Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Valentin Plotkin

Implementation and Evaluation of Data Filter Cache for a RISC-V processor

Master's thesis in Computer Science
Supervisor: Magnus Själander
June 2020

NTNU
Norwegian University of Science and Technology

Valentin Plotkin

Implementation and Evaluation of Data Filter Cache for a RISC-V processor

Master's thesis in Computer Science
Supervisor: Magnus Själander
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Improving energy efficiency is the major goal of contemporary processor microarchitecture design. The memory subsystem, being responsible for a significant portion of total energy consumption, is a natural candidate for applying power consumption-improving techniques.

Most publications on novel microarchitecture improvements, including those of memory hierarchy, usually design and evaluate their proposals using analytic models and high-level simulations, rather than a functional hardware implementations, as such implementations can be quite expensive and labor-intensive. While such high-level modeling might give reasonably accurate estimates of metrics like speed and power consumption, it might not tell us all the truth about real-world feasibility. Integrating a new enhancement into an existing processor might turn out harder than anticipated from the concept of the enhancement alone. Integration might also introduce some unexpected cost, reducing metrics from what is predicted by models. Hence it is important to test newly proposed enhancements in real life, production-ready processors.

This thesis describes our implementation of Data Filter Cache (DFC), a technique for improving performance and power consumption in low-power embedded processors, for the VexRiscv core, a production-ready open-source implementation of RISC-V ISA. Our implementation provides new insight into the practicability of integrating a DFC into a real-world microarchitecture. In particular, we found that line filling is not easily integrated into the VexRiscv pipeline.

Our implementation using fast speculative address calculation and 8-way lookup achieved a hit rate of 18% on Dhrystone benchmark, while running at 78 MHz vs. 94 MHz for the unmodified core, a 17% reduction in frequency. This could have been improved to 23% with instant fill implementation. Using full address calculation and 8-way lookup achieved 35% hit rate at the cost of further reducing frequency to 70 MHz.

Acknowledgment

First, I would like to thank my supervisor, Magnus Själander, for his invaluable guidance. I would also like to thank my fellow students David Metz, Erling Jellum and Khakim Akhunov for sharing their discoveries and for their support.

This thesis would not have been possible without the work of Dolu1990 and other VexRiscv project contributors. I thank them for their work and for explaining some aspects of the VexRiscv architecture.

V.P.

Contents

Abstract . . . . **i**
Acknowledgment . . . . **ii**
Contents . . . . **iii**
1 Introduction **1**
2 Background **2**
2.1 Memory hierarchy and cache organization 2
2.2 The classic RISC pipeline and RISC-V ISA 4
2.3 Data filter cache 4
2.4 SpinalHDL and the VexRiscv microarchitecture 5
2.5 Field Programmable Gate Arrays as hardware prototyping platforms 7
3 Implementation **8**
4 Methodology and results **10**
5 Discussion **13**
6 Conclusion and future work **14**
Bibliography **15**
A List of acronyms **17**

1 Introduction

Computers have become ubiquitous, with processors as their most important parts. From the largest supercomputers to microcontrollers, the most important factor limiting their performance is power consumption [1]. In large machines the processor frequency and the number of active cores are limited by the capacity of heat removal, as chips are damaged by high temperature. Battery-powered applications, such as smartphones, are extra sensitive to their energy usage, but even stationary embedded systems, running 24 hours a day, cannot afford to waste much power. Hence reducing power consumption is an important goal for microarchitecture designers.

Up to 25% of the power consumption by embedded processors is due First Level Cache (L1) data cache accesses [2] [3]. The Data Filter Cache is a technique to reduce L1 data cache access by introducing a smaller cache that checked before L1 access. An way to integrate a DFC into a processor without incurring a time penalty on DFC miss was proposed by [4]. Power consumption of such a DFC was modeled [5] for a $65\,\mathrm{nm}$ layout, with promising results. Yet the extent of core pipeline and L1 modifications necessary for implementing a DFC remained unknown.

VexRiscv [6] is an implementation of the open-source RISC-V [7] instruction set architecture. Unlike most conventional processor implementations, VexRiscv is designed to be flexible and extensible, allowing to modify many aspects of the implementation (such as number of pipeline stages, hazard and bypass logic, memory hierarchy interface) or introduce a new feature while reusing most of the existing code without modification. This is made possible by the feature-rich Spinal-HDL hardware description language and an extensive plugin and service systems. We believe that implementing a novel microarchitecture feature for VexRiscv gives an lower bound on the extent of modifications that would be required for a more traditional microarchitecture.

This thesis summarizes the work done during our master project conducted at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU).

2 Background

2.1 Memory hierarchy and cache organization

In almost all modern computers the high memory latency is the main factor determining the overall architecture. The time it take the main memory to provide data requested to the processor is substantially higher compared to time used by CPU operations, and the gap is growing since DRAM technology is advancing slower than logic. If the processor would have accessed the main memory every time the program requests to read the stored data, it would spend most of the time waiting for the response — and the performance would be unacceptably low. This is the so-called memory wall [8] [9].

Various techniques are applied in processor design to alleviate the bottleneck associated with the memory wall. The technique most commonly applied on CPUs is caching, as it retains compatibility to existing software and programming techniques. A cache is a smaller and faster memory which contains a subset of the data in the main memory. Requests for data that are present in the cache are served from there, eliminating the main memory latency. This is called a *cache hit*. If the requested data are not present in the cache, a request to main memory is needed, respectively called a *miss*.

