NTNU
Norwegian University of
Science and Technology

Combining the SHA and ELD^3 techniques to achieve energy-efficient data cache accesses

Salahuddin Asjad

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Magnus Själander, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

In recent years, CPU performance has become energy constrained. If performance is to continue increasing, new methods for creating more energy efficient CPUs will have to be explored. Current computing systems use complex CPUs that interface to the main memory through a hierarchy of caches. These performance-centric designs use a lot of power and chip-area to minimize the gap between CPU and main memory speeds. Caches contribute much of a systems's energy consumption. Conventional set-associative level-one data caches (L1 DCs) are performance-critical and are therefore optimized for speed. The access latency is optimized by accessing all ways in parallel for load operations. However, this results in a significant amount of wasted energy, since only data from one way is used. To reduce energy, numerous cache architectures, such as way-prediction, way-shutdown and highly-associative have been proposed. However, these optimizations in many cases increase latency and complexity, which makes them unattractive for L1 caches.

This thesis cover the implementation and evaluation of a combination of techniques that enables access to only the way where the data resides. The first technique is Speculative Halt-Tag Access (SHA) and works by halting cache ways that cannot possibly contain the requested data. The technique, Early Load Data Dependence Detection (ELD^3) , has no performance penalty and adds very little complexity to a conventional CPU core design. The second technique works by sequentially accessing tag and data ways when there is no data dependency with a subsequent instruction. These techniques have been implemented both independently and in combination. The SHMAC framework is used to evaluate the implementation by running a subset of MiBench benchmarks. The results show that on average, 40% of energy dissipation is reduced for loads with small displacement when using the SHA technique. By performing loads sequentially, the ELD^3 technique is able to reduce the overall enregy dissipation by 27%. When both techniques are combined, the ELD^3 technique is able to reduce the energy dissipation when the displacement is too large for the SHA technique. This combination of techniques gives an overall energy reduction of 43%.

Sammendrag

De siste årene har prosessorytelsen blitt mer og mer energibegrenset. Dersom ytelsen fortsetter å øke i tiden som kommer, må nye metoder til for å utvikle energieffektive prosessorsystemer. Dagens datamaskiner omfattes av avanserte prosessorer som kommuniserer med hovedminnet gjennom en rekke hurtigminner. Slike ytelsesfokuserte systemer bruker mye strøm og chip-plass for å redusere ytelsen mellom prosessor og hovedminnet. Energiforbuket fra hurtigminner står for betydelig mengde av det totale energiforbuket i et system. Vanlige set-associative L1 hurtigminner er ytelseskritiske og er dermed kun optimalisert for å oppnå best mulig ytelse. Dette blir blant annet gjort ved å aksessere alle feltene parallelt når data skal leses fra hurtigminnet. Dette medfører til en betydelig menge bortkastet energi, fordi den etterspurte dataen bare kan eksistere i en av feltene. Det er blitt forslått flere ulike teknikker for å løse dette problemet, blant annet ved help av way-predction, way-shutdown og highly-associative. Det som er felles for disse teknikkene, er at de øker aksesstiden og kompleksistenen i hurtigminner, noe som gjør disse teknikkene irrelevant for L1 cache.

I denne oppgaven vil to ulike teknikker bli implementert sammen, for å gjøre det mulig å bare aksessere feltene hvor den etterspurte dataen kan eksistere. Den første teknikken går ut på å filtrere bort feltene som ikke kan inneholde den etterspurte dataen i hurtigminnet. Den andre teknikken går ut på aksessere tag-feltene og data-feltene sekvensielt, over to klokkesykluser, dersom det ikke er dataavhengighet med en kommende instruksjon. Disse teknikkene har blitt implementert i et rammeverk kalt SHMAC. I tillegg har en rekke MiBench benchmark programmer blitt kjørt for å evaluere implementasjonen.

Preface

This report is submitted to the Norwegian University of Science and Technology in partial fulfilment of the requirements for an MSc degree in computer science.

This work has been performed at the Department of Computer and Information Science, NTNU, with Magnus Själander as the supervisor.

Table of Contents

List of Tables

List of Figures

Abbreviations

AMAT	=	Average Memory Access Time
APB	=	Advanced Peripheral Bus
ASIC	=	Application Specific Integrated Circuit
CPU	=	Central Processing Unit
CPI	=	Cycles Per Instruction
DC	=	Data Cache
DDB	=	Data Dependency Bit
DRAM	=	Dynamic Random Access Memory
DSL	=	Domain-Specific Language
DTLB	=	Data Translation Lookaside Buffer
EECS	=	Energy Efficient Computing Systems
ELD^3	=	Early Load Data Dependence Detection
FIFO	=	First-In, First-Out
FPGA	=	Field-Programmable Gate Array
GCC	=	GNU Compiler Collection
HCL	=	Hardware Construction Language
HDL	=	Hardware Description Language
ILP	=	Instruction Level Parallelism
IPC	=	Instructions Per Cycle
IoT	=	Internet of Things
ISA	=	Instruction Set Architecture
LRU	=	Least Recently Used
PC	=	Program Counter
PTC	=	Partial Tag Comparison
RAM	=	Random Access Memory
RISC	=	Reduced Instruction Set Computing
SHA	=	Speculative Halt-Tag Access
SHMAC	=	Single-ISA Heterogeneous MAny-core Computer
SRAM	=	Synchronous Random Access Memory
STA	=	Speculative Tag Access
VHDL	=	VHSIC Hardware Description Language
VHSIC	=	Very High Speed Integrated Circuit
WHC	=	Way-Halting Cache
XOR	=	Exclusive OR

Chapter 1

Introduction

A few decades ago, computing was only done by large mainframes, used for banking transactions, airline reservations, enterprise resource planning and in the industry. In the early 1990s, personal computers (PC) made their way into the homes of regular people. Since then, there has been huge development in the computing front. In the later years, the embedded area was introduced with small computers, such as microcontrollers. Today, embedded systems are in house appliances, wearables, electrical devices and industrial devices, and many of these devices such as smartphones carry more computing power than what was available in the early computers. The fact that computers have shrunk in size has led to an increased use of embedded systems in a wider range of applications. Today many of these systems are battery operated and often in places where recharging or replacing the batteries is not possible. With the increased growth of the Internet of Things (IoT), the need for energy efficient computing systems is more important than ever. This is why the researchers and semiconductor industry are using a significant amount of resources in increasing energy efficiency by developing embedded systems such that it consumes as little power as possible.

