UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

telecos BCN

Integration of a High Performance Cache with a RISC-V Core and Cost-Benefit Analysis

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Arnau Bigas Soldevila

In partial fulfillment
of the requirements for the
Master in Advanced Telecommunications Technologies

Advisors:
Francesc Moll (Universitat Politècnica de Catalunya, Barcelona Supercomputing Center),
Lluc Alvarez (Barcelona Supercomputing Center)
Barcelona, Date 22/10/2023

telecos BCN

Contents

List of Figures

List of Tables

Revision history and approval record

Revision	Date	Purpose
0	22/10/2023	Document creation

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Arnau Bigas	
Francesc Moll	
Lluc Alvarez	

Written by:		Reviewed and approved by:	
Date	22/10/2023	Date	22/10/2023
Name	Arnau Bigas	Name	Lluc Alvarez
Position	Project Author	Position	Project Supervisor

Abstract

It is widely known that memory operations have the highest latency in modern processors. Because of the high usage of these operations in many types of programs, it is crucial for a system to effectively tolerate this latency. To this end, many high performance processors have a highly complex and robust memory hierarchy. Arguably, one of the most important elements of the hierarchy are the lower level caches, which are the ones that are the closest to the core and directly interfacing with it.

This work consists on integrating an existing core with a high performance L1 data cache. At first, an initial implementation is done, assuring the full functionality and the ability to provide a Linux-capable core via the boot of the operating system in an FPGA. Afterwards, a series of incremental optimizations to that implementation are presented. These optimizations are later evaluated on several metrics, such as IPC, maximum frequency, area and power, using synthesis on a 7nm technology node. Using these metrics, a final design is chosen which achieves a frequency of 1.45GHz and is 4% better than the original design in IPC.

1 Introduction and Motivation

In the last decades, memory has not scaled down as fast as logic with newer technology nodes. In fact, while transistor counts in processors keep doubling every two years –with its consequent performance increase–, memory capacity only doubles every 3 years and, more importantly, latency has gone down by less than 25% in 10 years [6]. This trend has caused a large gap between the performance of processors and of memories, known as the Memory Wall. Given the importance of memory operations in High Performance Computing (HPC) workloads, which represent an average 30-35% of the dynamic operations performed [7], mitigating the memory access bottleneck has become critical in HPC processors. To this end, virtually all HPC processors include multiple levels of memories, which is known as the memory hierarchy.

The upper level of the memory hierarchy is the main memory and, from there down, it includes multiple levels of cache memories (usually two or three) all the way down to the cores. In this hierarchy, one of the most important pieces is the L1 data cache. This lower level cache directly interfaces with the core, and it is the responsible for hiding most of the memory access latency. To achieve high performance, the L1 data cache needs to be fast, i.e., it needs to (a) have a low access latency, (b) have a high hit-rate, (c) be non-blocking for high throughput, (d) allow for many operations in flight, to allow the processor to continue execution. In short, a data cache is a complex and key element of the memory hierarchy.

Thanks to the proliferation of open-source hardware, sparked by the every increasing popularity of RISC-V, some cache designs have started to emerge to support cores for this Instruction Set Architecture (ISA). This is the case of the High Performance Data Cache (HPDCache), an L1 data cache designed by CEA. The HPDCache is geared towards HPC workloads and ticks all the previously mentioned checks to provide high performance. In addition, the HPDCache is open-source and it has been integrated into several HPC processor prototypes, so it is has been thoroughly tested and is silicon-proven. Thus, the HPDCache is a prime candidate to be paired with Sargantana, an in-house RISC-V processor design developed at the Barcelona Supercomputing Center.

The main goal of this project is to integrate the HPDCache with Sargantana. This integration consists on adding the HPDCache into the top level design, modifying the existing interfaces of Sargantana to work with the HPDCache, and adapting the finite state machine (FSM) and control logic of the memory unit to work with the new protocol. Furthermore, Sargantana does the virtual-to-physical memory address translation and exception checking in the legacy cache, so these two pieces of logic have to be extracted form the legacy cache and integrated into the memory unit. Finally, some performance optimizations have also been performed in the new design with the HPD-Cache in order to increase the overall performance of the system.

1.1 Objectives

1. Integrate the High Performance Data Cache (HPDCache) with Sargantana, focusing first on functionality.
   (a) Enable the possibility of having many memory requests in flight.
   (b) Verify the integration using the standard ISA tests and random test generators with a high memory instruction count to stress-test the system.
   (c) Develop an FPGA prototype that is able to successfully boot Linux.
2. Modify the initial integration improving the initial performance, proposing incremental optimizations.
3. Evaluate the different optimizations and analyze their performance, area requirements and power consumption.
   (a) Verify that the new design with the HPDCache achieves better performance than the legacy Sargantana design.
   (b) Synthesize the new design with the HPDCache to verify that it can reach a target frequency of about 1.5 GHz in the slow corner of a 7nm technology node.

2 Background

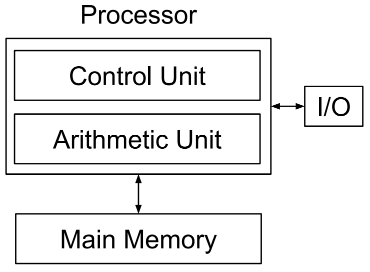
This section provides a brief background of all the computer architecture concepts relevant to this work.

2.1 Computer Architecture

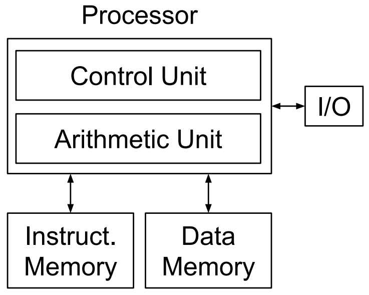
The **Von Neumann architecture** is one of the earliest and simplest proposed computer architectures. Figure 1a shows a diagram of this architecture, which defines 4 main components: the Control Unit, the Arithmetic/Logic unit, the Memory (which contains both instructions and data) and I/O. This architecture works as follows: the Control Unit directs the processor to fetch an instruction from memory, it decodes and interprets the instruction, and the Arithmetic/Logic unit executes it. Data is also fetched from the same memory when a store or load instruction is executed.

The **Harvard architecture**, shown in Figure 1b, is very similar to the Von Neumann architecture. The main difference is that the Harvard architecture defines two separate memories for instructions and data. This is done to overcome the main limiting factor of the Von Neumann architecture, which is the fact that the same memory is used for both fetching instructions and loading and storing data, so a new instruction cannot be fetched while the memory is being accessed for loading or storing data, even if the hardware for decoding and interpreting instructions is different to the one used for loading and storing data. The Harvard architecture solves this issue by using two separate memories for instructions and data, so it is able to fetch instructions while loading and storing data in parallel. However, this poses a challenge as there are cases in which it is desirable to treat code as data (e.g. when loading a program from disk, just-in-time compilation, or self-modifying code).

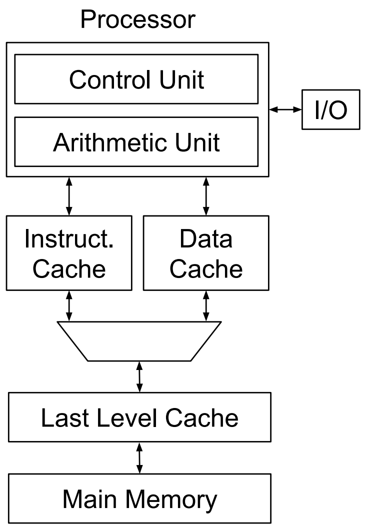
The **Modified Harvard architecture** combines aspects from the two aforementioned architectures to get the best of both worlds. Although there are different implementations on how these architectures should be combined, modern processors use the approach shown in Figure 1c: the



(a) Von Neumann



(b) Harvard



(c) Harvard Modified

Figure 1: The different general computer architectures.

core accesses a hierarchy of caches (a two level in the example figure) where the lower cache levels are split (i.e., the same level of the hierarchy consists of two separate caches, one for instructions and the other for data) and the upper level caches and the main memory contain both instructions and data.

2.2 Memory Hierarchy

As in other areas of engineering, memories present a clear trade-off between capacity, speed, and cost associated to storing data. This trade-off is shown in the diagram in Figure 2. The data storage methods with highest capacity and density, namely DRAM, are also the slowest. Meanwhile, the caches provide the fastest access although they are limited in size and have a very low density, limiting their capacity. The following subsections explain the typical memory hierarchy of a processor, from top to bottom. The hard disk drives are out of the scope of this work, so they are not explained in this section.

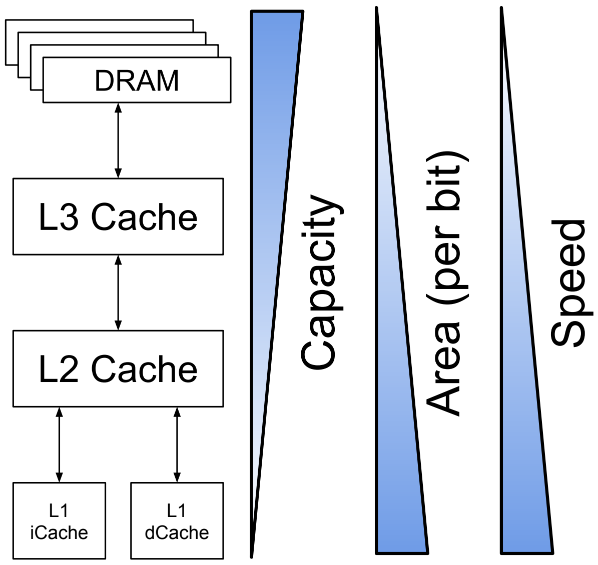


Figure 2: Schematic overview of a typical memory hierarchy within a processor and its trade-offs.

2.2.1 Main Memory

The main memory, shown as DRAM in the diagram, contains the instructions and the data of all the running applications. Conceptually, this corresponds to the main memory in the Von Neumann architecture. The main memory has a single interface to the rest of the processor[1].

Microarchitecturally, the way main memory stores information is the following: for each bit, there is a capacitor. This capacitor can either be charged to a high voltage or a low voltage. Connected to this single capacitor there is a single transistor working as a switch, which is opened when that bit is to be accessed. Many of these bits are connected to a wire called a bit line, and by connecting

[1]To be specific, in most processors there are multiple channels in this interface, but the whole memory is logically treated as one.

a single capacitor or bit to this wire it can be read or written to. This, of course, means that, due to leakage, these capacitors need to be recharged over time.

All this factors make the main memory have a large capacity, as each bit is incredibly small, but with a slow speed. This is due to multiple factors involving this microarchitecture. The first reason is that the cables and the bit cell form an RC circuit which takes time to charge. The second reason is that, because the capacitor is small compared to the bus into which it is connected, reading its contents is slow; it must slowly raise or lower the voltage of the bus and this change must be sensed and amplified. The third reason is that, unlike in an ideal world, the capacitors storing the bits have leakage currents and thus slowly discharge, requiring a periodic refresh. Thus, the main memory is the slowest memory in the system.

2.2.2 Cache

Caches are smaller memory structures embedded directly into the processor. They sit between the cores and the main memory, and serve as a buffer for the latter. Their construction, proximity to the cores and size make them much faster than main memory, speeding up memory accesses from the individual cores. As was shown in the previous memory hierarchy figure, most of the times they are also organized in levels or in a hierarchy, with larger and slower caches (L3) at the top of the hierarchy and smaller and faster caches (L2, L1s) at the bottom.

Caches work by exploiting temporal and spatial locality. Temporal locality refers to the likelihood of the same set of recently accessed data being accessed again in the future, while spatial locality refers to the tendency of accessing data in close proximity to the recently accessed data. To exploit these two sources of locality, caches are organized using cache lines, which are memory blocks of consecutive memory locations. When the CPU needs to access some data, a whole cache line is requested to memory and stored in the cache. Because the cache line is larger than a typical CPU memory access (commonly 512-bit cache lines as opposed to CPU accesses of at most 64 bits), nearby accesses (spatially local) will be served directly from the cache. And because this cache line will be kept in the cache for some time, future or temporally local accesses will also be served from the cache. In this way, caches greatly improve the latency of memory operations.

Caches are usually organized internally with two different memories: one, commonly referred to as data, for the actual data from memory and the other, commonly referred as tags, for keeping track of what memory is being stored. These tags are associated one to one with the memory blocks, and contain the most significant bits of the base address of the associated block which uniquely identify them. There are different ways of mapping the memory contents into the cache, such as direct mapping, associative mapping or set-associative mapping. These will split the address into different fields, one of them being the tag, and the others being mapping-specific bits to decide where in the cache a memory block is placed.

Memory accesses can either hit or miss the cache, depending on whether the requested data can be found within the cache. If the data is not in the cache, it must forward the request to the upper levels of the memory hierarchy. To do so, it uses a structure called miss status holding registers, or MSHR. These registers store the information to keep track of in-flight requests, as well as information to correctly tell the core which request has been completed once a response arrives. The number of MSHRs directly influences how many memory operations can be in flight, and thus the maximum memory level parallelism achievable in the system.

2.3 Virtual Memory

Another fundamental part of modern processors is virtual memory. In modern Operating Systems (OS), there can be many programs running simultaneously, each with their own code and data. It would be very impractical, complicated and insecure to allow the multiple running programs to be aware of each others' memory locations so, instead, programs are given a virtual address space. This gives programs the impression that they are alone in the system, and can use as much memory as they want, regardless of the physical location and restrictions of the system. This simplifies the way that the programs running in user mode as programmed, eliminating the need to know the underlying system organization.

In order to do this, the whole address space is first divided into pages. Each page is a fixed size contiguous block of memory (commonly 4KB, although in RISC-V there exist larger pages of 2MB or even 1GB). Then, each process running in the system has its own virtual address space, in which some pages of the virtual address space are mapped into some other page in the physical address space (i.e. into main memory). Thus, to enable virtual memory, the OS sets up a mapping of virtual memory pages to physical memory pages as it sees fit, and the memory unit of the hardware transparently translates the virtual addresses to physical addresses to access the memory hierarchy.