Since there is a trade-off between memory size and latency, a typical desktop contains a hierarchy of caches of increasing size and latency, where a miss at a lower level trigger a more expensive request to the higher level. Together with main memory (and eventual secondary storage) these form the *memory hierarchy*. The fastest and smallest cache, closest to the CPU in the memory hierarchy, is denoted L1 (and higher level caches are correspondingly L2 and possibly L3). CPUs typically contain two L1 caches — one for instructions and one for data. This arrangement, called *Modified Harvard Architecture*, allows to read data and instructions in parallel and reduces cache conflicts. The read latentcy of an L1 data cache is typically two cycles, meaning that the data present in L1 would be avaliable within two cycles after a request for them is issued.

Caches are also important for power dissipation. Accessing the main memory requires a lot of energy, hence cache hits reduce energy usage. The cache accesses themselves, however, consume power too, and contribute a substantial part of total consumption in embedded processors, with their simple logic [2] [3]. Therefore it is important to be aware of the energy requirements when designing a memory hierarchy. The power consumption increases with memory size and a wider memory, that returns a larger piece of data per requests, consumes more than a narrower memory of same size.

There is a number of choices when designing a cache, aside from its size. The most important one is what and when to store in the cache. An approach taken nearly universally is to cache the recently requested data, as well as data with addresses close to the recently requested, thereby exploiting *locality of reference*. In real-world workloads, data accessed in the past are more likely

to be accessed in the future (*temporal locality*). Data close to the recently accessed one is also more likely to be requested soon (*spatial locality*). Hence, the cache is organized into *lines*, aligned continuous areas of memory typically 32 to 128 bytes long. When a cache miss happens, a new line is allocated, and when the response is received from the higher levels of memory hierearchy, the line is filled with the received data.

Another important decision is how to store and find the data in the cache. The most straightforward is to store a line alongside with its address and check all lines in the cache when a request is issued. Such *fully associative caches* are extremely efficient in terms of *cache conflicts*, as no conflicts arise - any subset of main memory that can fit the cache in terms of size may be present in the cache. However such content-addressable memory takes a lot of chip area and checking every line is prohibitively slow and expensive in terms of power for caches larger than about 64 lines, depending on the technology.

An opposite extreme is to only allow every memory location fit into a single cache line. Cache can then be implemented as a power-of-two-sized random access memory. Lower bits of requested address (the *index*) are used to index the cache, while the higher bits (the *tag*) are compared to the tags stored together with the data to ensure that the cached line belongs to the right address. Called *direct-mapped cache*, this arrangement is quite efficient in terms of speed and chip area, but is prone to *cache conflicts*: a newly fetched line would force out the old line with the same index even if there is enough space in the cache to store both, hence a program alternatively accessing data one cache line size apart would miss every time.

Most cache implementations use a *set-associative* cache organization, which is somewhere in between the two extremes. Set-associative cache can be described as several, typically two to eight, direct-mapped caches (or *ways*) looked up in parallel. A single *set* of lines share the same index but contain multiple tag/data combinations spanned across ways. A request first fetches all tags at the requested index, then searches them for the requested tag and fetches the corresponding way (or reports a miss if no way matches). Set-associative caches provide a compromise: the conflict rate is much lower, though conflicts still do arise, and the implementation costs are low.

A subtle aspect which is still quite important for our work is the interaction between caches and virtual memory. The addresses issued by a program executed by processor core are called Virtual Addresss (VAs). In order to isolate different programs from each other, reduce physical memory fragmentation and allow to transparently offload unused memory to secondary storage, the VAs are translated into Physical Addresss (PAs). The lower bits of a VA are keept intact, as virtual memory is organized in continuous *pages*, typically 4 kB long, corresponding to 12 bits of VAs shared with a PAs. The upper bits are looked up in the Translation Lookaside Buffer (TLB), a specialized form of cache, typically implemented fully associative. Different VAs may map to a single PA, so-called *memory aliasing*, and a cache tagged and indexed by the VA would return invalid results in such case. Using physical address for both index and address works, but requires an expensive TLB lookup before every cache requests. The typical solution used in most of today's processors is to use the lower bits shared between PAs and VAs for indexing and PAs for tagging, restricting the span of a single cache way to one page. Thereby the TLB lookup can be done in parallel with tag fetch,

reducing latency. However, TLB lookup is still expensive in terms of power.

2.2 The classic RISC pipeline and RISC-V ISA

All modern processors are pipelined, which means that execution of next instruction starts before the previous instruction is complete. As much of instruction processing are independent from the preceding ones (i.e. General Purpose Register (GPR) that are not modified by the previous instruction can be read right away), allowing to keep the hardware busy most of the time and achieve higher frequencies due to shorter combinatorial parts. The combinatorial logic of pipeline *stages* alternate with registers that store the state of the instruction in the stage at the end of a clock cycle and provide it to the next stage in the next clock cycle.

The core idea of Reduced Instruction Set Computer (RISC) is to make all instructions to proceed in uniform stages, resulting in a simple pipeline structure. The RISC-V Instruction Set Architecture (ISA) [7] was designed with the classic RISC pipeline in mind. It assumes that most instructions can be executed by reading by at most two GPR, preforming an arithmetic action with the registers, possibly issuing a memory request with address given by the result and finally writing the result into a GPR.