For several decades, increased computing performance could be achieved by aggressively exploiting more and more instruction level parallelism (ILP). The rate of advancements was quick, due to rapid improvements in production technology. Moore's law and Dennard scaling worked together to increase the performance of CPUs while still maintaining a reasonable power consumption.

Dennard scaling was based on the observation that the power consumption of a CPU could be kept constant when adding more transistors, as long as the transistor's threshold voltage was reduced accordingly (Dennard et al., 1974). Unfortunately, sub-threshold leakage increases exponentially when the threshold voltage is reduced (Taur and Nowak, 1997). A reason for this is the short-channel effects, which became significant when the feature size of semiconductors went below 100 nm (Oh et al., 2000). This has led to the end of Dennard scaling (Esmaeilzadeh et al., 2011), but Moore's law has kept on going. This phenomenon

has caused the size of transistors to scale down faster than their power consumption. Thus, it is only possible to have a subset of the transistors active at any given time if the chip is to stay within its power budget. Having transistors that cannot be used simultaneously with other transistors, is known as the dark silicon effect (Esmaeilzadeh et al., 2011).

A heterogeneous computing architecture is one way of solving this challenge (Borkar and Chien, 2011). In such an architecture, different types of cores are used for different applications where they are best suited for maximum energy efficiency. The less energy efficient cores are switched off as long as that application is running. Accelerators can also be added to a heterogeneous computer architecture. Accelerators are specialized units optimized for performing a specific application very energy efficiently. If the application is not running, the accelerator is switched off.

Even with a heterogeneous computing architecture, it is still important that each core is as energy-efficient as possible (Själander et al., 2014). The usual techniques also apply to the cores in a heterogeneous computing architecture. Caches for instance, are commonly used to improve the speed and energy-efficiency of memory accesses. The fact that the memory subsystem has become one of the largest contributor to overall energy consumption, beside the processor, makes the implementation of caches even more important (Nevine AbouGhazaleh and Melhem, 2005).

Efficient level-one data cache (L1 DC) accesses are important as they occur frequently and are performance critical. L1 DC caches are usually set-associative, which means that a given line in memory only can be stored in a specific set in the L1 DC consisting of multiple ways. This reduces the access time for the requested data compared to using fully-associative cache where the data can be stored in any cache block inside L1 DC. Conventional level-one caches perform accesses to both the tag and data ways in parallel to avoid reducing the system performance. However, parallel access waste energy as all ways in the cache are accessed in parallel even though the data can reside in at most one of them. Several different access techniques have been proposed in order to reduce the overall energy usage by limiting the number of ways that are accessed. However, the L1 DC is time-critical and by increasing the critical path, the CPU clock rate will decrease and may increase the energy disspiation.

This project is exploring the energy efficiency of the L1 DC, by implementing the speculative halt-tag access (SHA) and early-load data dependency detection (ELD^3) techniques in the SHMAC architecture. Both techniques are first implemented and evaluated individually, before combining both techniques. The first technique, the speculative halt-tag access (Moreau et al., 2016), can be performed when the displacement for address calculations is small. This approach is accessing the low-order tag bits for each way in the address generation stage that act as a filter for which tag and data ways of the set-associative L1 DC to access in the memory stage. Should the speculation fail, the cache is accessed conventionally during the memory stage with no impact on performance.

Figure 1.1 shows a high-level overview of a three-stage pipeline with the SHA technique. In the execute stage the base address is read from the register file and used to access the halt tag bits. A comparison of the base address and the relative address is done to check if the speculation succeeds. If the speculation succeeds, the hit vector from the halt tag

comparision is used to access the tag and data ways in the memory stage. If the speculation fail, the bits in the hit vector is set to ones, indicating that a conventional data access needs to be performed in the memory stage where all tag and data way are accessed.

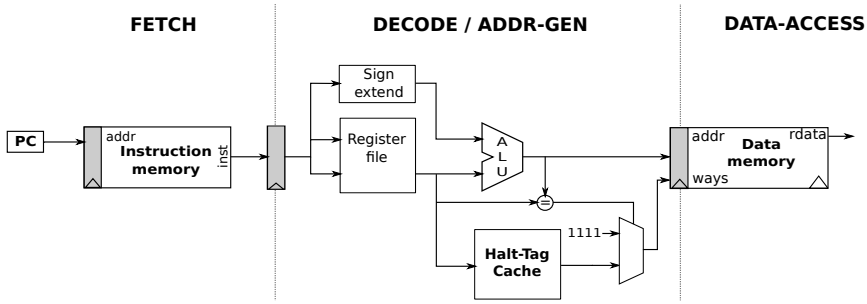

Figure 1.1: Three-stage pipeline with the SHA technique

The second technique is the early-load data dependency detection (ELD^3), that can detect if the load operation has a data dependency with a following instruction that will cause pipeline stall. If there is a data dependency, the tags and data ways are accessed in parallel to avoid stall cycles. If there is no data dependency between the load instruction and the following instructions, the tag and data ways for the load operation are performed sequentially, where all tags are accessed but only one data way in which the data resides is accessed in the next cycle. Figure 1.2 shows a high-level overview of a four-stage pipeline with the ELD^3 technique. In the execute stage, a data dependency bit (DDB) memory, consisting of dependency information for all load instructions in L1 IC is accessed. The dependency bit from the DDB memory is used to decide if the load operation will be perfomed sequentially with tag way access first, and data way access in the next cycle, or if the load operation will be performed parallelly with both tag and data ways accessed in the same cycle. As the instruction reaches writeback stage, a dependency information check is performed in order to keep the DDB memory updated.