However, if the whole virtual address space (512GB in RISC-V Sv39) is divided into 4KB chunks, the resulting table has upwards of 134 million entries, which is unmanageable. To solve this, the page tables are actually organized into a tree as shown in Figure 3. As can be seen, the tree is organized in 3 levels. The root contains 512 Page Table Entries (PTEs), which point at other tables in the lower level, which at the same time are also divided into 512 more entries pointing to other tables in the lowermost level. This structure allows for large portions of the address space to be unmapped, allowing for a reduced memory usage for tracking the memory map when processes have a sparse memory allocation with few allocations.

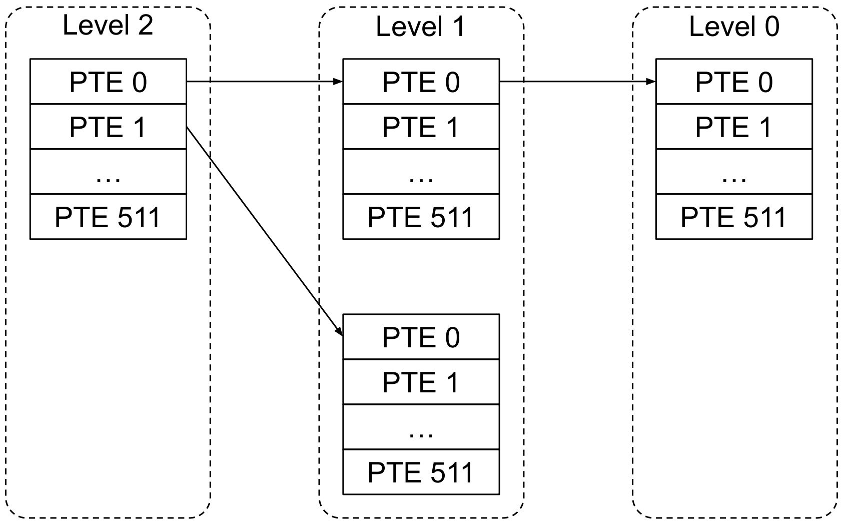


Figure 3: Representation of the page table tree using RISC-V's Sv39.

2.4 Common Architectural Elements in the Memory Pipeline

Given the previous background, this section explains the common hardware elements found in the memory pipeline to handle, optimize or speed-up memory accesses.

2.4.1 Load-Store Queue

The Load-Store Queue (LSQ) is a FIFO queue which manages the execution of memory instructions. It is located between the core execution stage and the data cache interface. When a memory instruction is issued from the core, it is inserted into the queue. When the data cache is ready to accept a new transaction, the oldest memory instruction is popped out of the LSQ and forwarded to the interface. In optimized designs such as the one in this project, bypasses can be added so that the instruction is forwarded directly to the interface if the queue is empty and the cache is ready. Moreover, in more optimized designs (which are not used in this project), the LSQ can be out-of-order to re-order memory operations to make them faster and more efficient.

2.4.2 Page Table Walker

The Page Table Walker (PTW) is the hardware module which translates virtual addresses into physical addresses. To do so, the OS provides a pointer[2] to the root of the page table tree of the active process via a configuration register. When a new address translation is requested, the PTW traverses or "walks" the page table tree, starting from the provided root, to find the physical page number corresponding to the provided virtual page number, as set up by the OS. Because the layout of these data structures is well-defined by the RISC-V ISA, the translation is able to be done entirely by hardware. The calculation of the addresses, offsets, and the following of the pointers from one PTE to the next are done by the PTW. In this way, the translation of the addresses is completely transparent to the software.

2.4.3 Translation Look-Aside Buffer

The Translation Look-aside Buffer (TLB) is a cache that stores the more recent virtual-to-physical address translations performed by the PTW. When a virtual address needs to be translated into a physical address, the processor must request a translation to be done by the PTW. Intuitively, this requires at least 3 additional memory accesses and thus has a considerable latency. So, using the PTW for every memory access is expensive. Hence, the TLB keeps the most recent translations to serve them much faster than the PTW.

[2]The physical address.

3 Previous Work

This section details the microarchitectural implementation of the previously mentioned computer architecture concepts in the core and System-on-Chip (SoC) that is used as the base of this project.

3.1 Sargantana Core

Sargantana [2] is a 64-bit processor based on RISC-V that implements the RV64G ISA. It features a highly optimized 7-stage pipeline implementing out-of-order write-back, register renaming, and a non-blocking memory pipeline. It also features a 16KB L1 instruction cache and a 32KB L1 data cache. The design has been taped-out using 22nm FD-SOI GlobalFoundries technology and reaching 0.89 GHz in the slow corner with an area of 0.466mm^2 [2].

Figure 4 shows the pipeline of Sargantana. It starts with two fetch stages, interfacing with a one cycle instruction cache. These two stages also contain a simple branch predictor. Next there is the decode stage, which inserts decoded instructions into the instruction queue. This queue splits the front-end (fetching and decoding of instructions) and the back-end (register renaming, reading and instruction execution). This allows to continue fetching instructions while the back-end is stalled, or continue execution of decoded instructions while the front-end is stalled. After the instruction queue there is the register renaming, in which virtual registers are mapped to physical registers and the write-after-write dependencies between instructions are eliminated. Next, the registers are read (or the data is gathered from the bypasses). Then, there are the different execution units (of varying latency) before the writeback stage.

The memory pipeline is in-order and non-blocking. It is comprised of a load-store queue, a store buffer and a pending memory request queue. It has a latency of 2 cycles and blocks the committing of instructions younger than the store in flight.

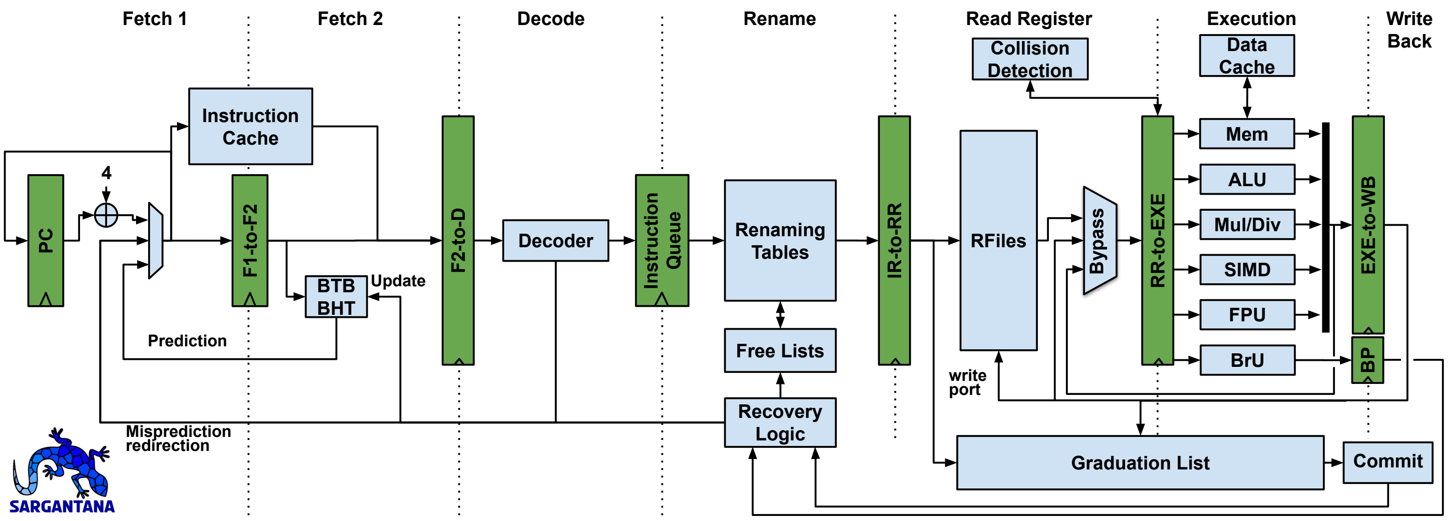


Figure 4: Sargantana pipeline.
Authorship by the authors of [2].

3.2 Lagarto Lowrisc System on Chip

The Lagarto Lowrisc SoC comprises most of the memory hierarchy (L2 and L1 data cache) as well as the peripherals used to support the Sargantana core. It is implemented in SystemVerilog and Chisel, making it slightly more difficult to maintain, modify or improve as opposed to a simple SystemVerilog implementation. As commented earlier, the L1 data cache has a 2 cycle hit latency, and responds with any memory exceptions that could have occurred after this same latency. The L1 is private, while the L2 is shared. The caches implement a basic coherence mechanism, also used by accelerators (if any), which theoretically allows for a multicore system.

The most important point about this SoC is that it is not oriented nor ready for HPC. The data cache has a relatively high latency, and it only has 2 MSHRs, so it only supports 2 memory requests in flight. The buses only support 128 bits, making transactions between the memory hierarchy slow. The connection to the core is of only 64 bits, preventing it from using wide vector units in the future. The coherence protocol is also very basic and has some known bugs in multicore setups. Finally, due to the simple interconnect or network on chip (NoC), the design is not scalable and so a many-core chip is infeasible.

3.3 High Performance Data Cache

The High Performance Data Cache (HPDCache) is an open-source, configurable L1 data cache design [3]. The diagram for its internal structure can be seen in Figure 5. The cache is out-of-order, meaning it can re-order memory operations to get a higher throughput and better performance. The HPDCache supports a bus of up to 512 bits and it implements a write-through policy with a write buffer, allowing multiple write operations to be in flight and reducing their latency. The HPDCache also supports up to 128 MSHRs, allowing up to 128 memory requests in flight. It supports multiple requesters, meaning that the PTW can be directly connected to the L1, speeding it up, and also allowing for hardware prefetchers to be installed.

In terms of physical design, the HPDCache implements a number of SRAM macros for reduced area footprints. These SRAMs correspond to the memories for the cache directory, data and MSHRs. The SRAMs are optimized for high bandwidth, providing up to 32 bytes per cycle. They are also optimized for low energy consumption by utilizing banking and enable wires (this can be seen in Figure 5), allowing some of the memories to remain disabled[3] and thus saving energy. Lastly, the SRAMs are optimized for area and latency, choosing the width and depth values resulting in a ratio close to a square. This shape minimizes the area and the latency by balancing the overhead area added by the decoding and driving circuitry, and also balancing the capacitance of the row and column wires.

[3]This will depend on the number of bytes requested.

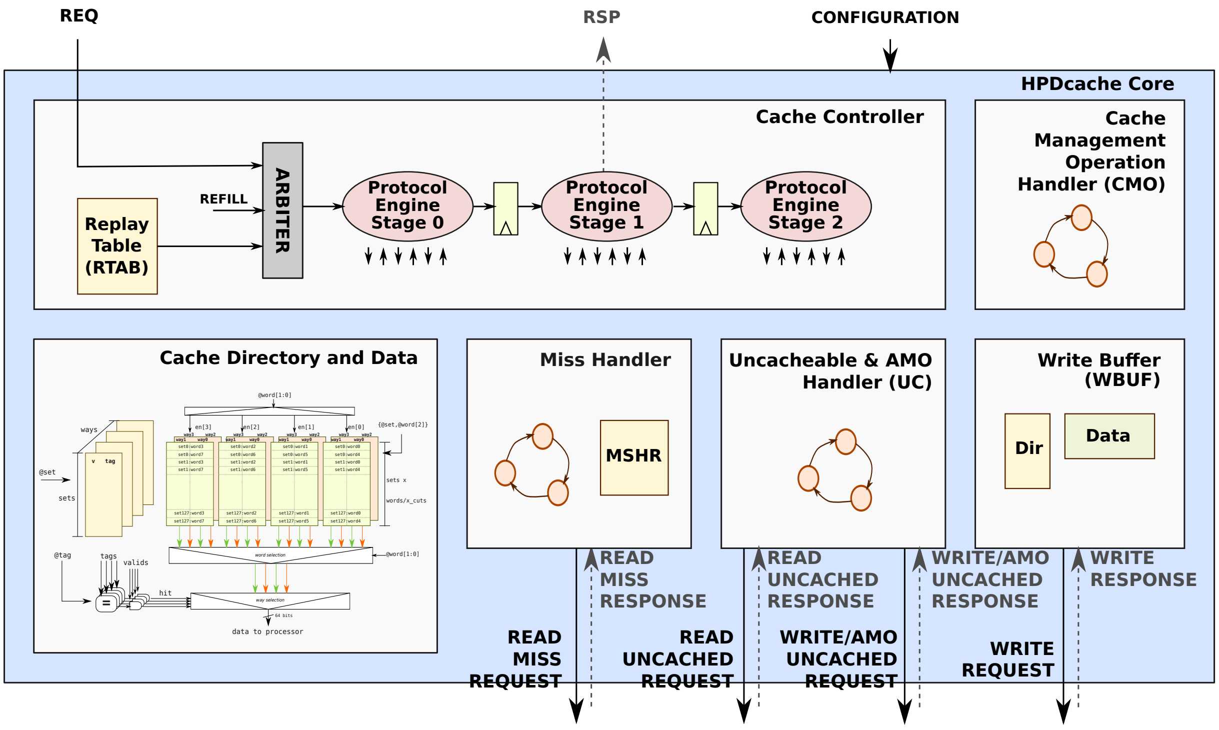


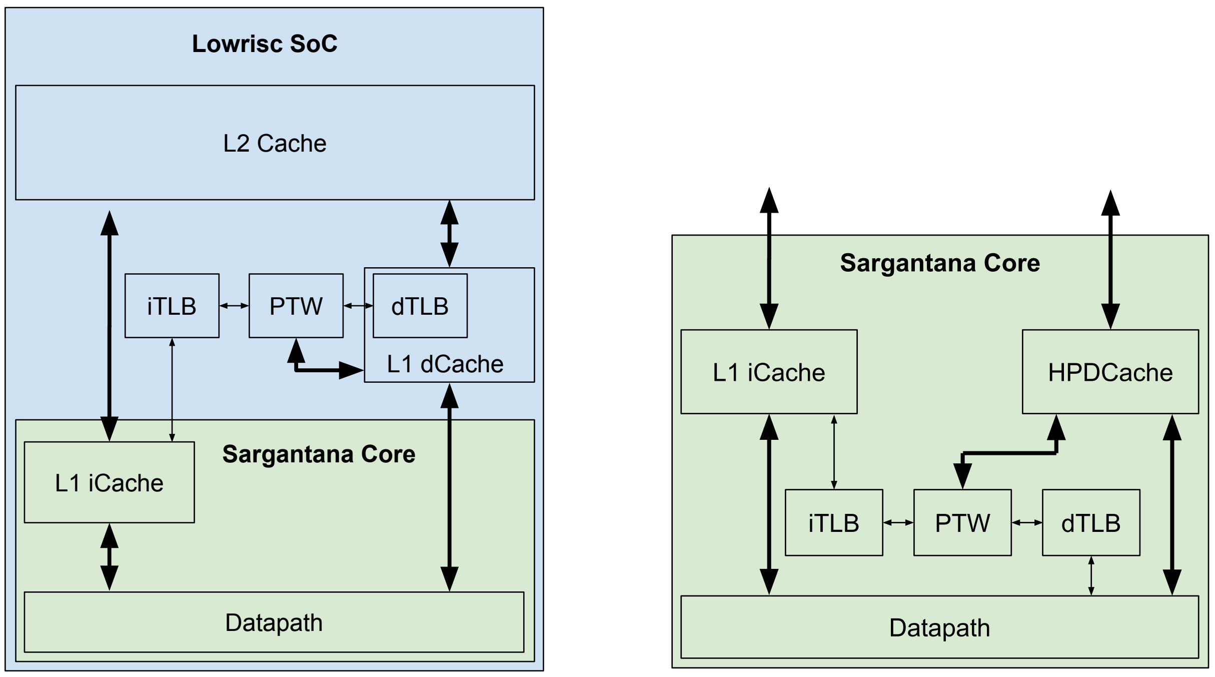
Figure 5: Architecture of the HPDCache.
Authorship by César Fuguet.