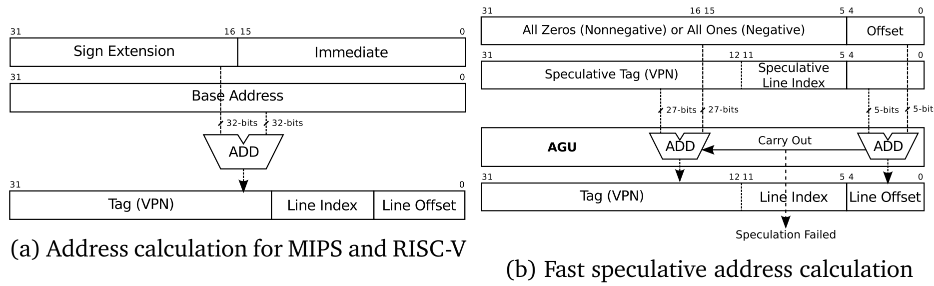
The classic RISC pipeline consists of five stages: fetch, decode, execute, memory and writeback. Register reads are issued by the decode stage and the read values are available to the execute stage. Address calculation (or other arithmetic) happens in execute stage and memory requests might be issued in the execute or memory stage depending on the implementation. The result is available at writeback. *Hazard detector* stalls instructions that read a register that is yet to be written by an uncommitted previous instruction. Alternatively, a *forward network* (also called *bypass*) might provide the register content if it is already available but not yet written: i. e. a purely arithmetic instruction can forward it result to the next one while in memory stage.

2.3 Data filter cache

The idea of the DFC as proposed by [4] is to provide a small fully associative cache looked up before the L1 cache. In case of a hit, the power-expensive L1 access is eliminated. Also, the results of the read can be immediately forwarded to the next instruction, thereby avoiding load-to-use stalls and improving performance as well.

Since the DFC have to be small, miss rate is quite substantial. Therefore it is unacceptable to stall the pipeline on a DFC miss: the DFC lookup must happen in-pipeline, without any miss penalty compared to not using a DFC at all. In order to improve power consumption, DFC lookup must happen even before the L1 access is triggered. Hence the only place in the RISC pipeline where DFC read would be useful is in the execute stage.

Associative lookup requires time even on smaller caches, but in order for the DFC to be useful it must fit into one stage together with address calculation. Fast speculative address calculation by [5] allows to alleviate the resulting timing penalty somewhat. To calculate the full address, which in case of RISC-V is the sum of 12-bit sign-extended offset instruction field with a 32-bit base register value, a full 32-bit carry propagation is necessary. Speculative address calculation



(a) Address calculation for MIPS and RISC-V

(b) Fast speculative address calculation

Figure 1: Address generation. Reproduced from Bardizbanyan et.al.[5]

involves preforming addition and carry propagation for the first five bits of the address, OR-ing the remaining 7 bits of the constant with the register:
True address $= \text{base}[31:0] + \text{offset}[11:0]$,
Speculated address $= \text{base}[31:12] \mathbin{+\!\!+} (\text{base}[11:5] \vee \text{offset}[11:5]) \mathbin{+\!\!+} (\text{base}[4:0] + \text{offset}[4:0])$,
with $\mathbin{+\!\!+}$ denoting concatenation (see Figure 1). Assuming the offset is to be a small positive integer (which is usually the case), the speculated address equals to the true address, otherwise a DFC miss is assumed. Using the speculative address for DFC lookup relaxes timings, as only a 5-bit carry is to be propagated.

In order to further reduce power consumption, virtual tags are used for the DFC. This eliminates the need for a TLB lookup in case of a DFC hit. To solve the problem of virtual address aliasing, the DFC is *strictly inclusive* with respect to L1: every line present in DFC must also be present in L1. A line evicted from L1 must also be evicted from DFC if present there.

2.4 SpinalHDL and the VexRiscv microarchitecture

The conventional Hardware Description Languages (HDLs) Verilog and VHDL are quite verbose and somewhat limited in their expressiveness. They offer little ability to manipulate compound data structures and require routing between modules to be done manually. While there are some facilities to generate hardware, these are quite limited, frequently requiring manual description. Also, their event-based execution model is much more complex than what is required to describe synchronous designs universally used today, leading to hard-to-find errors.

This led to development of several embedded HDLs, which use a general-purpose high-level language to construct a data structure describing the hardware, then converts the description into a RTL representation. Chisel [10] is one such language, based on Scala. The original implementation of RISC-V, the Rocket Chip SoC [11], is written in Chisel.

Originally we considered using the Chisel/Rocket infrastructure for our project. The experience is summarised in our half-year project report [12]. There was two main issues: the fragmentation of Rocket infrastructure (caused by the author team splitting) and the apparent lack of coding standards in the project. Rocket seems to empathize the "Don't repeat yourself'" principle at cost of both readability and expansibility.

SpinalHDL is an offshoot of Chisel, using the same Scala platform, but fixing some of Chisel's idiosyncrasies, such as lack of implicit conversions and issues with partial assignments. More importantly, it is the language of the VexRiscv processor [6], which also implements the RISC-V ISA, but takes a substantially different approach.