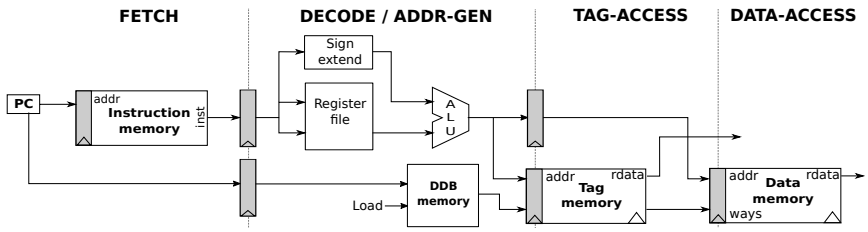

Figure 1.2: Four-stage pipeline with the ELD technique

The remainder of this document is organized as follows: Section 2 gives an overview of memories and their challenges, the SHMAC framwork and the RISC-V ISA. Section 3 describes the implementation of SHA and ELD^3 techniques and both combined. Section 4 describes the evaluation methods used in this project, followed by an evaluation with respect to performance and energy savings in Section 5. Section 6 reviews related work and Section 7 concludes this document with suggestions for future work.

Chapter 2

Background

In this section, the relevant background theory needed to understand the work in this project is presented. It gives the necessary knowledge to understand caches and the SHMAC architecture.

2.1 The Memory Wall

The performance increase of computational logic has outpaced the performance increase of memories for many years. Most of the reasons for this disparity are because of decisions made in the early 1980s. One of the main reason was the division from the semiconductor industry into microprocessor and memory fields. This led to the fields heading into different directions. While speed and performance was the main focus for CPUs, the capacity was the focus for memories (Carvalho, 2002). This resulted into a gap, as shown in Figure 2.1, where the improvement rate for microprocessor performance was 60% per year, while the access time to DRAM improved less than 10% per year (Carvalho, 2002). This observation is known as the memory wall, and can be traced back to 1994 when Dr. Wulf and Dr. McKee published a short paper about hitting the memory wall (Wulf and McKee, 1994). The Equation 2.1 which calculates the total average access time for a memory request, was used in the short paper to describe the reason for memory wall. The access time was estimated by the cache performance (T_{cache}), the probability of a cache hit (P_{cache_hit}) and the performance of main memory (T_{mem}). In the paper, it was assumed that the cache speed matched the speed of the CPU. The central argument of the short paper was that diverging exponential increase in performance of CPUs and memory, would drive drive the system performance to be completely memory-dominant.

$$T_{average} = T_{cache} \times P_{cache\_hit} + T_{mem} \times (1 - P_{cache\_hit}) \tag{2.1}$$

In the last years, it turned out to be difficult for DRAM technology to follow the pace of the processor scaling. As a consequence, the access time for main memory eventually became dominant, thus the bottleneck in computer performance. The short paper only considered challenges with access latency. In addition to the access latency, DRAM today also faces challenges from the increased bandwidth and high energy consumption. Today memory wall can be seen as latency, power, scalability, and bandwidth (Zhang, 2014).

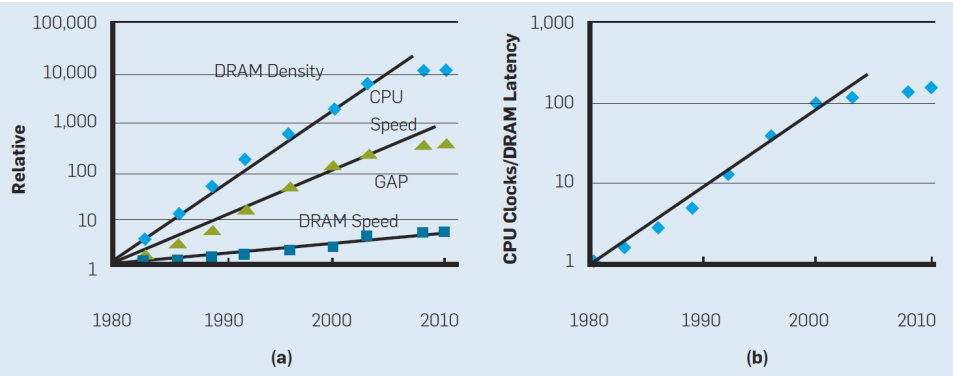

Figure 2.1: The increasing difference between CPU speed and memory speed is known as the memory wall (Borkar and Chien, 2011)

2.1.1 Storage technologies

Random-access memory (RAM) comes in two types: Static RAM (SRAM) and Dynamic RAM (DRAM). SRAM stores each bit in a memory cell, where each cell is implemented with a six-transistor circuit. An important attribute with the memory cell is that it can stay indefinitely in either of two states. Due to this bistable attribute of the memory cell, it will retain its value as long as it is kept powered. DRAM memories in the other hand is very sensitive to any disturbance, and needs to be refreshed periodically to prevent data loss. This leads to a significant decrease in performance as shown in table 2.1, because the memory is not available for a normal read or write operation while the memory refresh is performed. Because DRAM cells use fewer transistors than SRAM cells, DRAMs have higher density, thus usually used for main memory and lower level caches where the capacity is important.

	Transistors per bit	**Relative access time**	**Persistency**	**Relative cost**
SRAM	6	1x	Yes	100x
DRAM	1	10x	No	1x

Table 2.1: Characteristics of SRAM and DRAM memories

Figure 2.2 shows the trade-off between cache sizes and power dissipation for different transistor budgets. The insight here is that there is a trade-off between the amount of

