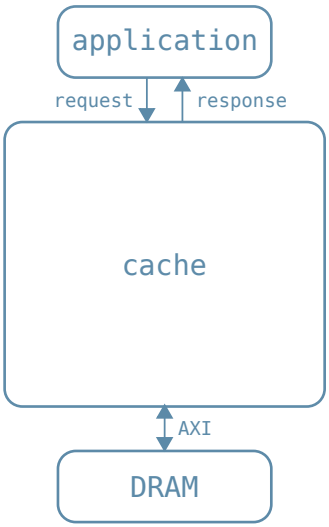

Figure 2.1: Outer view of the basic architecture of a kernel using the cache.

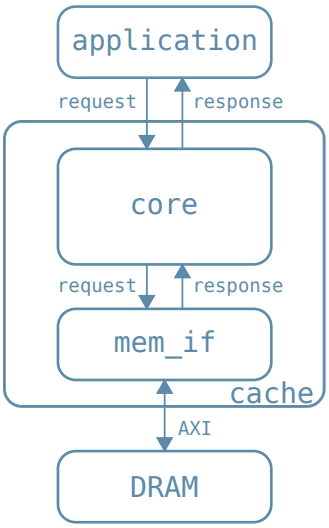

Figure 2.2: Inner view of the basic architecture of a kernel using the cache.

- memory interface: it is in charge of interfacing with the AXI interface.

This solution makes the core pipeline shorter since it does not have to deal with AXI interfacing and in case of miss the pipeline stalls thanks to the blocking read of the memory interface response.

2.2 Optimizations

L1 cache

Multiple read ports

Chapter 3

Implementation

3.1 Multi-process modeling

Process modeling

Communication modeling

3.2 RAW dependencies

Chapter 4

Results

Chapter 5

Conclusion