VexRiscv utilizes the facilities provided by Scala to build a fully modular processor. The various microarchitecture features, such as ISA subsets, hazard and bypass logic, memory subsystem and instruction fetcher are implemented as separate *plugins* which are assembled into the final pipeline. Cross-cutting concerns such as data dependencies, instruction format, branches and exception handling are handled by *services*. At elaboration a plugin provides a list interstage registers and services it requires. The elaboration creates the registers before instantiating the plugin. This results in a modular and configurable pipeline without forcing to specify the interconnect manually and allowing for high degree of customization such as changing number of stages or to use no pipelining at all. That also allows to extend the processor without much non-local code editing. In comparison, most of the Rocket core pipeline is implemented in a single-module, with little room for extension or customization.

We must note that Rocket Chip is silicon-proven, promoted by academia and adopted by multiple commercial companies (which is also one of the causes behind infrastructure fragmentation), while VexRiscv is a FPGA-oriented processor by a single developer, Dolu1990. Hence comparing these is not completely fair, as the larger scale of Rocket Chip necessary results in some code blowup. Nonetheless, the functionality provided by the two processor cores is more or less the same. We believe that Vex approach is the best practice to use the novel possibilities of embedded Hardware Description Languages.

Some of the numerous VexRiscV plugins and services are specially relevant for this work and will be elaborated upon. Plugin that implement instructions announces these instructions by calling `DecoderService`, specifying values of pipeline registers corresponding to the instruction. The `DecoderSimplePlugin` collects the announces and inserts the requested registers at the decode stage.

A set of pipeline registers is used to detect and handle data dependencies. These specify GPRs the instruction reads from and writes to. The `REGFILE_WRITE_DATA` pipeline register contains the result value to be written to the destination register. In addition, there are two flags, `BYPASSABLE_EXECUTE_STAGE` and `BYPASSABLE_MEMORY_STAGE`, which specify whatever `REGFILE_WRITE_DATA` is valid at the corresponding stage, allowing to bypass its value. `HazardSimplePlugin` is responsible for detecting the dependencies and bypassing values when possible or stalling the pipeline if the requested value is not yet available.

The `DBusCachedPlugin` is one choice for implementing memory access instructions, `DBusSimple` plugin being another one. Specially, `DBusCachedPlugin` instantiates a separate `DataCache` module, which implements a set-associative L1 data cache with customizable parameters such as total cache size, line size and number of ways. `DataCache` communicates with the `DBusCachedPlugin` and the core pipeline via three buses, one for each of the execute, memory and writeback stages.

2.5 Field Programmable Gate Arrays as hardware prototyping platforms

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured to implement arbitrary digital logic from a Register Transfer Level (RTL) description. FPGAs are universally used for fast and inexpensive processor prototyping. FPGA synthesis is much less labor-intensive compared to IC synthesis and layout process, yet it gives some insight into possible issues of the specific design and allows to interact with real hardware for a real-world functionality test. However, there are some important information that FPGA prototyping does not provide.

Power consumption of an FPGA implementation carries no information about power consumption of eventual IC implementation. The power overhead of FPGA fabric is such that it masks any useful signal, making any power measurements meaningless. This is particularly relevant for our work, as the DFC is first and foremost a power-saving enhancement. A power usage estimate for an isolated DFC was provided by Bardizbanyan et.al.[5]. Assuming power consumption of added pipeline control logic to be negligible, this estimate together with hit rates from our implementation can be used to provide a reasonable full system power usage estimate.

Chip area usage and maximal frequency are somewhat more meaningful, however the limitations imposed by FPGA architecture must be taken into the account. FPGAs provides registers virtually for free and FPGA memory blocks are much faster than random logic. On the other hand, IC standard cell library might provide fast implementation of standard blocks as content-addressable memory or asynchronous-read RAM, while FPGA would be implemented with the slow random logic. Yet the reports from an FPGA flow tools provides some insight into timing and area costs of a logic design.

3 Implementation

A subset of Data Filter Cache was implemented for the VexRiscv processor. The main modification was the introduction of `DBusDFCPlugin`, which is based on the existing `DBusCachedPlugin` but also implements a fully-associative, nonMRU write-through, allocate-on-write, no-fill DFC. The L1 implementation (`vexriscv.ip.DataCache`) was extended to provide information about evicted data and to provide the L1 line number on read. These are necessary to enforce the full inclusion property.

The DFC *area* within `DBusDFCPlugin` implements a Data Filter Cache with a single read port, a single write port and a separate L1 eviction report bus. The implementation consists of storage registers, tag comparators, data multiplexer, write logic and eviction logic. Validity flags for the whole line as well as for each word within the line are stored alongside the data. Since L1 is physically tagged while DFC uses virtual addresses, the L1 way number is also stored within each DFC line to enable eviction detection without looking up the physical address.

The DFC area also implements forward logic to ensures correctness in cases when the same location is written to and read from in adjacent instructions. Since the VexRiscv framework only provides bypass service for register write data, we implemented our own forwarder from scratch. The forwarder checks for write instructions with matching address in memory and writeback, assembling the correct word value depending on the mask, to also enable handling consequential writes to different bytes of the same word. The forwarder does not account for the flexibility of Vex core, so pipeline configurations other than the classic 5-stage are unsupported.

New registers `DFC_DATA` and `DFC_HIT` were added to the pipeline. The execute stage logic of `DBusCachedPlugin` was modified to issue a read request to the DFC, write the result to the pipeline registers and disable initiating a L1 lookup in case of a DFC hit. A multiplexer was added to choose between the `DFC_DATA` and the L1 response at the writeback stage based on the value of `DFC_HIT`.