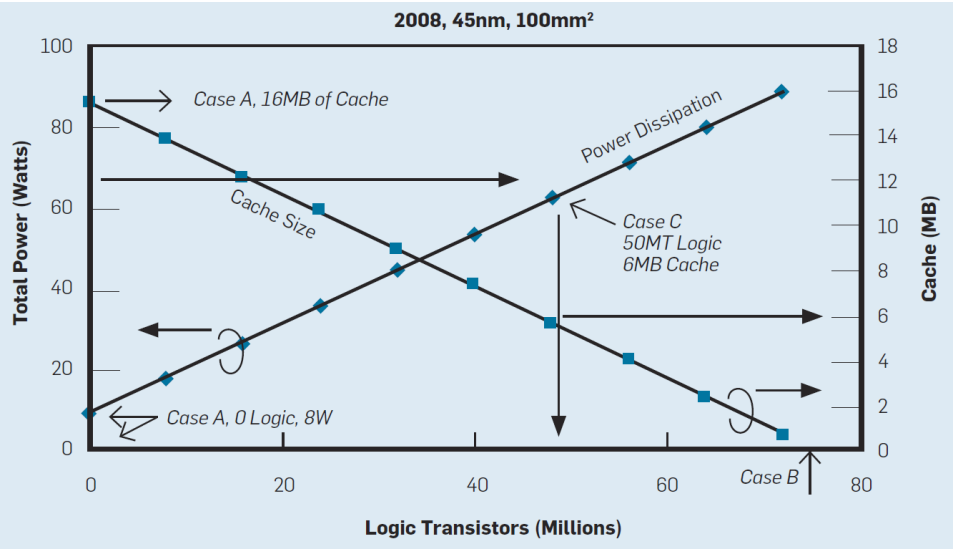

Figure 2.2: Transistor integration capacity at a fixed power envelope (Borkar and Chien, 2011)

computational logic and cache that can be integrated within a fixed power envelope. As computational logic is more power intensive than cache, integrating a larger cache can be used to keep the power dissipation low.

2.1.2 Memory system hiearchy

The performance of the CPU-memory interface is usually measured by two parameters: the latency and the bandwidth. The latency is the time between the initial memory request by the processor and the memory response, while the bandwidth the rate at which the data transferring can be performed to or from the memory. In a best-case scenario there would be zero latency and unlimited amount of bandwidth. Unfortunately, the memory gap results in high latencies between the processor and memory. Memory hiearchy provide one way of decreasing the memory latencies and reducing the bandwidth requirements. A memory hiearchy is a hiearchy of storage devices with different capabilities, costs and access rates. There are well known tradeoffs on memory hiearchies between cost, system performance and memory size. To provide the CPU with necessary data as quick as possible, the frequently used data is stored in the caches that are placed closer to the CPU. These are small, but fast caches consisting of a subset of data stored in relativley slower memory levels. Figure 2.3 shows a typical memory hiearchy structure existing in today's computers. L1 cache is usually placed on-chip and close to the CPU in order to expoloit locality by keeping data likely to be used again as close as possible. If there is a cache miss in L1 cache, the search request will begin for L2 cache, which is often larger than L1, thus results in higher latency. With each cache miss, the search proceeds to the next level memory until the requested data is found.

Early memory architectures used unified caches, known as Princeton architecture, where both instructions and data were stored in the same cache (Ghoshal, 2011). Instructions and data usually have different access patterns, and access different parts of memory. In addition, the instructions and data have to compete for cache lines, which results in increased cache trashing because instructions and data are conflicting. Therefore, in today's memory architectures the instructions and data are often stored into separate caches, where instruction cache only stores instructions and data cache only stores data.

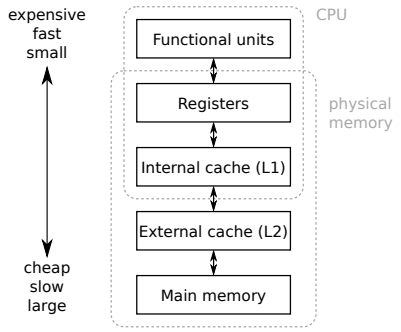

Figure 2.3: A hiearchy with on-chip and off-chip memories

Associativity

In order to reduce the search time for a data requests, the caches often have a restricted placement policy based on an entry's locality. Cache hit is then detected through an associative search of all tags, instead of searching through the entire cache. This is known as the cache associativity, and there are three main methods:

- **Direct-mapped cache:** Entries can go to exactly one place in the cache.
- **N-way set associative cache:** Entries can go to one of N places in the cache.
- **Fully associative cache:** Entries can go to any place in the cache.

A lower associativity leads to a less complicated and faster cache, because the data can only reside in less cache lines. However, it will increase the miss rate due to possible increase of cache conflicts. Conversely, a cache with higher associativity will have fewer conflict misses, but will also be slower and more complicated. This represents a trade-off, which CPU designs will have to take into account. For example, the L1 cache has to be fast, so it is advantageous to stick to a limited associativity. However, the L1 cache is also usually quite small, and a higher associativity can increase the effective use of the cache. In practice, most commercial CPUs have a 4-way or 8-way L1 cache. Increasing the associativity beyond this level shows less benefit. Figure 2.4 shows a four-way set associative virtually indexed, virtually tagged cache. The index bits from the memory

address are used to locate to the corresponding set in the cache. The tag bits from memory address are compared with tag arrays, while the offset bits from memory address is used to get the correct word the data array.

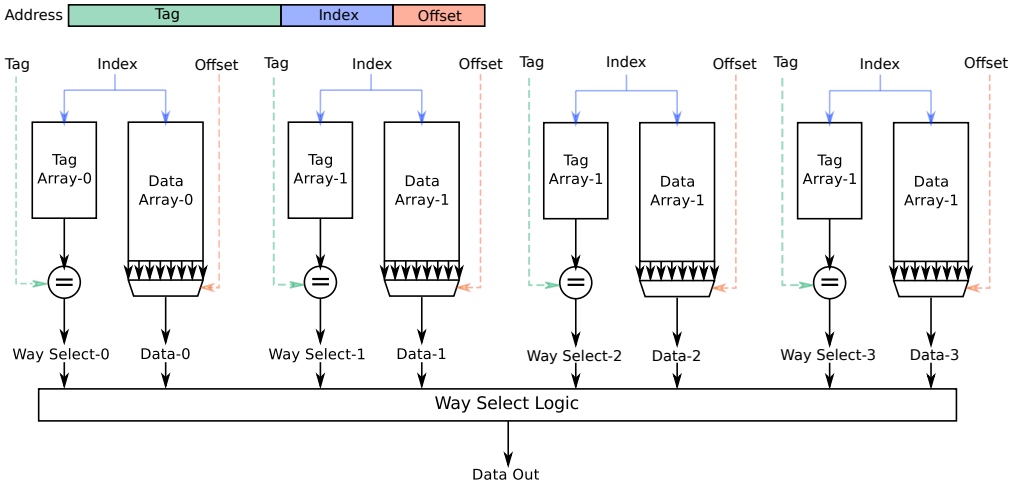

Figure 2.4: 4-way set associative cache with separate tag and data arrays

It is the division of caches into sets that enables the speculative halt-tag access(SHA) technique to selectively disable some of the ways without impacting the functionality of the cache. Other techniques have also exploited this, and some of them will be discussed in Section 6.