4 Integration of HPDCache with Sargantana

This chapter describes the development process followed for this work. It first explains how the HPDCache has been integrated with Sargantana, and next the optimizations done to the memory pipeline are presented.

4.1 Cache Integration

This section details how the integration of the HPDCache has been done. An overview of the integration can be seen in Figure 6. As can be seen, the integration also implies moving a number of blocks from the SoC (or, more specifically, the "tile" in Lowrisc terms) into the actual core. The old interface with the SoC had a connection between the L2 and the L1 iCache, a TLB connection for the iCache and a connection between the datapath and the L1 dCache, which uses virtual addressing. Within the old SoC there was also a connection between the PTW and the L1 dCache. For the integration to be successful, all of this has to be moved into the core, exposing only the connections to an outside memory hierarchy.



(a) Original architecture of the Sargantana core using the Lowrisc SoC.

(b) Final architecture of the Sargantana core with the integrated HPDCache.

Figure 6: Initial and final architectures of the core.

The first step, as is apparent from the figure, is the incorporation of the TLB into the core, as it was previously integrated within the data cache of the Lowrisc SoC. The second step is to change the interface of the core to the data cache, as the new cache has some differences with regards to the previous interface. The third step, although from the figure it might not be as apparent, is to adapt the control logic and state machines of the memory pipeline to work with the new cache communication protocol.

4.1.1 Incorporation of the Translation Look-Aside Buffer

The memory pipeline has been reconfigured as shown in Figure 7. In the first cycle of execution of a memory instruction, the address of the memory operation is calculated[4] and the instruction is added into the LSQ (shown in this figure as a register). In the next cycle, the virtual address is translated by the TLB into a physical address. In the third cycle, the instruction leaves the LSQ and is sent to the data cache. As is apparent, this adds a lot of latency to the execution of memory instructions and is a clear optimization opportunity, as explained in Section 5.

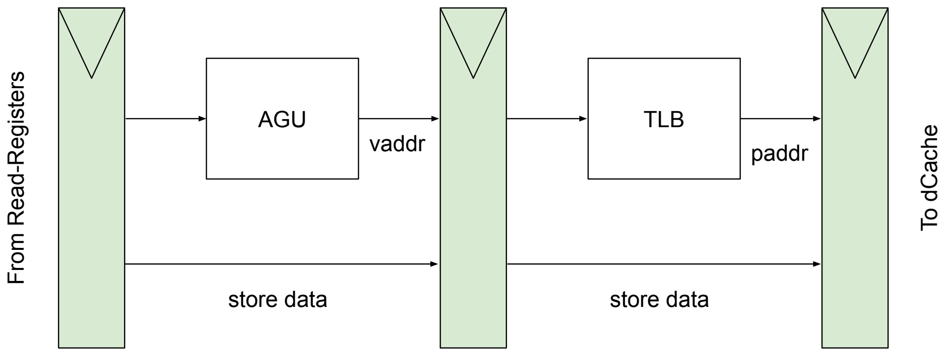


Figure 7: View of the new memory pipeline.

Going into detail, Figure 8 shows the implementation of the Load-Store Queue with the address translation logic in place. In the previous version, the LSQ was a typical FIFO queue with two pointers, one for the `tail` entry where new instructions are added, and one for the `head` entry where instructions are popped. As can be seen in the figure, for the new version of the queue a new pointer has been added, `tlb_tail`. This is used to keep track of which instruction to translate next. This pointer moves away from the `head` and towards the `tail`, i.e. it translates instructions from oldest to youngest. When the `tlb_tail` points to a valid instruction to be translated, a request is sent to the TLB. If the response is valid (either the translation is done successfuly or there is an exception), the pointer is advanced to the next instruction. An instruction will only leave the LSQ when it has been translated or it has an exception (misaligned address and page or access faults).

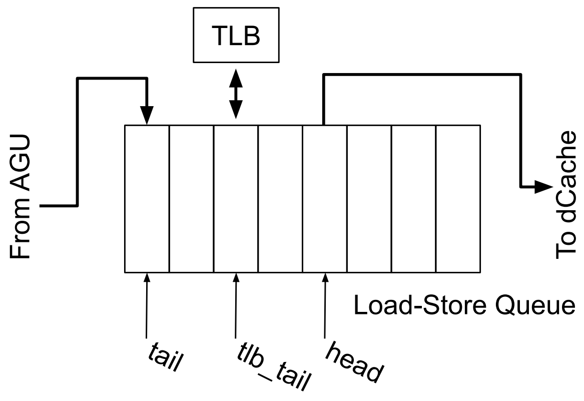
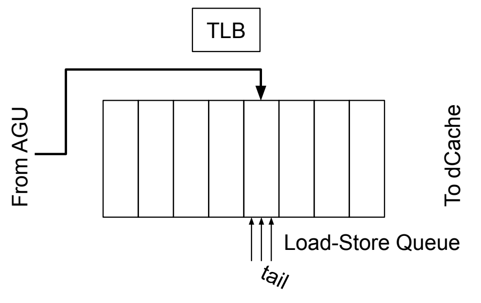


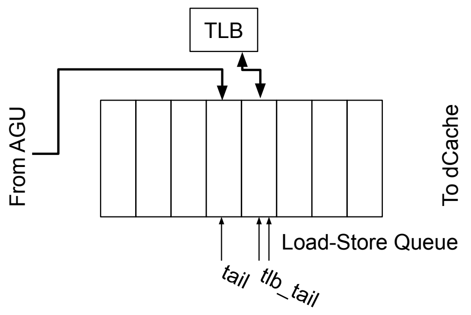
Figure 8: Detailed view of the LSQ implementation with the TLB.

[4]As defined by the RISC-V ISA, by adding the contents of `rs1` with the immediate value encoded within the instruction.

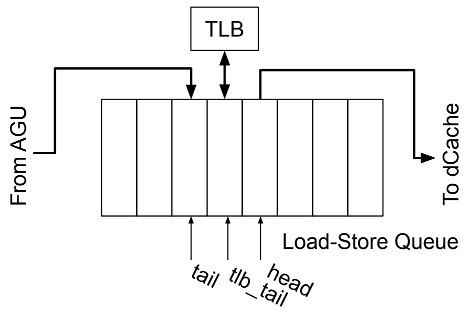
To better understand how the LSQ works, Figure 9 provides an example in which 3 instructions are added to the LSQ in 3 consecutive cycles. The LSQ is initially empty, and all pointers point to the same entry in the middle of the queue. Figure 9a shows the first cycle, in which an instruction is written into the entry pointed by `tail`, which will advance at the next rising edge of the clock. In the next cycle, Figure 9b, another instruction is inserted again at the entry pointed by `tail`. In the same cycle, because `tlb_tail` points to a valid instruction, the address is sent to the TLB to be translated. In this example, the address is correctly translated which means in the next rising edge the `tlb_tail` will also advance. Finally in the third cycle, Figrue 9c, a third instruction is added again at the `tail`. Because `tlb_tail` points to a valid instruction, the instruction added in the second cycle is translated. Finally, because the entry pointed by `head` has been translated, it is sent to the dCache.



(a) Cycle 1. New instruction is written into `tail`. `tail` pointer is advanced.



(b) Cycle 2. New instruction is written into `tail`, instruction at `tlb_tail` is translated. `tail` and `tlb_tail` pointers are advanced.



(c) Cycle 3. New instruction is written into `tail`, instruction at `tlb_tail` is translated, instruction at `head` is sent to the cache. All pointers are advanced.

Figure 9: Example of how the LSQ works. 3 instructions are inserted in consecutive cycles.

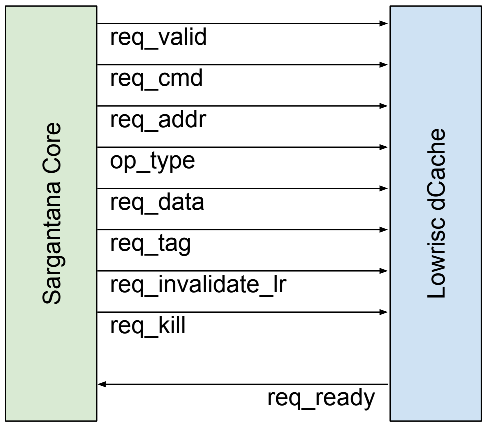
In short, this architecture provides 3 key benefits. First, because the translation is integrated into the LSQ, it allows for the front-end and the instruction queue to continue issuing instructions while there is a TLB miss instead of blocking the whole pipeline. Second, the early detection of exceptions allows for a simplified memory unit (Section 4.1.3). Third, having non-aliased physical addresses allows for value forwarding from the Store Buffer to be implemented in the future (i.e. forwarding a value recently stored to a load without going to the dCache).

4.1.2 Modification of the Cache Interface

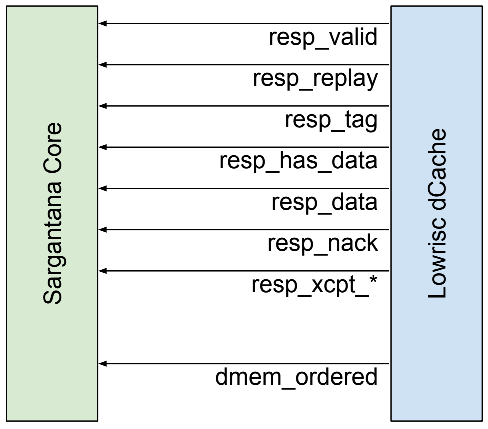
The Lowrisc cache used previously and the new HPDCache have similar interfaces, but they do not match exactly. The differences can be seen between Figure 10 and Figure 11.

Starting with the request from the core to the cache, we can see that there are some similarities. Some of the signal names change (e.g. `op_type` for Lowrisc is `req_size` for the HPDCache, the `tag` is `tid`, `dmem_ordered` is `wbuf_empty`[5]), but some others are missing or new. The Lowrisc cache (Figure 10a) has the signals `req_invalidate_lr` and `req_kill` which are missing in the HPDCache. They have to do with the fact that exceptions are detected late, and an ongoing request might have to be killed, which is not the case with the HPDCache; all requests will be valid. The new signals in the HPDCache (11a) are `req_uncacheable`, which indicates that the cache should not cache the data, and `req_be` which is explained next.

As can be seen in Figure 12, the HPDCache has some requirements with respect to the alignment of the writen data. In particular, it must be naturally aligned to the address within the 128 bits of the data bus. Furthermore, the `req_be` signal must indicate which of the bytes within that bus are valid/enabled. These bits must, again, be naturally aligned to the address *and* the size of the operation. Thus, using the request address and size, some multiplexers have been added to the cache interface to align the LSBs from the data coming from the core to the aligned position within the data bus, and some additional logic has been added to generate the byte enable signal.



(a) Request.



(b) Response.

Figure 10: Interface between Sargantana core and Lowrisc cache.

[5]These two signals are not identical, but they are effectively used for the same purpose: respecting the Memory Model after the recovery of a miss-prediction.

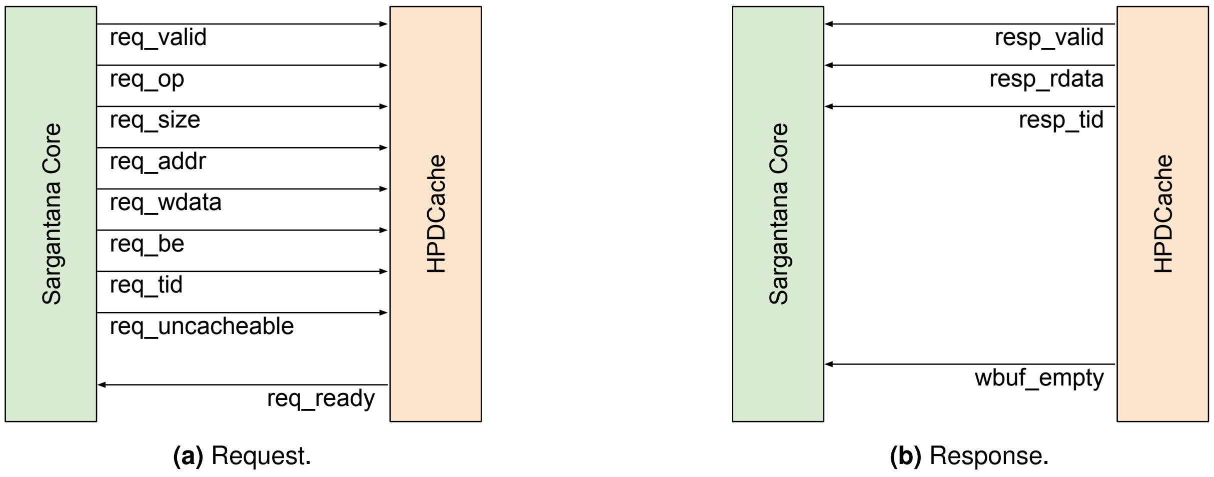


Figure 11: Interface between Sargantana core and HPDCache.