Writes to the DFC are preformed at the writeback stage. An alternative implementation that preforms writes earlier was explored as it would eliminate the forwarding logic. However this raised the issue of canceling side effects of speculated instructions and exceptions, as well as worsened timings, so this approach was abandoned.

Other modification of `DBusDFCPlugin` involved introduction of performance counters to measure hit and miss rates and factoring out to enable code reuse for functions that are duplicated in the DFC and the old plugin.

We were unable to introduce bypass of the values provided by the DFC, so the performance improvement of DFC is not exploited by our implementation. The DFC hazard detection framework, while quite flexible in itself, is somewhat limited by some conventions adopted by other modules. In particular, the `IntAluPlugin`, responsible for handling simple arithmetic expressions, assigns to

the result register (`REGFILE_WRITE_DATA`) at the execute stage even if the executed instruction is not arithmetic. Since no other module assigns `REGFILE_WRITE_DATA` at this stage, that causes no problem: the value is overwritten at later stages. However, trying to assign the value returned by the DFC the register resulted in a conflict. This can be fixed by modifying `IntAluPlugin` check if the executed instruction is within its competence, but that caused more problems at other modules. Assigning to the result register before the writeback stage (ever without setting the bypass flag, hence the assigned value is never marked as valid) resulted in test errors, that are probably caused by interaction between hazard unit and branch prediction. Hence it is either that we do not understand how to use the forward unit or it is simply unsuited to such use.

The most important deficiency in our implementation is the lack of line filling. When a single word is requested by the processor core, only this word is brought into the DFC, not the rest of the data in the same line. Hence spatial locality is not exploited by the DFC, which results in a significant penalty. Implementing line filling would require extensive modifications to the `DataCache` unit. The L1 pipeline, tightly matching the three last stages of the core pipeline, provides no way to fetch the rest of an opened line, requiring a TLB and way lookup going through three stages on every request. A modification that stalls the execute-L1 and memory-L1 stages and complete line fill would be required. Arbitration between the core, DFC and the memory bus would raise several choices, in particular whatever a line fill should be aborted on a read request from the core and whatever the line filler should wait for the response from the memory bus when a line is brought into the L1 and the DFC simultaneously. Filling the lines lazily, as described later in this thesis, would elude the issue of arbitration, but still require some modification to the L1 read logic.

There are several processor configurations provided with the VexRiscv distribution. `GenFull` is the one used for regression tests, which we modified by replacing data cache with `DBusDFCPlugin` to get `GenFullDFC`, used for the simulation. `VexRiscvAvalonWithIntegratedJtag` is the FPGA-oriented configuration that matched our board best, so we used it as a basis for our setup.

4 Methodology and results

The implementation was tested using the VexRiscv regression test suite for different DFC sizes and address speculation schemes:

- Full address calculation, by preforming 32-bit addition
- Only using the base register value for speculated address (BASE)
- Fast speculative address generation with 5-bit addition and 7-bit bitwise OR (OR)
- Speculative address generation with 12-bit addition

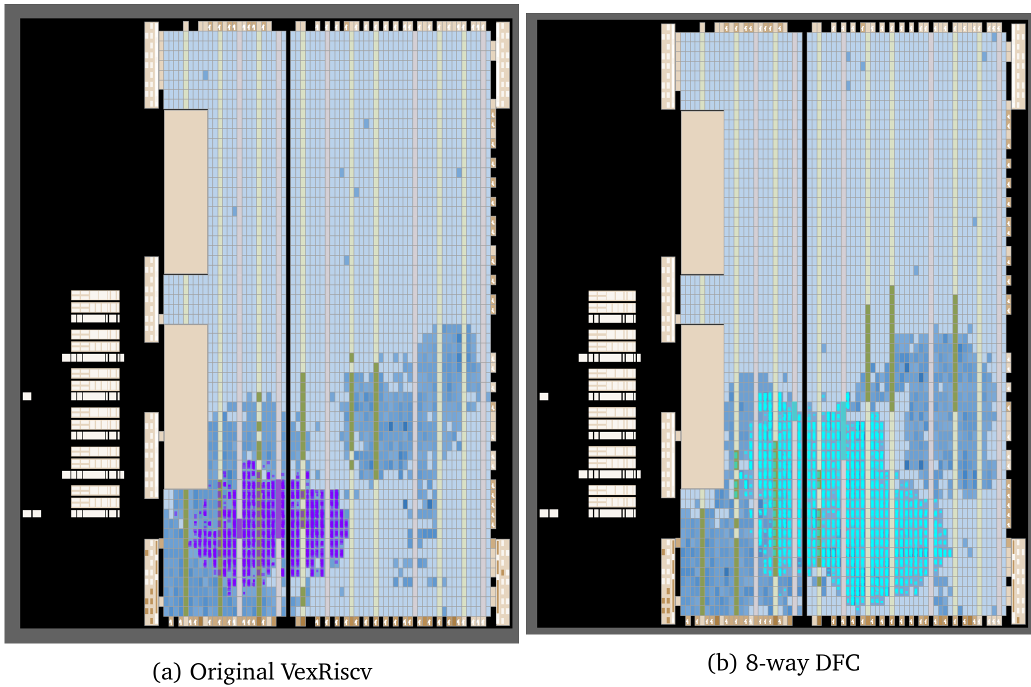
The test results matched those of the original core, hence our DFC implementation is probably correct in that it does not visibly disrupt the functionality of the processor. The DFC hit rates for the Dhrystone benchmark [13] as provided in the test suite are reported in Figure 3. The column labels indicate the number of DFC ways with address speculation prefix. The configuration used for the tests was based on `GenFull`, with 32 bytes per cache line and a direct-mapped $4\,\mathrm{kB}$ data L1.