2.2 Dependencies in pipelined processors

- **Structural dependency:** When there is a resource conflict in the pipeline, where more than one instruction tries to access the same resource.
- **Control dependency:** When there is a control instruction such as branch, call, jump etc. The target address is not known before during execution.
- **Data dependency:** When an output of one instruction is required as an input for one of the next instructions in the pipeline.

There are several approaches to deal with these dependencies. Structural hazards are often common when unified caches are used, causing resource conflict when there is a instruction fetch and load or data operation in same cycle. Although separate instruction and data caches are used in this project, structural hazard is one of the drawbacks of the ELD[3] technique which will be discussed in Section 5.2. To avoid control dependencies, branch

predictors are often used to predict the outcome of a control instruction. Compile-time scheme such as loop-unrolling is also used to reduce loop branches. Data hazards can simply be avoided by forwarding data from the memory stage as long as the data is received from memory. Data forwarding is necessary for the ELD^3 technique to operate correctly as parallel accessed data needs to be forwarded from tag/data access stage in order to avoid a stall cycle. This will be discussed in Section 3.2.

2.3 The Single-ISA Heterogeneous MAny-core Computer (SHMAC)

In order to explore the challenges and possibilities of heterogeneous computing systems, the EECS group at NTNU has initiated the Single-ISA Heterogeneous MAny-core Computer (SHMAC) research project. SHMAC itself is an infrastructure created with the purpose of investigating heterogeneous computing systems at all abstraction levels. Within this framework, different heterogeneous processors can be instantiated from a collection of processing tiles. These processing tiles may be very different in their implementations, but the programming model is the same for all tiles. This common programming model is achieved by implementing the same memory model and ISA for all tiles.

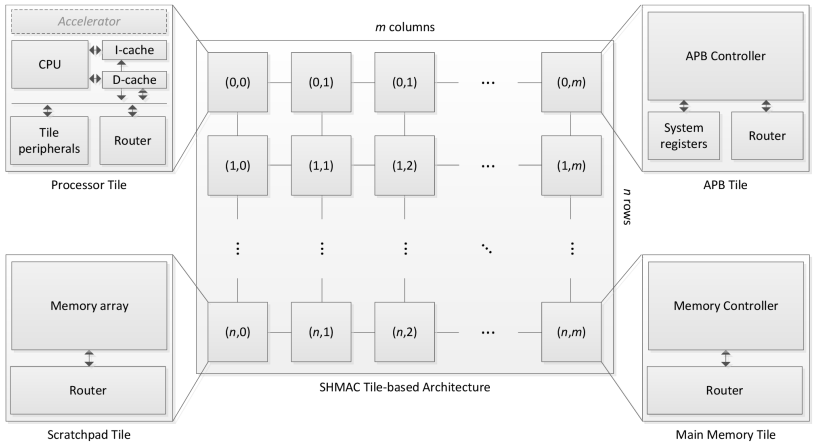

Figure 2.5: The high-level architecture of SHMAC

As stated above, SHMAC is a tile-based architecture. This means that the tiles are placed in a grid, with connections to their nearest neighbors. These connections are ordered in a mesh, and as such the connections go to neighboring tiles north, south, east and west of the tile, but not diagonally. Additionally, tiles at the edge of the grid does not have any outward-facing connections. The grid has to be rectangular, but not necessarily square. An illustration of this can be seen in Figure 2.5.

The heterogeneous aspect of the SHMAC architecture is enabled by having different types of tiles. These tiles have different functions, but they all have to implement the SHMAC router interface as a minimum requirement. SHMAC currently has these types of tiles:

- **Processor Tile:** Contains a processor, caches and peripherals, and can optionally also contain accelerators.
- **Scratchpad Tile:** Contains a memory, but no processor. This is a software-managed on-chip memory, with an address space of 128 MB.
- **Main Memory Tile:** Contains a memory controller used for giving SHMAC access to off-chip memory.
- **APB Interface Tile:** Contains an Advanced Peripherals Bus (APB) slave. This is used for communication between SHMAC and the host machine, and gives the host machine access to SHMAC's memory space.
- **Dummy Tile:** Contains only the router, and does not do anything else. Because this tile uses few resources, it is useful for making the grid rectangular on a resource-constrained FPGA.