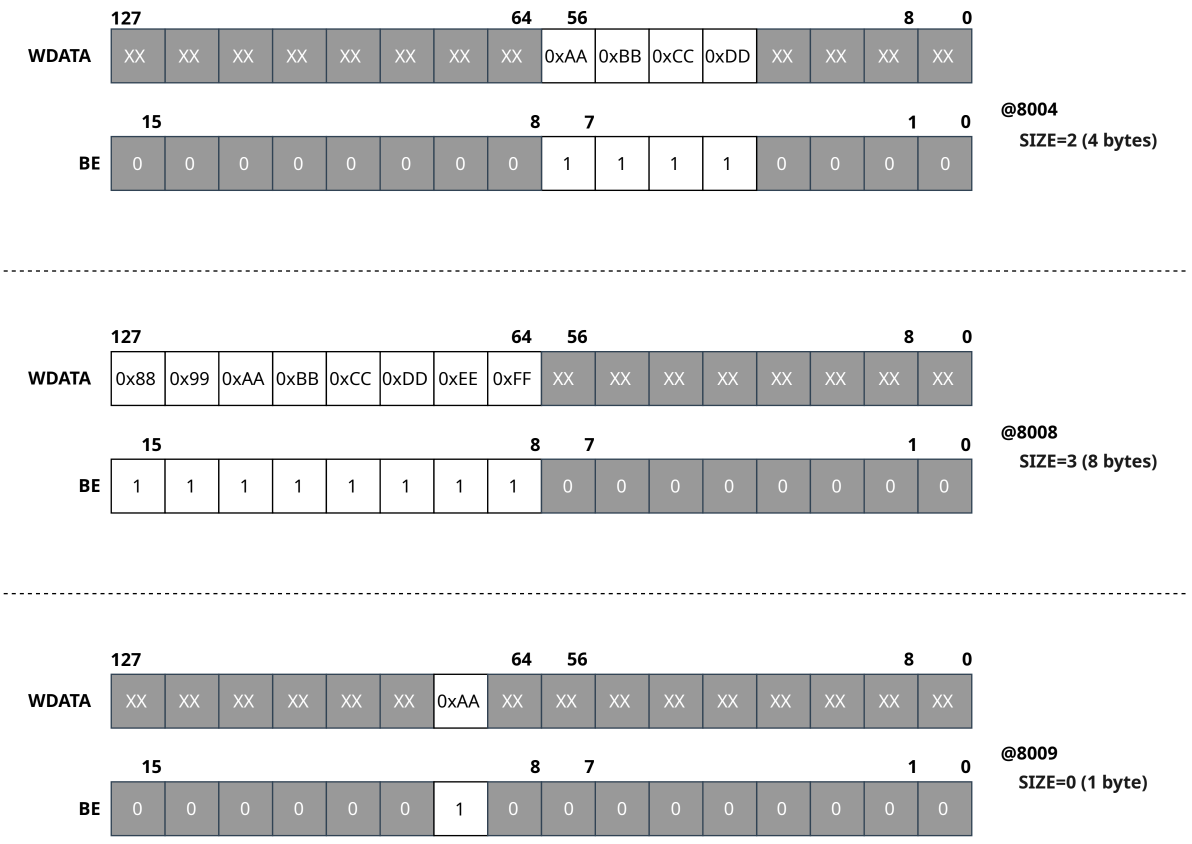


Figure 12: Alignment requisites of the HPDCache. Authorship by César Fuguet.

Continuing with the response, the signals of the HPDCache (Figure 11b) are much more simple, having only the valid bit, the response data and the tag of the request (Figure `resp_tid`). The additional signals in the Lowrisc interface (Figure 10b) have to do with the exception detection (`resp_xcpt_*`), with the fact that sometimes the cache will not acknowledge a handshaked request, or with control data which can be superfluous (e.g. whether the response is to a miss/replay or not, or to a load or a store). The omission of these signals will make the memory unit much simpler, as will be described next.

Finally, the data signals of the response are also naturally aligned, as with the request data, so the interface from the cache to the core has multiplexers for selecting the data depending on the address and the size. Additionally, because this is in the critical path (as the SRAM output has a long delay), the output is registered right after these multiplexers.

4.1.3 Modification of the Memory Unit

Due to the change in the interface between the core and the data cache, the control logic in the memory unit must also change. As mentioned in the previous section, these changes consist mainly on the simplification of the finite state machine governing the memory pipeline. Figure 13 shows the FSM that was used for the Lowrisc cache. As can be seen, it is relatively complex. It starts with the "ResetState", which is entered right after a reset or when there is a recovery of the speculative state (e.g. a branch miss-prediction). This state basically waits for the memory hierarchy to reach an ordered state, which is needed in order to obey the Memory Model and ensure the memory accesses are done in an order expected by the programmer. This is specially relevant for multithreaded applications in a multicore environment. The "ReadHead" performs memory operations as they come. When the cache isn't ready for a new transaction or hasn't acknowledged the last request, the FSM transitions into "WaitReady", in which as the name implies it waits until the cache returns to a normal state.

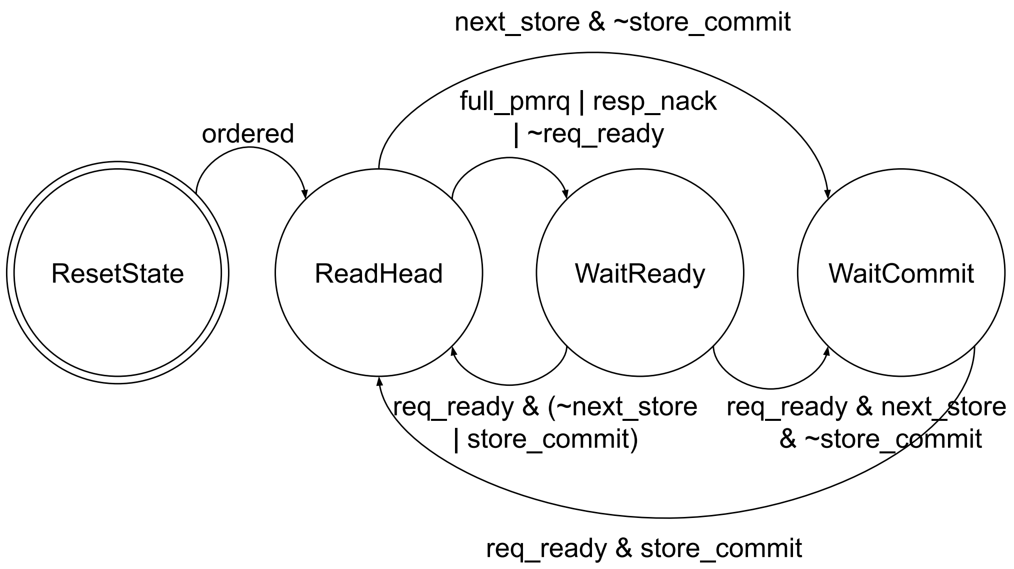

next_store & ~store_commit
full_pmrq | resp_nack | ~req_ready
ordered
ResetState
ReadHead
WaitReady
WaitCommit
req_ready & (~next_store | store_commit)
req_ready & next_store & ~store_commit
req_ready & store_commit

Figure 13: FSM of the memory unit using the Lowrisc cache.

There is also the "WaitCommit" state, which is entered when a store must be sent to the cache but it isn't the oldest instruction in the pipeline (i.e. it isn't the next instruction to be commited). This is because if the store is sent to memory and an older instruction has an exception, it would be practically impossible to recover the previous state. Thus, the instructions would appear to execute out-of-order and break the programming model.

For the new cache interface, only the "ResetState" and "ReadHead" states are kept. This is because the HPDCache protocol specifies the requester mustn't wait for the cache to be ready. That is, it must make the request as available until the cache is ready instead of not sending the request until the cache is ready. The other reason is because the HPDCache cannot respond a "not acknowledged", i.e. all requests sent to the cache can't fail. The handling of the cases in which the Pending Memory Request Queue is full and that the store isn't the oldest instruction are handled within the "ReadHead" state, there isn't a need for a special state for them. The behaviour of the "ResetState" is kept the same, as the issues with preserving the Memory Model still hold true.

5 Memory Pipeline Optimizations

This section details all the optimizations done to the initial architecture explained in the previous sections. As mentioned before, the initial implementation is relatively simple and has room for optimization. This is because the first goal was to integrate the HPDCache with Sargantana to achieve a relatively simple initial design which can be extensively tested, and then focus on improving the performance as a second goal.

Also note that this work does not compare the addition of a register at the output of the memory interface, mentioned in the previous section. This is because this was a clear critical path (below 1.1GHz) and the registers were added by design in anticipation to this.

5.1 Translation of an Incoming Instruction

The first optimization consists on translating the virtual address of an incoming instruction before inserting it in the Load-Store Queue. The rationale is that there is a low amount of work done after the instruction enters the memory pipeline (only the address computation in the AGU) and that the TLB lookup is a relatively fast operation due to the reduced size of the TLB.

Figure 14 shows the implementation for this optimization. The specific changes done are shown in red. The optimization basically consists on two multiplexers. The first one, at the input of the TLB, chooses which instruction to translate: either the new one entering the memory pipeline or a pending one in the queue. The other multiplexer chooses which instruction to insert into the queue: the one entering the memory pipeline or the same one with the translated address.

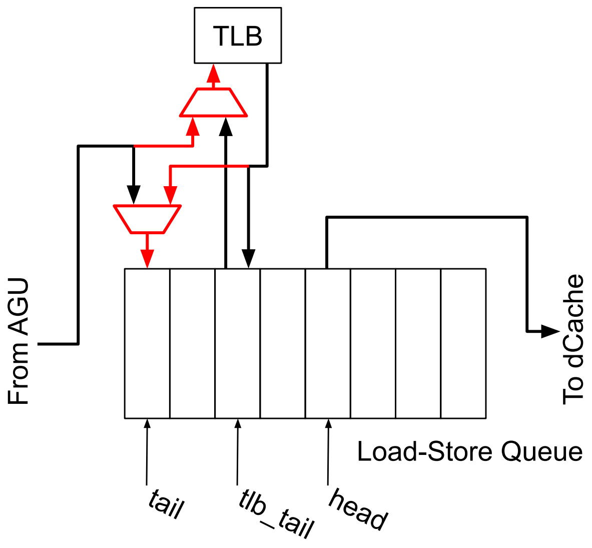


Figure 14: Multiplexers added to translate an incoming instruction to the LSQ in the same cycle.

These two multiplexers work in tandem, that is, they form two distinct circuits. This is because the TLB is shared between two potential "producers", either the incoming instruction or a pending instruction in the queue, and thus there is a structural hazard. Thus, either the TLB input multiplexer selects and incoming instruction and the queue multiplexer selects the translated instruction, or the TLB multiplexer selects to translate a pending instruction and the queue multiplexer selects to write the incoming instruction as-is.

A number of conditions must be met to enable this bypass. The first condition is that the queue must have no instructions pending to be translated (i.e. the `tail` and `tlb_tail` must point to the same entry), otherwise they are translated first. The other condition is that the response from the TLB must be valid. In this case, even though the incoming instruction is forwarded to the TLB, the response is not used, i.e. the bypass for writing into the queue selects the incoming instruction so that it can be translated later in the queue. When this bypass is enabled, both tail pointers are increased.

5.2 Bypass of the Load-Store Queue

The second optimization consists on completely bypassing the LSQ, allowing for an instruction to be sent immediately to the cache. This greatly reduces the latency of memory operations, at the cost of an increased critical path from the pipeline registers to the dCache. Figure 15 shows an schematic of how this bypass is implemented. This bypass requires the previous optimization and thus the same conditions apply. This new multiplexer selects the translated instruction from the TLB when it is valid, the queue is empty and the instruction is a load[6]. Regardless of this bypass, memory operations are always written into the queue, because they have to be kept somewhere in case the cache is not ready to accept a new request. If it is accepted, all pointers are advanced.

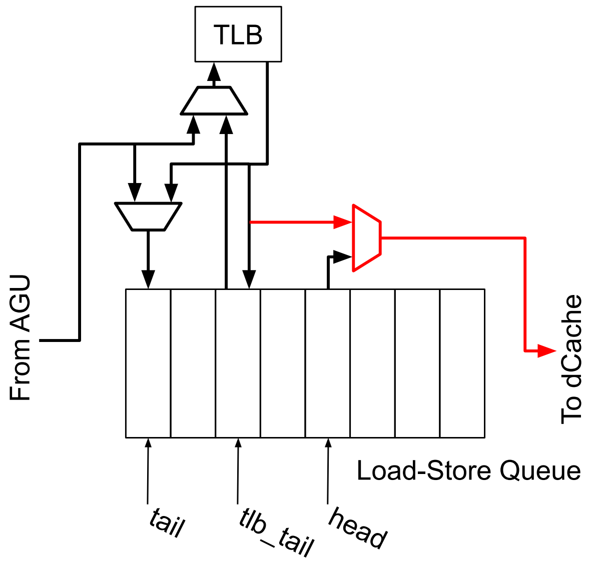


Figure 15: Multiplexer added to completely bypass the LSQ.