Additionally, the number of misses caused by no line filling was measured on the 8-way fast speculative address configuration. It contributed 6% of misses. Hence implementing line fill could improve the hit rate by almost 30%.

A functional FPGA system with a 8-way DFC was synthesized and tested for different DFC address speculation schemes. The CPU was based on the original VexRiscvAvalonWithIntegratedJtag, but used a $4\,\mathrm{kB}$ 2-way data cache, a virtual memory subsystem and DBusDFCPlugin with 8-way DFC. The system also included Altera DRAM controller and IO peripherals. The system was synthesized using Intel Quartus Prime version 19.1.0 Lite Edition targeting 5CGXFC5C6F27C7N device and run on Cyclone V GX Starting Kit board. The maximal clock frequency as computed by Quartus Timing Analyzer is reported in Figure 4. The FMAX of the original core with same L1 and peripherals setup is labeled as `noDFC` in the figure. For the purpose of evaluating timings, the target core frequency was set at $100\,\mathrm{MHz}$.

The 8-way DFC-enchaced VexRiscv core used about 3500 ALMs and 4300 registers, with difference between difference between various address generation schemes being below 100 ALMs. The complete system used about 6000 ALMs and 9400 registers, with the LPDDR2 memory controller being the second largest unit after the processor, using 2000 ALMs and 2400 registers. In comparison, the unmodified VexRiscv system with same configuration used about 4700 ALMs and 6700 registers, only about 1600 ALMs and as roughly as many registers used for the processor itself. The Figure 2 shows the FPGA floor plans for both the unmodified VexRiscv and the DFC version.

The critical path of the DFC-enabled cores was at the DFC read lookup regardless of the address generation scheme. The worst-case negative slack is $1.7\,\mathrm{ns}$ for the core with fast speculative address calculation enabled. The original core was limited by the memory controller and L1 cache, with $0.2\,\mathrm{ns}$ of negative slack.