2.3.1 The SHMAC Processor Tile

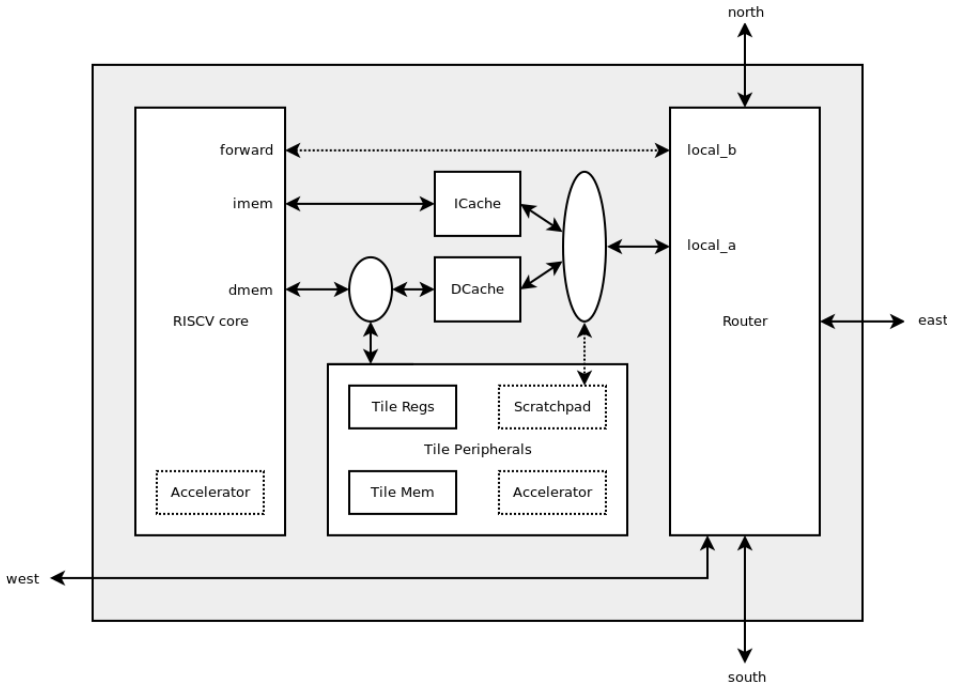

Figure 2.6: Overview of the SHMAC processor tile

In addition to the router, the SHMAC processor tile contains a CPU core, instruction and data caches, and tile peripherals. The tile peripherals include tile registers and tile memory, and can optionally contain a scratchpad and an accelerator. The CPU core itself may also implement an accelerator. This is illustrated in Figure 2.6.

The CPU core implemented in the SHMAC processor tile is a modified version of the 3-stage pipelined version of the RISC-V Sodor CPU, also known as the Z-Scale (Celio and Love, 2014). In addition to the 3-stage pipeline, there are also a number of other RISC-V integer pipelines like 1-stage, 2-stage and 5-stage pipeline processors. The 3-stage Z-Scale pipeline differs from the others in the sense that it uses sequential memory. Figure 2.7 shows the datapath for the Z-scale processor. The frontend consists of instruction fetch stage, while the backend consists of execute and memory/writeback stage. This processor is implemented in Chisel, and supports the RV32IS instruction set architecture (ISA), which is 32-bit integer RISC-V ISA with supervisor mode implemented, such that the RISC-V proxy kernel (riscv-pk) can be executed. The modifications done to the CPU core by the SHMAC project include, but is not limited to, support for external interrupts, support for variable latency memory with back pressure, support for multiplier and division instructions, support for load-reserved/store-conditional instructions and support for the SHMAC specific store-forward instruction.

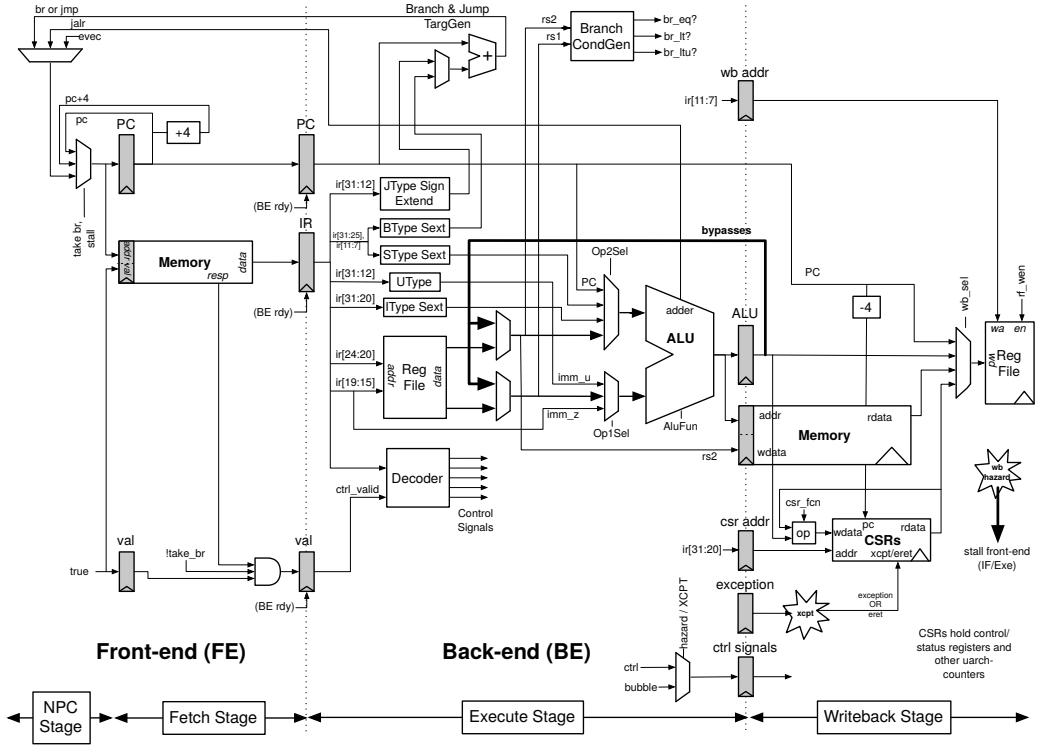

Figure 2.7: Overview of the 3-stage Z-scale core

2.3.2 Mesh configuration

In this project, heterogeneity and multicores have not been an area of focus. Therefore, a minimal instantiation of the SHMAC architecture has been employed. This SHMAC processor consists of a processor tile, a main memory tile, an APB interface tile, and a dummy tile to fill the rectangle, thereby creating a 2×2 grid. Figure 2.8 shows a high-level overview. A more detailed overview will be provided in Section 3.

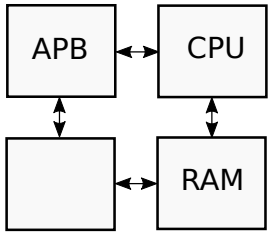

Figure 2.8: High-level architecture of the SHMAC instance used in this project

2.3.3 The RISC-V ISA

Previous iterations of the SHMAC project were based on the ARMv4t (ISA) (Lefsaker, 2014). This ISA is proprietary, which is problematic for a scientific research project. In order to facilitate the heterogeneous computing in SHMAC, it would be beneficial to use an ISA that is easily extensible. It was therefore decided to abandon ARMv4t in favor of RISC-V.