5.3 Register in Multiplier Unit Output

After adding the previous optimization, and as shown in the evaluation (section 7), a critical path appears between the output of the multiplier unit and the memory pipeline. To solve this issue, this optimization consists on adding a register in that path, as shown in Figure 16. Before the integration with the HPDCache, the paths in the first stage of the execution pipelines were short enough that the little logic there is in the second stage of the multiplication unit (MUL2 in the figure) did not cause any

[6]Again, this is done because the store has to wait to be the oldest instruction in the pipeline.

critical paths. However, that is not the case after the integration of the HPDCache, so an additional register is needed.

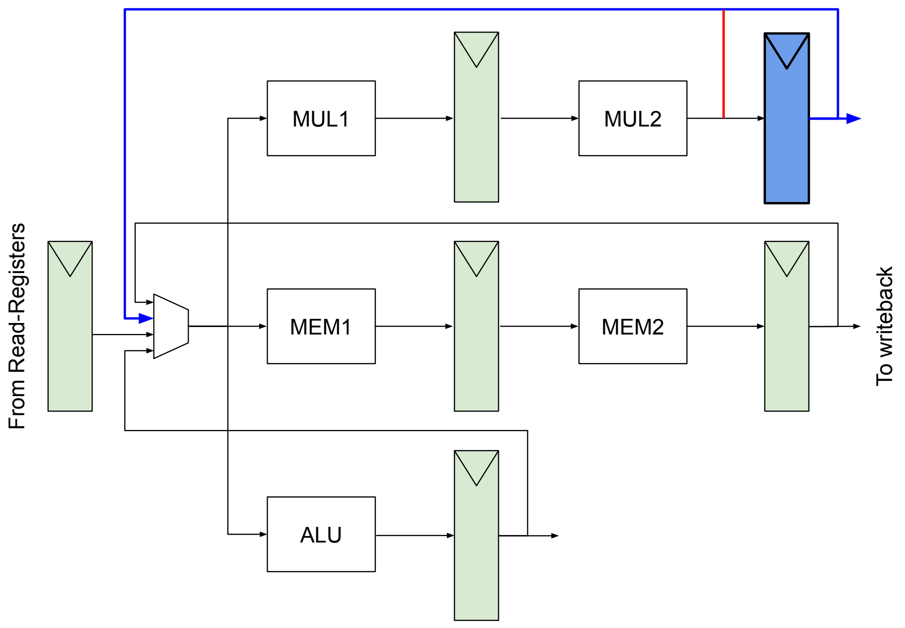


Figure 16: Addition of a register in multiplication pipeline. Additions in blue and removals in red.

This change increases the latency of integer multiplications in the best case[7] from 1 to 2 cycles, but allows for a higher frequency. The rationale behind this decision is that, in typical HPC workloads, there are fewer integer multiplications than memory operations[1]. Thus, it is much more critical to improve the loads and stores (by the means of the LSQ bypass or increasing the frequency) than the multiplication pipeline, as demonstrated later in the evaluation.

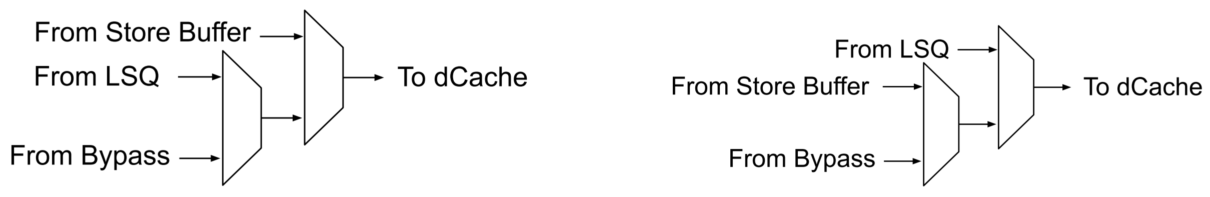
5.4 Prioritization of Loads Over Stores

In this optimization, the priority encoder before the output of the LSQ to the dCache is reorganized in order to give a higher priority to loads than to stores. The reason for this change is that the HPDCache uses single-ported SRAMs to reduce area and have increased speed. However, this comes at the cost of having structural hazards. One of these hazards is that it cannot accept a load after a store. Within the internal pipeline of the cache, stores write to memory one cycle after the request. Thus, when the load comes in the following cycle, the memory is occupied performing the write and the request cannot be accepted. This causes a stall in the memory pipeline of 1 cycle in the cases in which there is a load after a store.

To avoid this situation, the idea behind this optimization is to perform as many loads as possible before a store, and delay stores as much as possible. This prevents the structural hazard explained previously, since it avoids the store-load case. The optimization is implemented as shown in Figure 17. All the other cases (load-load, store-store, load-store) are compatible within the cache pipeline without causing any stalls. The cost of this, however, is a higher retention of stores and

[7]When the result is needed by the next instruction, i.e. a 1-cycle dependency.

increased pressure in the store buffer. This means that, potentially, the store buffer is filled up faster and it must be emptied more often, sometimes stalling incoming memory operations.



(a) Before. **(b)** After.

Figure 17: Priority encoder for the output of the LSQ.

5.5 Early and Back-to-Back Store Execution

This optimization seeks to reduce the latency and augment the throughput of stores as much as possible.

One of the clear optimization opportunities for stores is reducing the time it has to wait for the instruction to become the oldest in the pipeline. Previously to this work, the memory pipeline could not send a store until the instruction was not in the head of the graduation list, or in the first out of the two commit ports. However, by taking advantage that there are two commit ports, if the second port has a store and the instruction in the first port does not have any exception, this store can be safely launched[8], and committed in the following cycle. This effectively reduces the latency of stores by one cycle in the cases where the previous instruction takes more than one cycle to execute (and thus the store has had time to enter the back-end).

Another improvement done consists on enabling launching stores one after the other. This is specially useful in functions where variables are pushed to the stack at the start. This functionality was not implemented in the old version of the core due the interface of the Lowrisc cache. The protocol required the address of the store to be sent in one cycle, the data in the next one, and to not launch any other request until the third cycle in case there was an exception or the cache failed to acknowledge the request. However, with the HPDCache this is not the case anymore; once a request is made it cannot fail. Thus, all the logic preventing new stores to be launched can be safely removed, allowing the execution of stores back-to-back to improve performance.

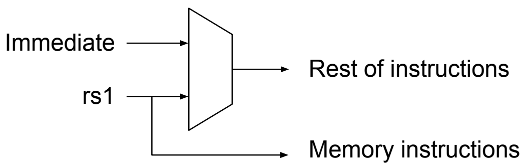
5.6 Final Critical Path Optimization

Finally, after all these optimizations, it is time to analyze the critical paths and to improve the maximum frequency. As commented previously, the critical path is the one flowing from the pipeline registers into the address generation unit, the TLB, and finally the cache. In between these there are, of course, a number of bypasses which are targeted for optimization. The rest of logic does not leave a lot of room for optimization: the address generation unit is a simple adder, which (a) is impossible to skip and (b) does not add enough delay to justify its modification. The TLB is a simple CAM with a few entries, so there is not much room for improvement. Finally, modifying the cache is

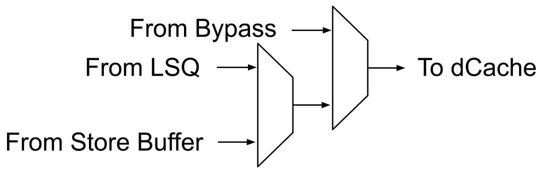
[8]The actual implementation is slightly more involved, as the graduation list index of the store at commit must match the one in the store buffer, and some other checks must also performed. However, the general idea is the same.

far too complex and outside the scope of this project. Thus, optimizing the different multiplexers is the more viable choice.

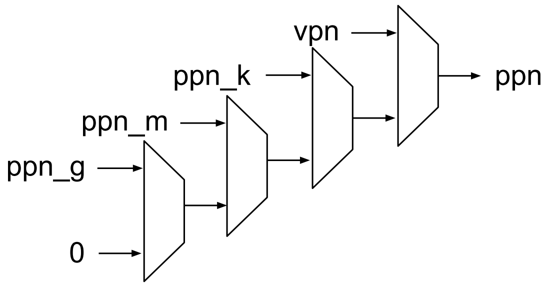
Figure 18 shows all the optimizations performed to optimize the critical paths, going from their source to their sink. Some of these optimizations are simple, but since in the past they were not in the critical path they were never looked at or optimized.



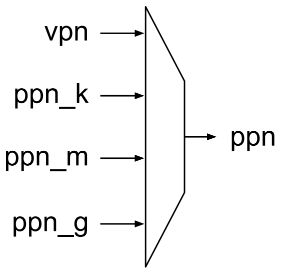
(a) Optimization of the address source.



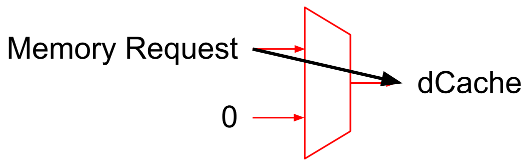
(b) Optimization of LSQ bypass.



(c) TLB output before optimization.



(d) TLB output after optimization.



(e) Optimization of dCache interface output.

Figure 18: Optimizations done to the critical path.

Beginning with Figure 18a, at the start of the execution stage there is a multiplexer for choosing the operand of the instruction, either the immediate value encoded within the instruction or the value of rs1. Since the ISA defines all memory instructions will use rs1, this multiplexer can be safely skipped.

As mentioned before, at the output of the LSQ there is a priority encoder for routing a memory operation from the LSQ itself, the Store Buffer or the LSQ bypass. Because the latter is in the critical path, it is beneficial to give it a higher priority, as shown in Figure 18b, so that it goes through 1 less multiplexer. Since the bypass is only activated when the queue is empty, the logic is not modified.

At the output of the TLB there is a priority encoder as is shown in Figure 18c. This is because the RISC-V ISA supports a number of different page sizes, and so the TLB must support them and choose a physical page number based on that. As can be seen, in the worst case there is a lot of multiplexers in the way, adding a non-negligible amount of logic. To improve this, the priority encoder can be reconfigured as a single 4:1 multiplexer as shown in Figure 18d. Because only one of these options is valid at the same time, the logic is once again preserved.

Finally, at the dCache interface, there was a multiplexer for zeroing-out a request in case it was invalid. This was remnant of the previous interface with the Lowrisc cache. However, this is no longer needed because requests now use the valid-ready handshake and only look at the valid bit coming out of the core. Thus, as shown in Figure 18e, this multiplexer can be safely removed.

6 Evaluation Methodology

This section details the evaluation methodology used to validate the integration and the optimizations, as well as the evaluation method to do the cost-benefit analysis.

6.1 Architectural Parameters

The main architectural parameters of Sargantana used in the evaluation are specified in Table 1. It is important to note that evaluation compares Sargantana using two different L1 data caches, Lowrisc and HPDCache, so the parameters of both caches are set to the same values (L1 Dcache size and L1 Dcache associtiativity in the table) for a fair comparison.

Parameter	**Value**
Integer physical register count	64
Floating-point physical register count	64
Instruction Queue length	16
Load-Store Queue entries	8
Store Buffer entries	8
Pending Memory Request Queue length	16
L1 iCache size	16KB
L1 iCache associativity	4-way
L1 dCache size	32KB
L1 dCache associativity	4-way
iTLB/dTLB entries	8

Table 1: Architectural parameters used for the evaluation.

6.2 Simulation Environment

The simulations have been done using Verilator [14], an open-source simulator. Verilator compiles the RTL code into an executable binary, providing very fast simulations. It is a 2-state simulator, meaning that it treats all logic as "1" and "0", and so it is very fast but comes at the cost of losing X and Z propagation. Nevertheless, this is not an issue as the core does not have any interfaces using tri-state ports.

The simulator includes support for loading ELF binaries, so it can execute any program compiled for RISC-V with GCC. Additionally, Verilator also supports dumping a log of all instructions as they are executed, logs of memory reads and writes, and also a dump of the pipeline status so that it can be inspected using a graphical interface providing a typical pipeline diagram. All of this proved helpful in the debugging of the integration and the optimizations.

With regards to the memory interface between the L1 iCache and dCaches, the simulator provides an ideal memory with multiple independent channels and configurable latency for each of them. The evaluation uses a typical latency of 20 cycles [17] for all the channels (iCache fetch, dCache reads, writes, uncacheable and atomic operations). This is representative of the delay of an L2 cache and similar to the average latency the Lowrisc caches, so it is a fair comparison point.

6.3 Testbenches

Different testbenches are used for verifying the integration and the optimizations. One of the testbenches is the RISC-V tests. These tests target specific instructions for specific environments, e.g. there is a test for loads in bare metal and another test for loads using virtual memory. Each of them test the outcome of each instruction in a number of different edge cases. These are useful for identifying the source of the issue, as they are quite fine-grained.

However, the RISC-V tests alone are not enough since there are a few of them (as they are handmade), and they do not test many combinations of instructions and how they interact between them once they enter the core pipeline. For this reason, torture tests have also been used. These consist on generating a binary, using riscv-torture [15], which contains a sequence of valid instructions. Parameters such as the number of instructions or the instruction mix are configurable, so different configurations are used for stressing different parts of the system. All of these tests are executed under virtual memory to stress-test the TLB and the PTW integration.

Additionally, the benchmarks that are explained in the following section are also useful to make sure everything works, as some of them also have a verification routine.

6.4 Benchmarking Suites

Two benchmark suits are used for the performance evaluation. One is the RISC-V ISA tests, containing an assorted selection of common computing kernels, as shown in Table 2. The other is the EEMBC, containing benchmarks for stress-testing embedded microprocessors, and is shown in Table 3. A more detailed description of the EEMBC benchmarks can be found in [16].

Name	**Description**
bsort	Bubble sort
Mat Mul	Integer matrix multiplication
Multiply	Software integer multiplication
qsort	Quicksort
fibonacci	Recursive fibonacci
rsort	Radix sort
histogram	Computation of a histogram
vvadd	Adds two integer vectors
median	Median filter
spmv_int	FP64 sparse matrix-vector multiplication
dhrystone	Synthetic benchmark

Table 2: Description of benchmarks found in the RISC-V ISA benchmark suite.