(a) Original VexRiscv

(b) 8-way DFC

Figure 2: FPGA floor plan usage with processor highlighted

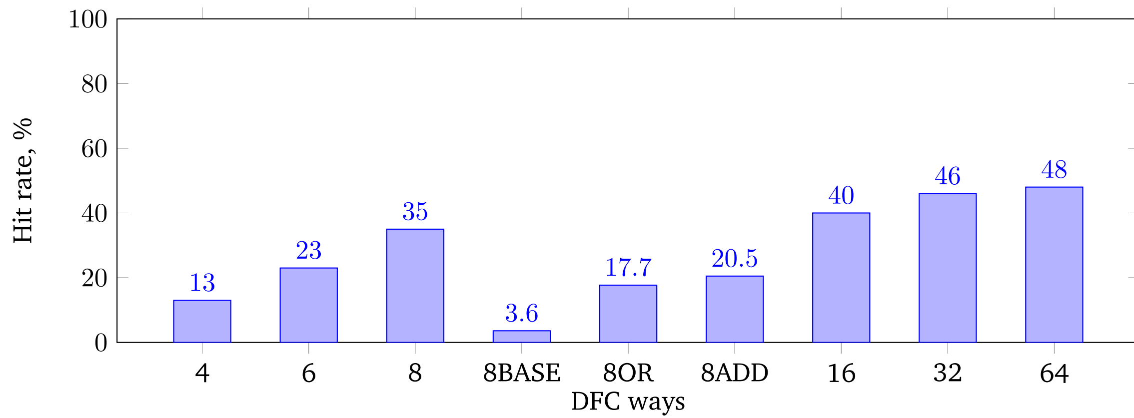


Figure 3: DFC hit rate on Dhrystone benchmark

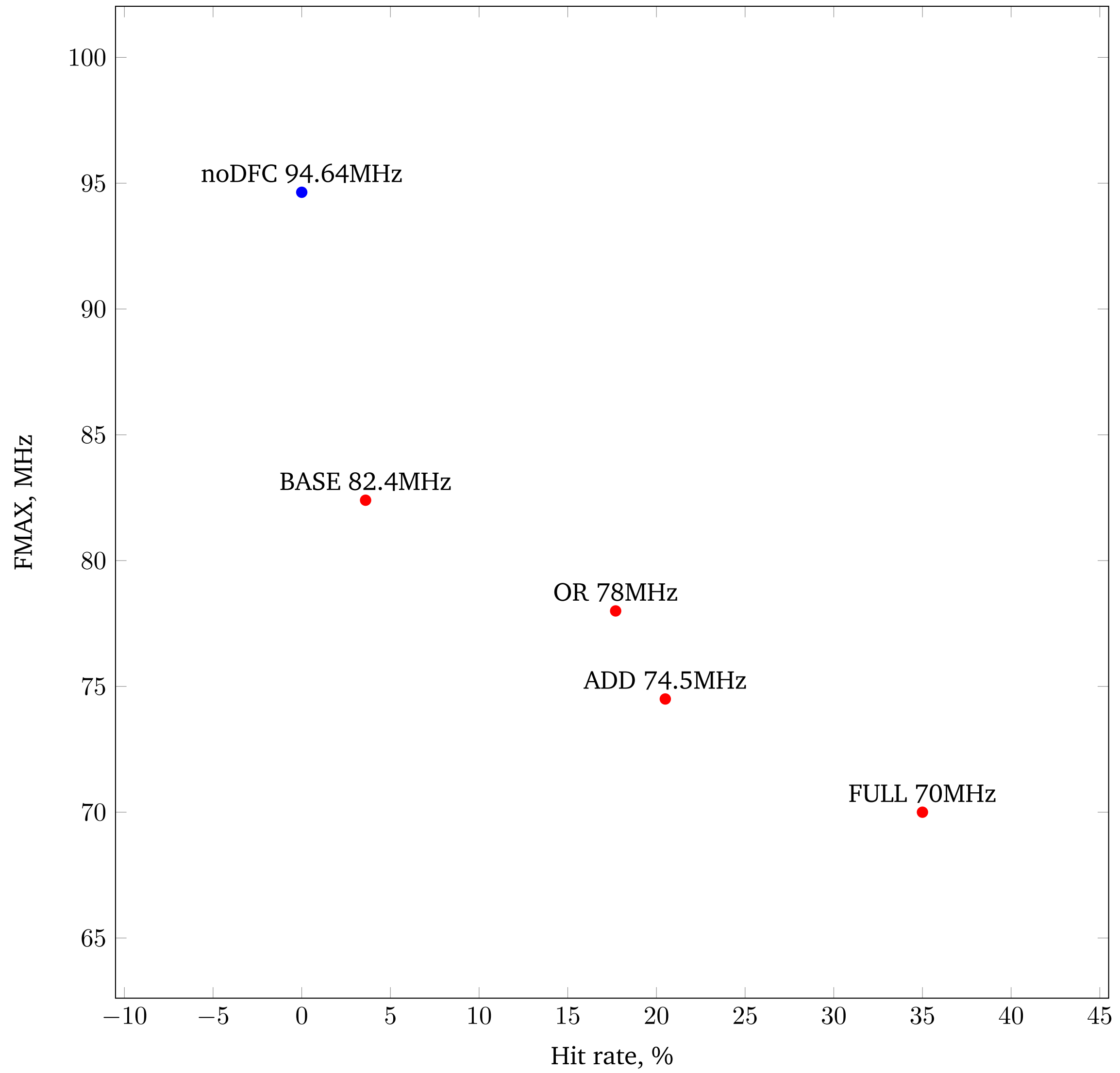


Figure 4: Frequency vs DFC hit rate on Dhrystone benchmark

5 Discussion

The required modifications were well localized within VexRiscv codebase. Only two files from the processor core were edited to provide a working implementation, hence the VexRiscv fulfills its goal of extensibility quite well. The only cross-cutting issue, and also the one we were unable to solve, was enabling result bypass. A possible reason for that is our use case us unlike what was previously implemented in VexRiscv: all of the original modules have their result available at a well defined stage, which can be predicted already at decoding. We are currently unsure how extensive an edit to fix that would be.

Implementing line filling would, however, require extensive modifications of the data cache module. Currently, the CPU-L1 interface only supports accessing single word by a virtual address. In order to allow power-efficient line filling, fetching data by specifying the way, set and line offset is necessary. Another complication in implementing line filling is the necessarily for arbitrating between L1 requests issued by the program and the DFC. A possible solution is to fetch the line lazily, that is preform no line filling, but access L1 by way, set and line instead of the virtual address on subsequent requests. That would still allow to take full advantage of DFC power efficiency, as TLB and L1 tag requests are fully eliminated. Also, unnecessary L1 data accesses would be reduced. Bypassing would however be hindered in some situations, as the results of such "direct" L1 request are still not available until the writeback stage, so more research is needed to evaluate this.

The hit rate measured is consistently lower than what was reported before Bardizbanyan et.al [5]. But our workloads are different: while the aforesaid work used a set of EEMBC benchmarks[14], designed to emulate real workloads, we used Dhrystone [13] benchmark, which is synthetic and therefore might not be fully representative.

FPGA are not a good platform to analyze power usage, as they are very power-inefficient themselves compared to hard silicone. Hence in practical terms a DFC implementation in an FPGA is quite useless in itself. Yet FPGA synthesis gives some insight into timings. In particular, we see that a fully-associative DFC carries a hefty timing penalty, with a roughly 20% frequency reduction. Hence any improvement in performance per megahertz that a DFC could offer is nullified. The promised 50% improvement of power dissipation will also be less significant in practice, as a processor running at lower frequency would spend more time awake, increasing static energy consumption. However, we believe that the deterioration of timing and area usage is an artifact of FPGA prototyping and might not apply in an IC layout. The fully-associative lookup can only be implemented in FPGAs using random logic, which is slow relative to memory blocks. The speed of random logic in ICs, on the other hand, is comparable to those of on-chip SRAM. Indeed, the timing evaluation by Bardizbanyan et.al. shows that the DFC lookup time is comparable to the time required to access a four-way L1.