The RISC-V ISA is an open instruction set architecture that is freely available to academia and industry (Waterman et al., 2016). It was originally designed to support computer architecture research and education.

RISC-V has several design features that makes it well suited for use in the SHMAC project, in addition to being open. First, it has a small base integer ISA, that is entirely usable by itself. Second, RISC-V is not specifically designed for a particular microarchitecture or implementation technology. Third, it can be specialized and extended to a large degree. These features make RISC-V a good fit for the heterogeneous computing of SHMAC.

2.3.4 Addressing mode

There are four core instruction formats (R/I/S/U) in the base RISC-V ISA, where all instruction formats are fixed 32 bits in length and must be aligned in a four-byte boundary in memory. Figure 2.9 shows the four core instruction formats and the semantics of the positions of the instruction formats. Instructions such as ADD, SUB and AND and other register-register operations, are encoded in the R-type instruction format whereas register-immediate operations such as ADDI, SLTI and load instructions are encoded in I-type

instruction format. Store operations are encoded in S-type. Like in many other ISAs, the RISC-V ISA keeps the same position for source (rs1 and rs2) and destination (rd) registers in all instruction formats at the expense of having to move immediate bits across formats. Bearing in mind that decoding register specifiers often is on the critical paths in the implementations, keeping the register specifiers at the same position makes the instruction decoding simpler and faster.

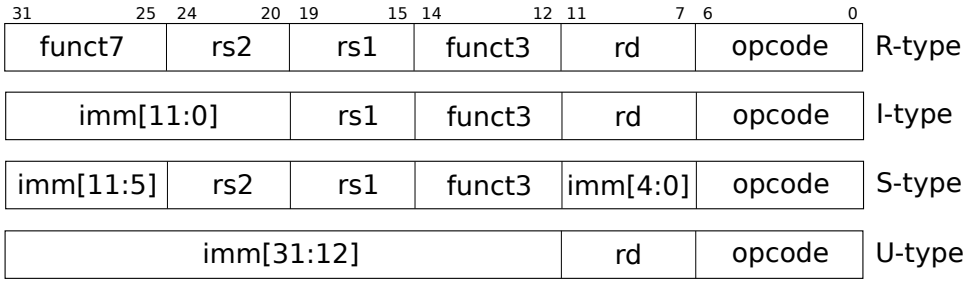

Figure 2.9: RISC-V base instruction formats.

2.3.5 Chisel

Hardware description languages (HDLs) are languages used for developing hardware. Among the most commonly used are VHDL and Verilog. In this project, Chisel is used.

Chisel is a hadware construction language (HCL) developed at UC Berkeley (Bachrach et al., 2012). Many HDL languages such as Verilog were designed first to be simulation languages. Chisel is designed specifically for constructing actual hardware (Celio and Love, 2014). It is embedded in the Scala programming language, and supports advanced hardware design. Chisel aims to reduce the hardware development time by being a more flexible alternative to VHDL and Verilog. Because Chisel is embedded in Scala, it provides a high abstraction level for the hardware design, and supports many advanced programming concepts, such as object orientation, functional programming, parameterized types, and type inference. When compiled, Chisel can generate a C++-based software simulator or Verilog code for synthesis on an FPGA or an ASIC.

Chapter 3

Implementation

The aim of this project was to implement the speculative halt-tag access (SHA) and early load data dependency detection (ELD^3) approach into the SHMAC architecture to evaluate their effectiveness and potential to reduce the energy dissipation. Section 3.1 covers the implementation of SHA. Section 3.2 covers the implementation of Early Load Data Dependence Detection (ELD^3), followed by Section 3.3 which covers the combined implementation of SHA and ELD^3.

3.1 Speculative Halt-Tag Access (SHA)

The speculative halt-tag access (SHA) technique is an approach to determine which tag and data ways need to be accessed during the beginning of the memory stage. The idea is to access the low-order bits of the tag in the address generation stage together with the calculation of the relative address. It is then possible to exclude accesses to cache ways in the memory stage that cannot possibly contain the requested data. This technique is based on the observation that the address displacement is often small, and often only change the offset of the relative memory address. This makes it possible to use the index of the base address to get the halt tags in parallel with the relative address calculation that is used to access the particular word in a cache line.

Figure 3.1 shows how a check is done to know if the address displacement is too large for the SHA technique. We simply add the base address and displacement together, and then compare the high-order bits of the result of the addition with the high-order bits of the base address. If the address displacement is too large, the line index and tag are likely to change during the relative address calculation, thus in these cases the halt tag should not be accessed. When there is a deeper pipeline with a separate decode stage, the sign and

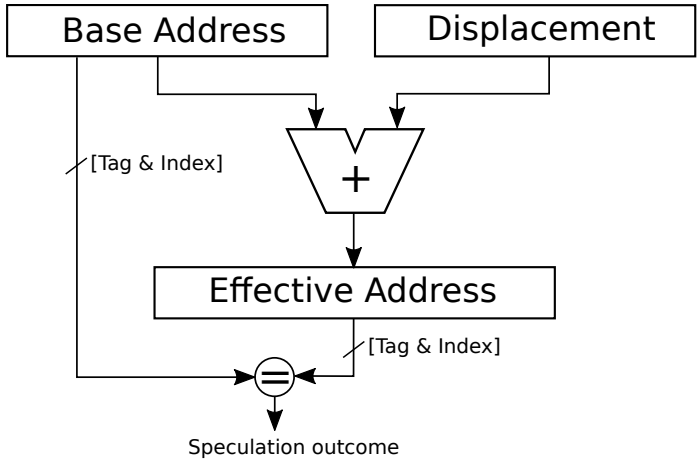

Figure 3.1: Halt-tag address calculation

width of the displacement can be checked earlier by performing an AND and OR on the higher bits.