Name	Description
aifir	FIR filter
bitmn	Bit manipulation
cacheb	Long sections of control code, low temporal locality
canrdr	CAN node RDR message processing
pntrch	Double linked list search
puwmod	H-bridge pulse width modulation
rspeed	Road speed calculation
ttspark	Timing calculation for engine ignition
a2time	Angle to time conversion
aifftr	Fast fourier transform
aiifft	Inverse fast fourier transform
basefp	Basic integer and floating point operations
idctrn	Inverse discrete cosine transform
iirflt	IIR Filter
matrix	LU decomposition, determinant computation
tblook	2D/3D table lookup and interpolation

Table 3: Description of benchmarks found in EEMBC automotive benchmark suite.

6.5 FPGA Emulation

To further test the integration, an FPGA is used to implement the prototypes and to verify that they are able to boot Linux. To do so, some of the existing infrastructure which provides support for the memory and UART peripherals, and implements the RTL in an Alveo U280 running at 50MHz, has been adapted. The existing infrastructure connects the peripherals to the old Lowrisc SoC via AXI. To support the new architecture, open-source AXI blocks such as crossbars, width and protocol converters are used to connect the L1 caches to the peripherals exposed by the existing infrastructure. Additionally, a simple AXI-mapped timer is connected via an interrupt request port to the core. This timer simply divides the clock into a known frequency, compares the timer counter to a register, and sends an interrupt request when it matches. These are the minimum requirements to run Linux, and allow the verification objective of achieving a Linux-capable integration.

6.6 Synthesis Environment

The synthesis performed in the evaluation uses the TSMC 7nm technology. This choice is not necessarily motivated by any specific tape-out of this design, but rather to get an idea of the performance using a relatively advanced technology node. The basic characteristics of this process node are shown in Table 4. The synthesis uses mixed threshold voltages in order to reduce space and power where performance is not needed while still using high performance standard cells in the critical paths. SVT corresponds to Standard V_t, LVT is Low V_t and ULVT is Ultra-Low V_t. The evaluation considers two corners, typical and slow, with the voltage and temperature conditions specified in Table 5.

For the multiple memories in the design, specifically those found in the caches, the Memory Compiler from Synopsys is used to generate Hard IP Blocks containing SRAMs. This reduces the area considerably as opposed to using registers. The SRAMs are optimized for high performance,

Characteristic	Value(s)
CPP	64
Tracks	7.5
Threshold Voltage	SVT, LVT, ULVT

Table 4: Characteristics of the technology process.

View	Voltage	Temperature
Typical	0.750V	85 ℃
Slow	0.675V	125 ℃

Table 5: Characteristics for the typical and slow corners.

and are subject to the same corners as specified previously. All the SRAMs macros are single port, in order to reduce the area, and the sizes are shown in Table 6.

Description	Ports	Depth	Width
iCache Tag Array	1	64	80
iCache Memory Array	1	256	128
dCache Tag Array	1	128	38
dCache Memory Array	1	256	128
dCache MSHR	1	64	104

Table 6: SRAM Macros used in the synthesis.

6.7 Evaluated Metrics

The metrics that are used in the evaluation are specified in Table 7. These are typical physical design metrics such as area, power and frequency, as well as some computer architecture metrics such as Instructions Per Cycle (IPC) and Millions of Instructions Per Second (MIPS), aimed at evaluating the computing performance of the design. A figure of merit is also computed to measure the overall benefit of the different optimizations. The figure of merit combines power, area and MIPS with the weight specified in Table 7. Note that the IPC and the frequency metrics do not have a weight assigned to them, marked with an asterisk (*), as they are combined in the MIPS metric which does have a weight and is included in the figure of merit.

Metric	Description	Weight
Power	Power consumption of the synthesized design	20%
Area	Area of the synthesized design	20%
IPC	Instructions per cycle executed in average in the benchmarks	*
Frequency	Maximum frequency in the slow corner of the synthesized design	*
MIPS	Millions of instructions per second, result of IPC times the frequency	60%

Table 7: Description of metrics used in the evaluation and their weight for the decision making.

The power metric comprises 2 parts: static and dynamic power. Static or leakage power refers to the power consumption of transistors even when they are off and not actively switching, such as the result of the leakage current flowing through the transistor. The dynamic power corresponds to switching a transistor on and off and charging and discharging its load. Because not all transistors are switching in every clock cycle, this value is modulated by an activity factor α, which intuitively means how much activity or switching of the transistors there is on average for each cycle. The reported power values are estimated by the synthesis tool using the default activity factor of $\alpha = 0.2$. This is quite a conservative factor, since a lot of the registers are found in the cache SRAMs, and aside from having "enable" signals which actively save power, they don't switch much of the time. However, because this is a rough approximation and there are no simulations nor place and route details, this is the best estimate.

The figure of merit is calculated as specified in Equation 1. The calculation incorporates the metrics of the previous table with their corresponding weights. Intuitively, the improvement in design quality is inversely related to its power and area requirements. In terms of what this metric represents, because of the units used, the seconds in MIPS and in the power (since watts are joules per second) cancel out. This leaves a metric which represents how many instructions are completed per unit of energy and area. In the units used in the evaluation, this corresponds to $10^6 Instructions \cdot J^{-1} \cdot mm^{-2}$.

$$f = \frac{0.6MIPS}{0.2Power \times 0.2Area} \tag{1}$$

7 Evaluation

This section evaluates Sargantana with the HPDCache and compares it against the baseline design using Lowrisc L1 data cache. The evaluation also considers all the optimizations, which are summarized in Table 8. Note that the designs with optimizations are incremental, that is, a version labeled as vX includes the optimizations labeled as vX-1, vX-2, ..., v1. Thus, the design labeled as v7 includes all the optimizations and is considered the final design.

Label	Optimization	Section
base	Sargantana baseline using Lowrisc caches	–
v1	Initial HPCD Integration	–
v2	Translation of an Incoming Instruction	5.1
v3	Bypass of the Load-Store Queue	5.2
v4	Register in Multiplier Unit	5.3
v5	Prioritization of Loads Over Stores	5.4
v6	Early and Back-to-Back Store Execution	5.5
v7	Final Critical Path Optimization	5.6

Table 8: Overview of the optimizations and the labels used for them in the charts.

7.1 Instructions Per Cycle (IPC)

Figure 19 shows the geometric mean of the IPC obtained when running all the benchmarks of the two benchmark suites, and the combined overall IPC. In the horizontal axis, the groups of bars represent a suite (or, in the third group, the geomean of both suites), while each individual bar in a group represents one of the different versions of the integration. The colors, as well as the order of the bars within a group, correspond to the version; first and in red is the baseline, last and in yellow is the latest optimization as described in 8. The vertical axis is the IPC.

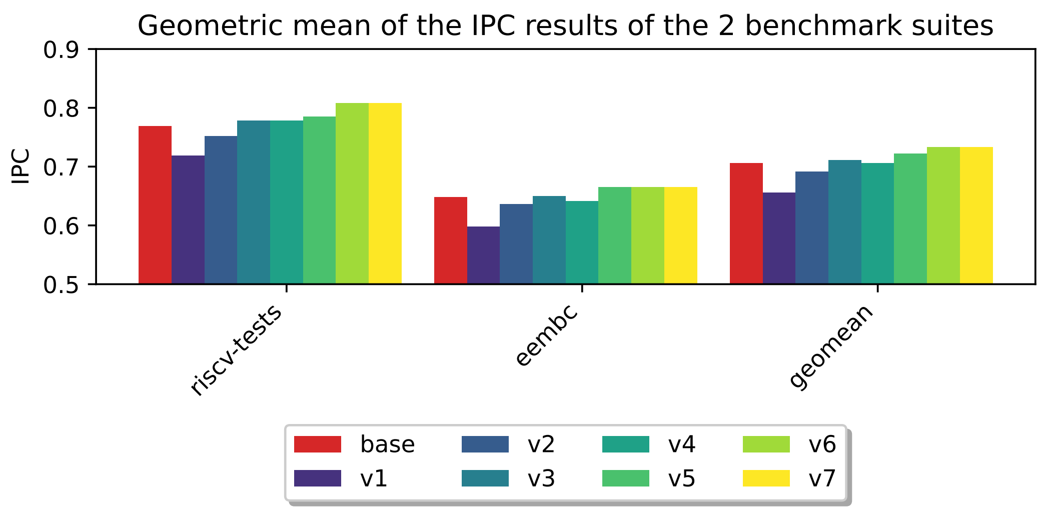


Figure 19: Geomean IPC of the different optimizations obtained in the benchmark suites.

It can be observed that the initial integration of the HPDCache performs worse than the baseline (the original Sargantana using Lowrisc caches). This is due to the extra cycles added by the TLB and the load-store queue. However, once these cycles are removed via the optimizations described in Sections 5.1 and 5.2, the performance is on par with the baseline. In fact, performance improves nearly by 4% with respect to the baseline after all the optimizations.

It is worth mentioning that this chart only shows improvement in the IPC, and thus the last optimization (described in 5.6) does not show an actual improvement. This is because this optimization reduces the critical path, aimed at improving the maximum frequency, without any impact on IPC. Then, this optimization will come into play when the frequency metric is evaluated. A similar thing happens to version 4, which degrades the IPC slightly, but will improve the frequency (and therefore the figure of merit).

7.1.1 RISC-V Test Benchmarks

Figure 20 shows the IPC of the benchmarks in the riscv-tests suite. The chart follows the same format as in the previous chart, in which in the X axis there are groups of bars representing each benchmark in the suite, with individual bars representing the different versions of the integration.

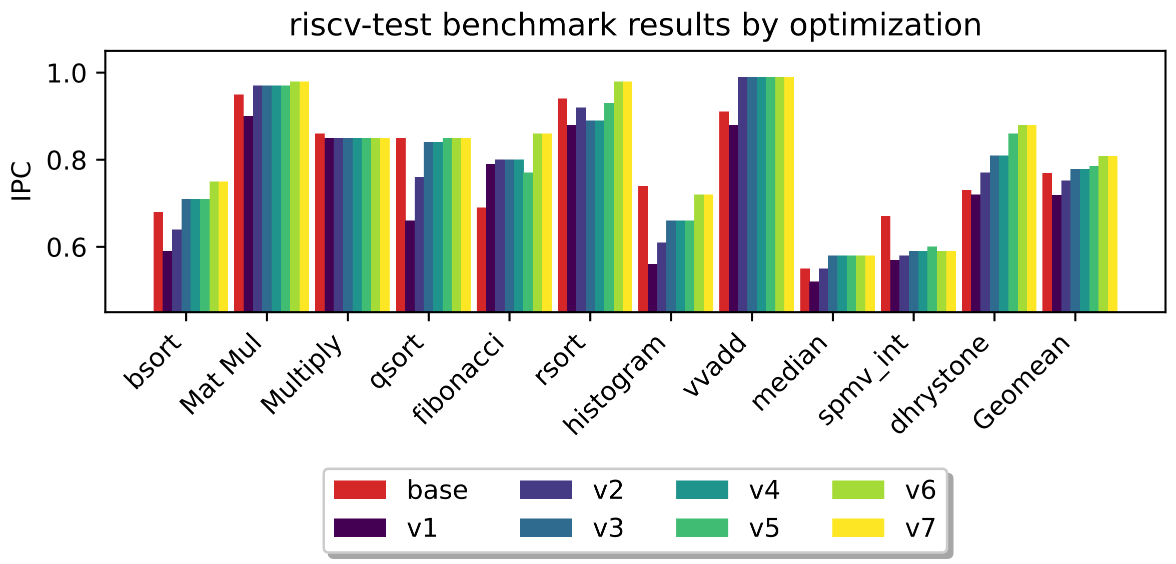


Figure 20: IPC of the benchmarks in the riscv-tests suite by optimization.

The results show that the initial integration of the HPDCache (v1) is slower than the baseline in all the cases except in fibonacci. However, the introduction of the optimizations gradually improves performance, and the final version with all the optimizations (v7) achieves large speedups over the baseline in many benchmarks such as bsort, rsort, vvadd and dhrystone. In two other cases, histogram and spmv_int, the baseline design with Lowrisc L1 dCache outperforms the one with the HPDCache and the optimizations.

7.1.2 EEMBC Benchmarks

Figure 21 shows the results for the EEMBC benchmark suite. The format of the chart is the same as for the previous benchmark suite. As can be seen, the situation is much similar as well. In general, the first implementation performs poorly, but with the last optimization provides a geometric mean speedup of 2.5%.

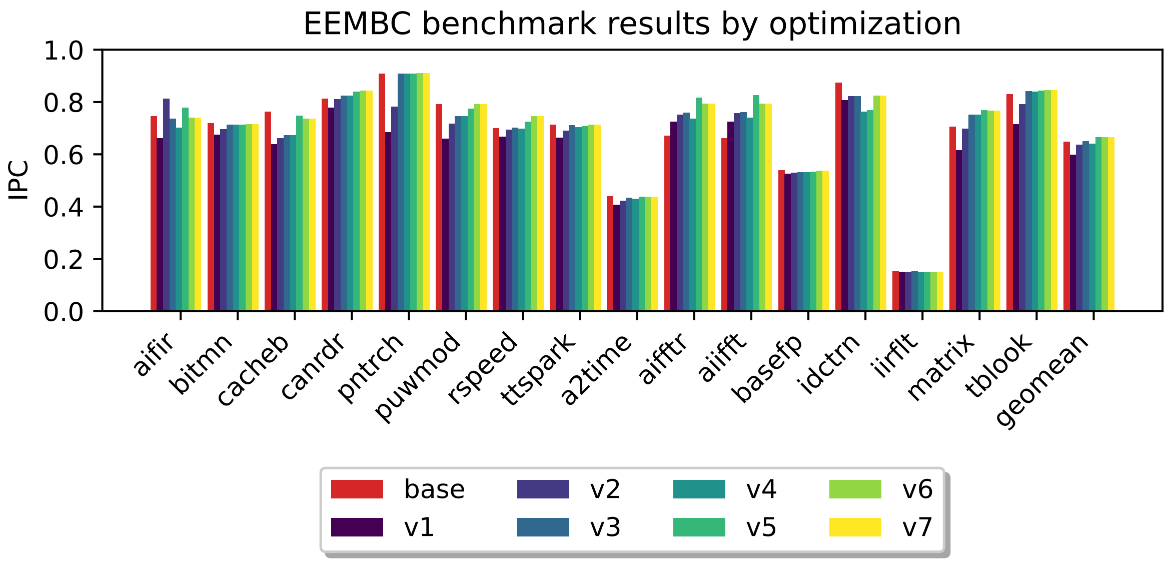


Figure 21: IPC of the benchmarks in the EEMBC suite by optimization.