6 Conclusion and future work

This thesis discussed implementation and evaluation of Data Filter Cache for the VexRiscv microarchitecture. Most importantly, it gave insight into extensibility of VexRiscv. The extension implemented clearly was not planned for in the VexRiscv framework, yet the modifications required were not very extensive.

Some important features, namely result bypassing and line filling, were left out for the purpose of the work. Implementing and testing these is left for future work. Another important direction for future work would be to test the idea of lazy line filling, proposed in this thesis.

Data Filter Cache is first and foremost an energy-saving enhancement. Energy considerations are less relevant for FPGA prototyping because of the overhead FPGA incurs. Synthesizing the implemented processor for a IC technology and evaluating its energy usage and frequency more precisely using a relevant implementation technology is beyond the scope of this work, but could be the next step.

Bibliography

[1] Själander, M., Martonosi, M., & Kaxiras, S. December 2014. *Power-Efficient Computer Architectures: Recent Advances*. Synthesis Lectures on Computer Architecture.

[2] Dally, W., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R., Parikh, V., Park, J., & Sheffield, D. 08 2008. Efficient embedded computing. *Computer*, 41, 27 – 32. doi:10.1109/MC.2008.224.

[3] Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B., Richardson, S., Kozyrakis, C., & Horowitz, M. 06 2010. Understanding sources of inefficiency in general-purpose chips. volume 38, 37–47. doi:10.1145/1815961.1815968.

[4] Bardizbanyan, A., Själander, M., Whalley, D., & Larsson-Edefors, P. 02 2013. Towards a performance- and energy-efficient data filter cache. 21–28. doi:10.1145/2443608.2443614.

[5] Bardizbanyan, A., Själander, M., Whalley, D., & Larsson-Edefors, P. 12 2013. Designing a practical data filter cache to improve both energy efficiency and performance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10. doi:10.1145/2555289.2555310.

[6] Dolu1990. 2020. URL: https://github.com/SpinalHDL/VexRiscv/.

[7] Asanović, K. & Patterson, D. A. Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[8] McKee, S. A. & Wisniewski, R. W. 2011. *Encyclopedia of Parallel Computing*. Springer US, Boston, MA. URL: https://doi.org/10.1007/978-0-387-09766-4_234, doi:10.1007/978-0-387-09766-4_234.

[9] Wulf, W. & McKee, S. 01 1996. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23.

[10] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., & Asanović, K. June 2012. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, 1212–1221. doi:10.1145/2228360.2228584.

[11] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., & Waterman, A. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[12] Plotkin, V. Rocket chip as a platform for evaluating novel microarchitectural improvements. Technical report, Dec 2019.

[13] Weicker, R. P. October 1984. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM*, 27(10), 1013–1030. URL: https://doi.org/10.1145/358274.358283, doi:10.1145/358274.358283.

[14] Embedded microprocessor benchmark consortium. URL: https://www.eembc.org/.

A List of acronyms

ISA Instruction Set Architecture

TLB Translation Lookaside Buffer

RISC Reduced Instruction Set Computer

PA Physical Address

VA Virtual Address

GPR General Purpose Register

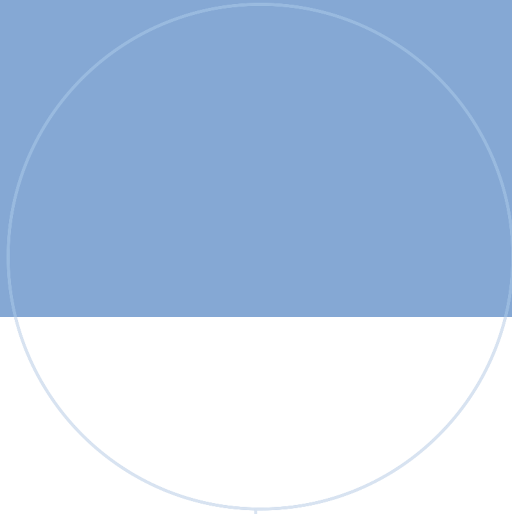
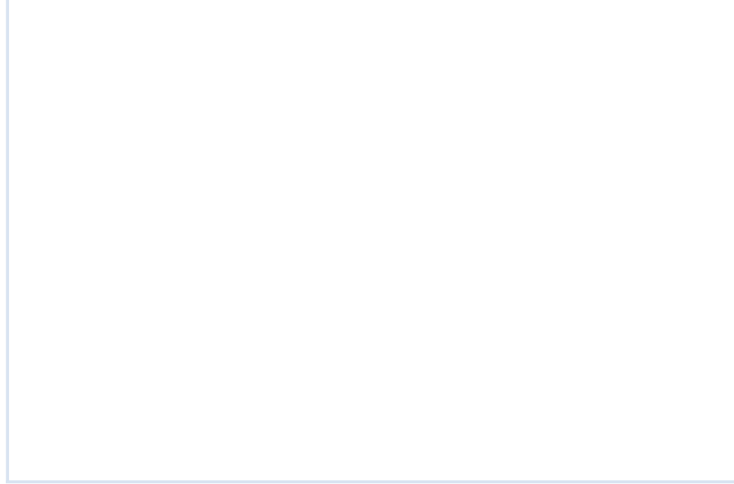
DFC Data Filter Cache

L1 First Level Cache

HDL Hardware Description Language

RTL Register Transfer Level

FPGA Field Programmable Gate Array

NTNU
Norwegian University of
Science and Technology