One thing that can actually be seen in this chart, as opposed to with the other benchmark suite, is the slowdown that is introduced by the addition of the register at the end of the multiplication pipeline (label v4). This is specially relevant in computation-intensive workloads such as in benchmarks aifir, aifftr, aiifft and ictrn (FIR filter, fast fourier transform, inverse fast fourier transform and inverse discrete cosine transform respectively). However, this slowdown is compensated with the latter optimizations. Furthermore, because this register was added to cut a critical path, it will actually appear as beneficial in the following sections in which I show the frequency improvements.

7.2 Frequency

Figure 22 shows the maximum frequency in the slow corner of the different versions of the integration. As can be seen, the initial integration achieves the highest frequency, at close to 1.5GHz. This is because the initial integration was quite simple and conservative; all steps are done in different cycles and the critical path is at the output of the dCache SRAMs to the end of the execution stage. The initial integration is closely followed by the latest optimization that trims the critical path and thus has good maximum frequency results. In this case, and as mentioned in 5.6, the critical path is the one going from the start of the execution stage, through the address generation, the TLB, the exception checking and the dCache pipeline registers.

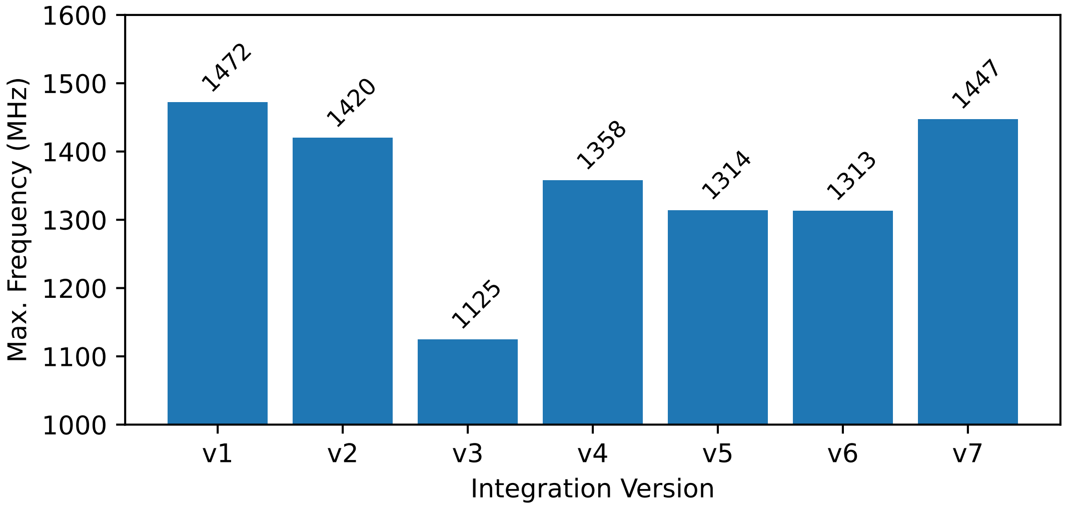


Figure 22: Maximum frequency in the slow corner by optimization.

The version with the worst maximum frequency is v3, at close to 1.1GHz. As mentioned in the development of the optimization of the next version, in section 5.3, this is because after this optimization (section 5.2) there is a path going from the end of the multiplication unit, via the bypasses of the memory unit, into the dCache pipeline registers. This clearly is too much logic and so the maximum frequency is considerably degraded.

7.3 Millions of Instructions per Second (MIPS)

Combining the results of the 2 previous sections, the MIPS metric takes into account the trade-off between IPC and the maximum frequency. As a reminder, MIPS, or Million Instructions per Second, is the product of IPC times frequency (in MHz). The MIPS achieved by benchmark suite, as well as the geomean of both suites, with the different designs and optimizations is shown in Figure 23. This plot clearly shows the effectiveness of the optimizations labeled as v4 and v7. In v4, the core has a slightly worse IPC but a significantly improved MIPS thanks to the improvement in the frequency domain. Similarly, v7 introduces changes that improve the frequency and do not degrade the IPC, resulting in a significant increase of MIPS.

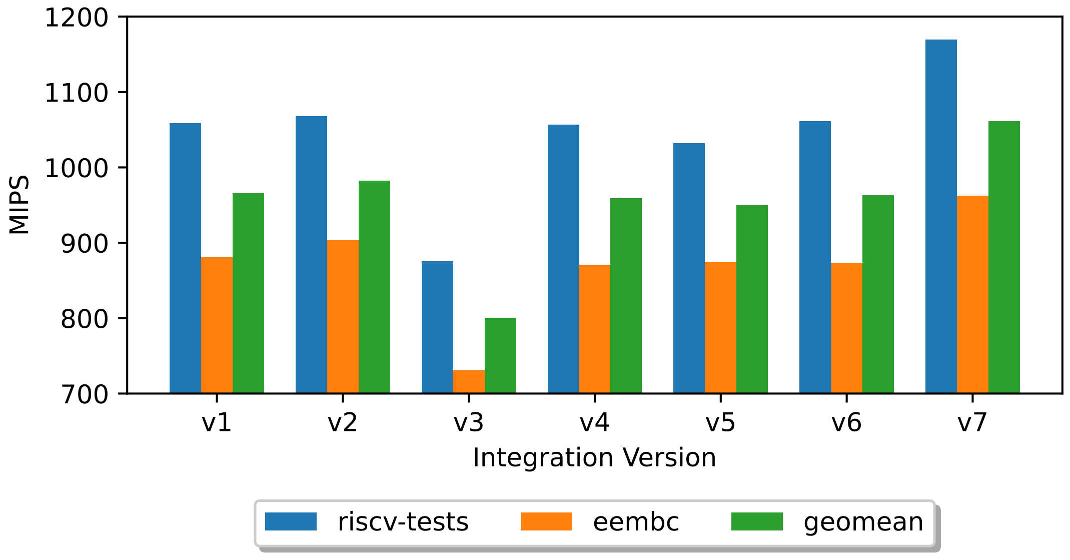


Figure 23: Average MIPS by optimization.

7.4 Area

Figure 24 shows the area of the different versions of the integration. Each bar on the horizontal axis is a different version, and in the vertical axis there is the final area of the synthesis in µm². The total area is split into some of the different components of the the core.

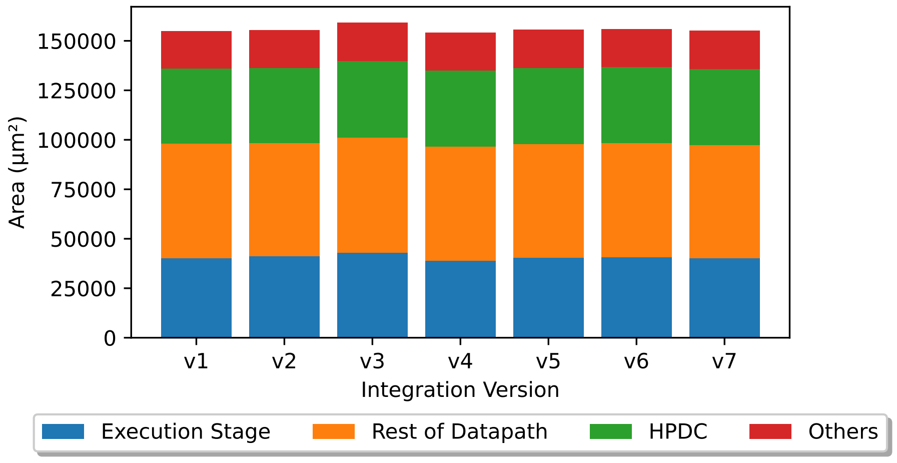


Figure 24: Occupied area by optimization.

Figure 25 shows the relative changes in area between optimizations, all of them normalized to the area of the baseline v1. As can be seen, the results do not have large changes; between the highest and the lowest result there is a difference of only 2.8%.

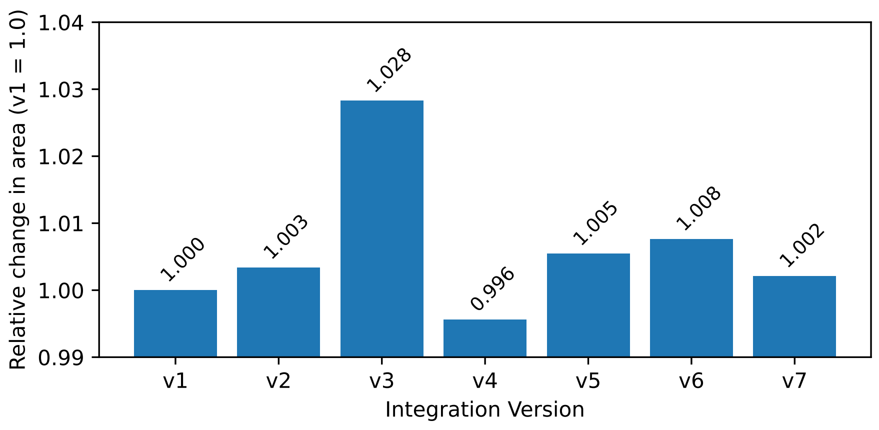


Figure 25: Relative change in area between optimizations.

The optimization v3 occupies significantly more area than the rest and, as shown in Figure 22, it also has the worst frequency results. In this optimization, most of the area increase is not because of change in the architecture itself (which is essentially adding an extra multiplexer), but rather because the tool is trying to achieve the target frequency and hence uses larger, more powerful, standard cells to reduce the delay of the critical path at the cost of a larger area. Indeed, the following optimization v4 eliminates this critical path and the area goes back down to values close to v1. In the following section, which discusses power, the usage of faster cells is further assessed.

7.5 Power

Figure 26 shows the power figure by optimization, with the same format as the previous plots. The horizontal axis shows the different versions of the integration and the vertical axis is the estimated static and dynamic power in milliwats. Again, the difference between the highest and the lowest result is low, at less than 5%. Another point to make is that most of the consumption is in static power. This is because the technology used is 7nm, which have smaller transistors and thus larger leakage currents due to quantum tunneling, short channel effects or the higher doping levels used.

As discussed in the previous section, in the integration v3 the synthesis tool is struggling to reach the target frequency, so it uses standard cells with a lower threshold voltage, shown in Figure 27. These cells are faster, helping lower the delay of the critical path, however they come at the cost of a higher leakage power. Thus, the static power of v3 is the largest one across all the optimizations.

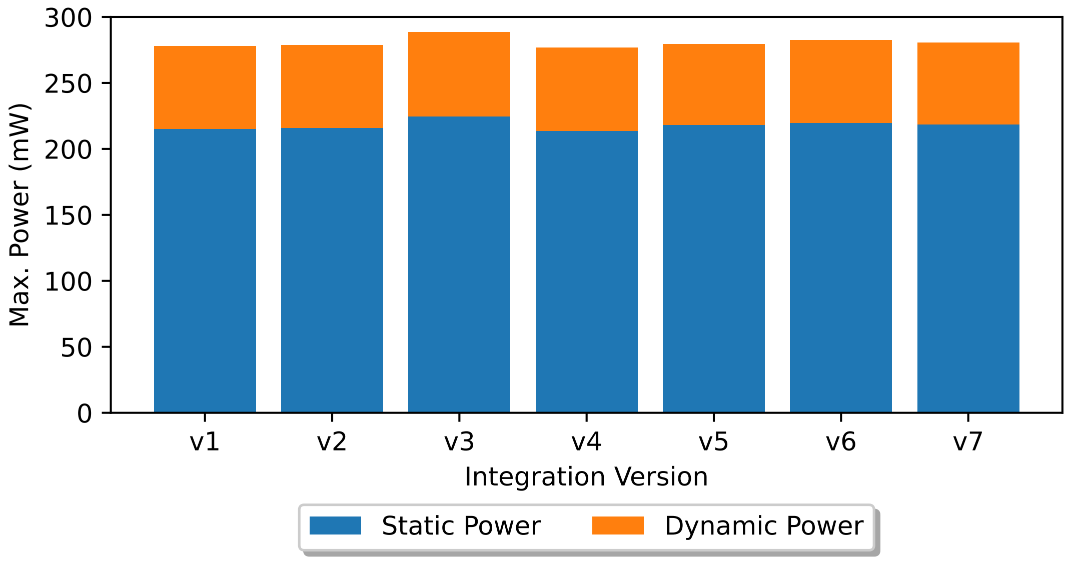
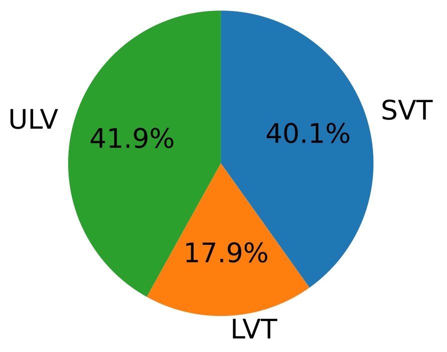
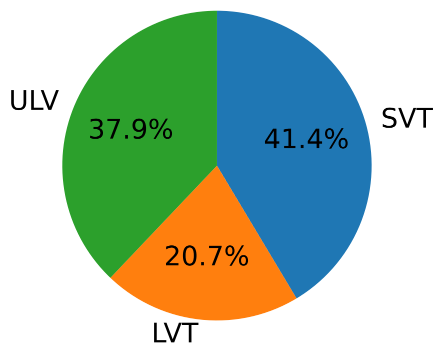


Figure 26: Power requirements by optimization.



(a) v3



(b) v4

Figure 27: Percentage of gates using Standard, Low and Ultra-Low Vt in v3 and v4.

7.6 Figure of Merit

Lastly, Figure 28 shows an aggregated figure of merit for the overall score of each version of the integration (by menchmark suite and the geomean of both suites). It is calculated from the previous metrics, following the formula and relative weights specified in the methodology section. Table 9 summarizes all the metrics. As can be seen, the initial implementation v1 does not fare well when considering all the metrics but, with some bypassing, the score improves quite a bit. While most of the optimizations improve one metric or another, when considering the aggregate, they seem to mostly cancel out and improvement mostly stagnates from version to version. However, doing the final optimization v7 in the critical path to improve the maximum frequency after doing all the architectural improvements proves worthwhile. The last version of the integration, v7, with all the improvements is clearly the best since it has the best figure of merit.

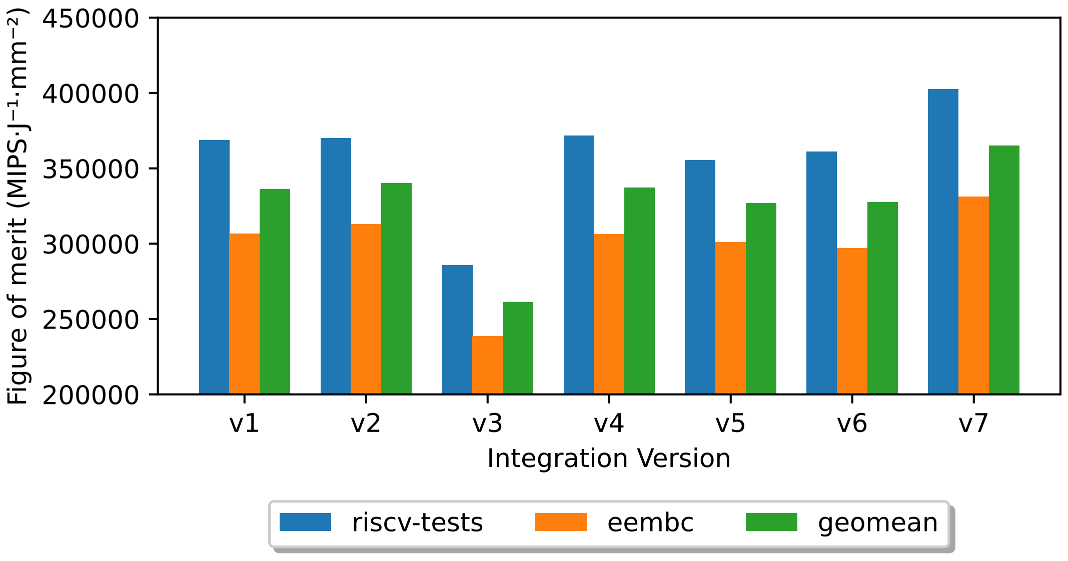


Figure 28: Figure of merit by optimization.

	Area	Fmax	Power	IPC			MIPS			Figure of merit		
	(mm²)	(MHz)	(Watts)	rv-test	eembc	geomean	rv-test	eembc	geomean	rv-test	eembc	geomean
v1	0,155	1472	0,278	0,719	0,598	0,656	1,06E+03	8,80E+02	9,65E+02	3,69E+05	3,07E+05	3,36E+05
v2	0,155	1420	0,279	0,752	0,636	0,692	1,07E+03	9,03E+02	9,82E+02	3,70E+05	3,13E+05	3,40E+05
v3	0,159	1125	0,289	0,778	0,650	0,711	8,75E+02	7,31E+02	8,00E+02	2,86E+05	2,39E+05	2,61E+05
v4	0,154	1358	0,277	0,778	0,641	0,706	1,06E+03	8,71E+02	9,59E+02	3,72E+05	3,06E+05	3,37E+05
v5	0,156	1314	0,280	0,785	0,665	0,723	1,03E+03	8,74E+02	9,50E+02	3,55E+05	3,01E+05	3,27E+05
v6	0,156	1313	0,282	0,808	0,665	0,733	1,06E+03	8,73E+02	9,63E+02	3,61E+05	2,97E+05	3,28E+05
v7	0,155	1447	0,281	0,808	0,665	0,733	1,17E+03	9,62E+02	1,06E+03	4,03E+05	3,31E+05	3,65E+05

Table 9: Table showing the results for all the metrics of the evaluation.

8 Related and Future Work

This work has been developed in the scope of a larger project. This section explains the scope of the project, the future work, and some relevant related work.

8.1 Project Scope and Future Work

This work is part of a project carried out at the Barcelona Supercomputing Center (BSC). The goal of this broader project is designing and developing HPC processors based on RISC-V, leveraging open source Intellectual Property (IP) blocks, improving them, and also developing new open source IP blocks that can be contributed to the community.

One of the key components of the HPC processors contemplated in the project is OpenPiton, an open source platform for building scalable manycore processor architectures [5]. This scalability is what makes the OpenPiton platform a good candidate for HPC, where applications extensively exploit parallelism. OpenPiton is generic and supports virtually any core [12], although the two cores currently supported are OpenSPARC T1 and Ariane.

Ariane [4], or more specifically CVA6, is an open source, 6-stage, in-order RV64GC RISC-V core developed by ETH Zurich. Ariane is one of the most widely-used open source cores, and has been previously taped-out using GlobalFoundries 22 FDX reaching 902MHz in the slow corner [4]. Although it has a very respectable performance, it is not geared towards HPC, and its architecture shows it. For example, its memory unit is not able to have more than 2 memory operations in flight.

In order to build an HPC multicore processor, the BSC is currently integrating the HPDCache with the Ariane core in the OpenPiton framework and is also performing multiple optimizations to the cache hierarchy and the on-chip interconnection network of OpenPiton [13]. The goal of such work is to increase the performance of the memory hierarchy of OpenPiton-based multicore processors. However, the cores are a key element of HPC architectures, and the characteristics of the Ariane core are a severe limiting factor.

To further improve performance of OpenPiton-based multicores, one of the next steps of the project developed at BSC is to change the Ariane core for in-house high-performance cores of the Lagarto family. The Sargantana core used in this work is the first processor of the Lagarto family and, although its performance is far from HPC standards, it is currently used as a base for the development of the next generations of the Lagarto core family. In addition, its early integration with the HPDCache and OpenPiton allows having a Linux-capable prototype for software development.

Lagarto Ka and Lagarto Ox are the second and third generations of the Lagarto core family developed at BSC. These are 2-wide and 4-wide out-of-order RISC-V cores, respectively, and they incorporate features typically found in HPC cores such as sophisticated mechanisms for branch prediction and speculation, deeper pipelines, vector instructions, etc. Many of the components of Lagarto Ka and Lagarto Ox are based on Sargantana, including the memory unit. Thus, the effort done in this project of integrating the HPDCache with Sargantana will pave the way and directly benefit the integration of the HPDCache with Lagarto Ka and Lagarto Ox. This will ultimately be a key feature of the OpenPiton-based multicores developed at BSC, so this work will contribute to the design and development of HPC multicore processors.

Another important aspect of this work is that it will be completely open source. The current design based on Sargantana with the HPDC will be available to the whole community and, hopefully, it will be used by engineers and researchers all around the world as a base for their prototypes, as well as by students to carry out their projects.

8.2 Related Work

Several groups and institutions are working on similar projects that aim at developing RISC-V multi-core processors. The next paragraphs summarize the most relevant prototypes developed so far.

BlackParrot [9] is a 64-bit RISC-V multi-core processor featuring a cache hierarchy with private L1 and shared L2 cache levels. It employs VI, MSI, and MESI cache coherency protocols and introduces specific-purpose tiles to enhance the L2 cache. Each L2 tile provides an additional slice of the L2 cache. Similar to OpenPiton, both share multiple memory controllers, a 2-D mesh using three physical channels NoC routers and lack virtual channeling. Unlike OpenPiton, BlackParrot integrates a network to connect memory controllers, and each L2 slice accesses this network. While system scalability is not detailed, a taped-out quad-core design exists.

The PULP platform [10] is a low-power SoC targeting IoT applications. It features a RISC-V core and an octa-core accelerator. Its cache hierarchy includes a 512KB L2 shared cache split into four banks. The RISC-V core lacks a private cache level but has two 32 KB banks for program stack and private data. The octa-core accelerator directly accesses the L2 cache and a scratch cache. PULP employs AXI-based interconnections for core and memory communication. Unlike OpenPiton, PULP is not focused on manycore systems.

Agiler [11] is a RISC-V multi-core architecture designed for heterogeneous systems, featuring two types of processing elements. The main type comprises a quad-core using AXI-based interconnection for communication between cores, shared instruction cache, and memory controller. The second type consists of accelerators with distinct tile architectures interconnected via mesh routers. This includes a 64-bit dual-core and a 32-bit quad-core RISC-V architecture, both internally linked with AXI. Tiles are mapped to memory regions, and tasks are allocated by loading data and instructions into corresponding memory spaces during compilation. In contrast, in OpenPiton, each tile works within the same memory space.

Open ESP (Open Embedded Systems Platform) [8] is an open-source framework for accelerator-rich SoC prototyping. In addition to providing tools and libraries to create software applications, it features a modular FPGA SoC architecture using tiles interconnected via a 2D-Mesh NoC with look-ahead routing. The four tile types (processor, accelerator, memory, and auxiliary) offer diverse functionalities. The processor tiles house a dual-cache core with MESI-coherent L2 cache, the accelerator tiles facilitate efficient data exchange with memory, the memory tiles include a shared LLC slice and a memory controller port, and the auxiliary tiles manage peripherals. Similar to OpenPiton, ESP supports multiple memory controllers and coherence protocols, enhancing scalability in manycores.

9 Conclusions

The outcome of this project is a successful integration of an open-source high-performance data cache with a RISC-V Core, achieving all the proposed objectives. For one, the integration is functional, passes all tests, completes all benchmarks and is able to boot Linux in an FPGA prototype. For the integration, 6 different optimizations have been proposed, both from the architectural and from the physical design point of view. Finally, the evaluation of the integration and its optimizations shows that the best design is 4% better than the baseline in terms of instructions per clock and it reaches a maximum frequency of 1.45GHz.

Among the he different optimizations, the final design is considered the best one, and it works as follows. The memory pipeline calculates the effective virtual address, translates it into the physical address, checks for any exceptions, bypasses the load-store queue if it is empty and, finally, it accesses the data cache through its internal arbiter and using the address to access the tag and data SRAMs. Because this is a long logic path, the final design also includes an optimization to reduce its delay and thus increase the maximum frequency. To do so, some priority encoders are re-arranged, some other priority encoder is converted into a regular multiplexer, and some others removed. All in all, this results in a design with excellent per-cycle performance and maximum frequency.

On a more personal and academic note, I have learned a lot about computer architecture, of which I already had some experience thanks to my degree in computer engineering, and, specially and about above all, about physical design. Without the knowledge acquired during my stay of 2 years in this master I would not have been able to do all the synthesis and physical design which have guided most of the optimizations presented in this work. In short, I think that, thanks to this project, I have become a much more competent and professional hardware designer. In addition, I consider that I have accomplished all the objectives I had set myself at the start of the project.

References

[1] Sarabjeet Singh and Manu Awasthi. “Memory centric characterization and analysis of spec cpu2017 suite”. In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 2019, pp. 285–292.

[2] Víctor Soria-Pardos et al. “Sargantana: A 1 GHz+ in-order RISC-V processor with SIMD vector extensions in 22nm FD-SOI”. In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE. 2022, pp. 254–261.

[3] César Fuguet. “HPDcache: Open-source high-performance L1 data cache for RISC-V cores”. In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023, pp. 377–378.

[4] Florian Zaruba and Luca Benini. “The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640.

[5] Jonathan Balkind et al. “OpenPiton: An open source manycore research framework”. In: *ACM SIGPLAN Notices* 51.4 (2016), pp. 217–232.

[6] Onur Mutlu. “Memory scaling: A systems architecture perspective”. In: *2013 5th IEEE International Memory Workshop*. IEEE. 2013, pp. 21–25.

[7] Ankur Limaye and Tosiron Adegbija. “A workload characterization of the spec cpu2017 benchmark suite”. In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2018, pp. 149–158.

[8] Paolo Mantovani et al. “Agile SoC Development with Open ESP”. In: ICCAD ’20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: `10.1145/3400302.3415753`. URL: `https://doi-org.recursos.biblioteca.upc.edu/10.1145/3400302.3415753`.

[9] Daniel Petrisko et al. “BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs”. In: *IEEE Micro* 40.4 (2020), pp. 93–102. DOI: `10.1109/MM.2020.2996145`.

[10] Antonio Pullini et al. “Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing”. In: *IEEE Journal of Solid-State Circuits* 54.7 (2019), pp. 1970–1981. DOI: `10.1109/JSSC.2019.2912307`.

[11] Ahmed Kamaleldin and Diana Göhringer. “AGILER: An Adaptive Heterogeneous Tile-Based Many-Core Architecture for RISC-V Processors”. In: *IEEE Access* 10 (2022), pp. 43895–43913. DOI: `10.1109/ACCESS.2022.3168686`.

[12] Jonathan Balkind et al. “BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 699–714. ISBN: 9781450371025. DOI: `10.1145/3373376.3378479`. URL: `https://doi.org/10.1145/3373376.3378479`.

[13] Neiel Leyva et al. “OpenPiton Optimizations Towards High Performance Manycores”. In: *Proceedings of the 16th International Workshop on Network on Chip Architectures*. NoCArc ’23. Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 27–33. ISBN: 9798400703072. DOI: `10.1145/3610396.3623265`. URL: `https://doi.org/10.1145/3610396.3623265`.

[14] Verilator. *Verilator*. 2023. URL: `https://github.com/verilator/verilator`.

[15] UC Berkeley Architecture Research. *RISC-V Torture Test*. 2021. URL: `https://github.com/ucb-bar/riscv-torture`.

[16] Jason A Poovey et al. "A benchmark characterization of the EEMBC benchmark suite". In: *IEEE micro* 29.5 (2009), pp. 18–29.

[17] Bradford M Beckmann and David A Wood. "Managing wire delay in large chip-multiprocessor caches". In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE. 2004, pp. 319–330.