Initially, the halt-tag cache is getting the base address from the datapath. Since the base address can be a memory request to a location that may not be in the cache region, we first need to check if the base address is pointing to a location within the cacheable region. This is simply done by checking the upper level bits of the base address to determine if the address is within the range. If the base address is not within the region, the halt-tag request is disregarded, and we know that the memory request in the next clock cycle will need to get the data directly from main memory. If the base address is within the cache region, the index of the base address is first used to access the halt tags from all ways. We also check if there is a write request to the same index position we are looking to get the halt tags from. If that is the case, we need to forward the halt tags from the write port so we get the correct halt tags. Another approach to get the correct halt tags would be to stall the pipeline to let the halt tags be written to the cache before performing a halt-tag access to read the correct halt tags from all ways. This would make the halt-tag cache simpler, because only a single port would be used to do the reading and writing of the halt-tag cache. However, it would not only decrease the overall performance due to the additional pipeline stalling, but it might also increase the energy dissipation. When the halt tags have been accessed from all ways, the halt-tag comparison is performed, where the halt tag from the base address is compared with the stored halt tag of each way. If there is a halt-tag match between the halt tag from the base address and the halt tag from a way, the bit corresponding to the way in the response vector is set to 1 to indicate that there is a halt-tag match. When the halt tags of all ways are compared, the response vector is sent to the datapath to be used in the next clock cycle.

3.1.1 Implementation of the Halt-Tag Cache

Figure 3.2 shows what a halt-tag cache line looks like. In this case the halt-tag width is set to 5 bits. In Section 5 we show how the performance results vary when we change the halt-tag width that is stored in the halt-tag cache. We also show how much impact the stored valid bit has to speculation accuracy.

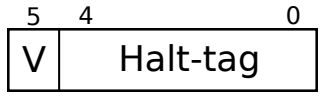

Figure 3.2: Halt-tag cache line using 5 bit wide halt tag

There are several possible ways to store the halt tags. However, we wanted to add a low amount of complexity to maintain the halt tags whenever the data cache was updated. Figure 3.3 shows how the halt-tag cache is implemented, together with the existing data cache. Memory requests from the datapath are connected to the storage module, which consists of the halt-tag cache and data cache. Instead of using a separate state machine for the halt-tag cache, the state machine for the data cache from the CoreFSM module is used to control the halt-tag cache. For example, if a cache line in the data cache is to be invalidated, the halt tags for the corresponding cache line will also be invalidated in the halt-tag cache by observing that the data cache is in the invalidate state. We also added performance counters to the CoreFSM module to get statistics during execution.

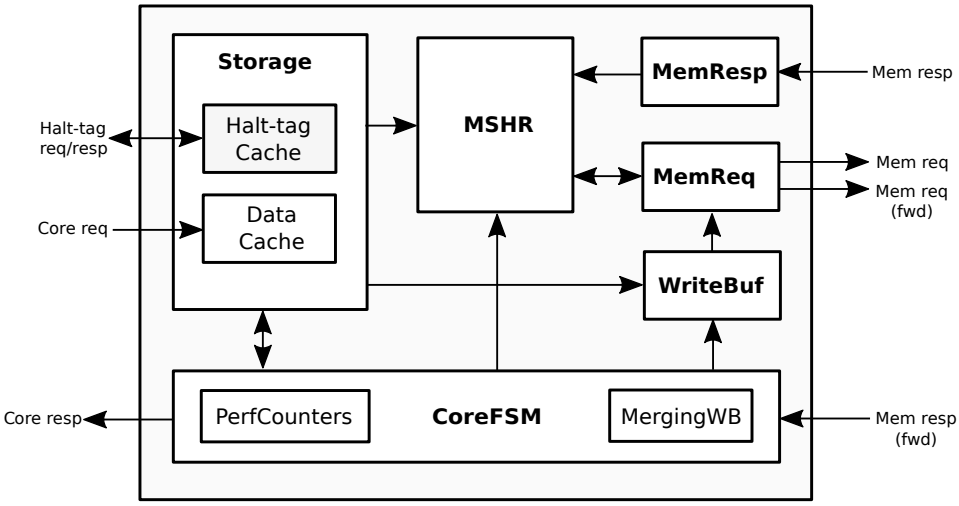

Figure 3.3: Block diagram of the data cache

We decided to implement a dual-port Block RAM with two independent ports that enable shared access to a single memory space where the halt tags are stored. Port names, includ-

ing the control signals for the Block RAM, are shown in Figure 3.4. Port A provides read and write access to the memory. This port is used when the data cache is modified and the halt-tag cache needs to be updated with a write operation. Port B only provides read access to the memory, which is used when there is a request from the processor.

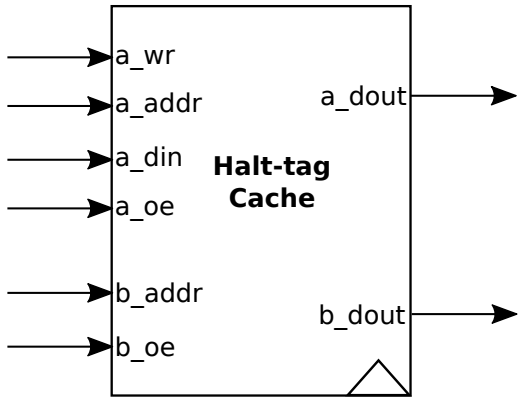

Figure 3.4: Dual-port Block RAM

As mentioned in Section 2.3.1, the implemented CPU-core is a modified 3-stage pipeline version of the RISC-V Sodor CPU, also called Z-Scale. Figure 3.5 shows the the datapath of the Z-Scale processor with SHA implemented. The execute stage is modified with new logic and functionality for SHA, and the resulting hit signals from the execute stage are routed to the enable-inputs of the data cache. The rest of the datapath remains unchanged from the original Z-Scale processor. The displacement checker decides whether or not a halt-tag access should be performed. If the displacement is too large, the vector '1111' (in the case of a 4-way set associative scheme), will propagate to the memory stage, which means that a conventional data access will be performed, where all ways are accessed. When the displacement is small, the control signal b_oe of the halt-tag cache will be set, meaning that it will perform a read access. The halt-tag cache will use the base address to look for halt-tag matches and output a vector with a length equal to the number of ways. The vector is then propagated to the memory stage to control which way to access.

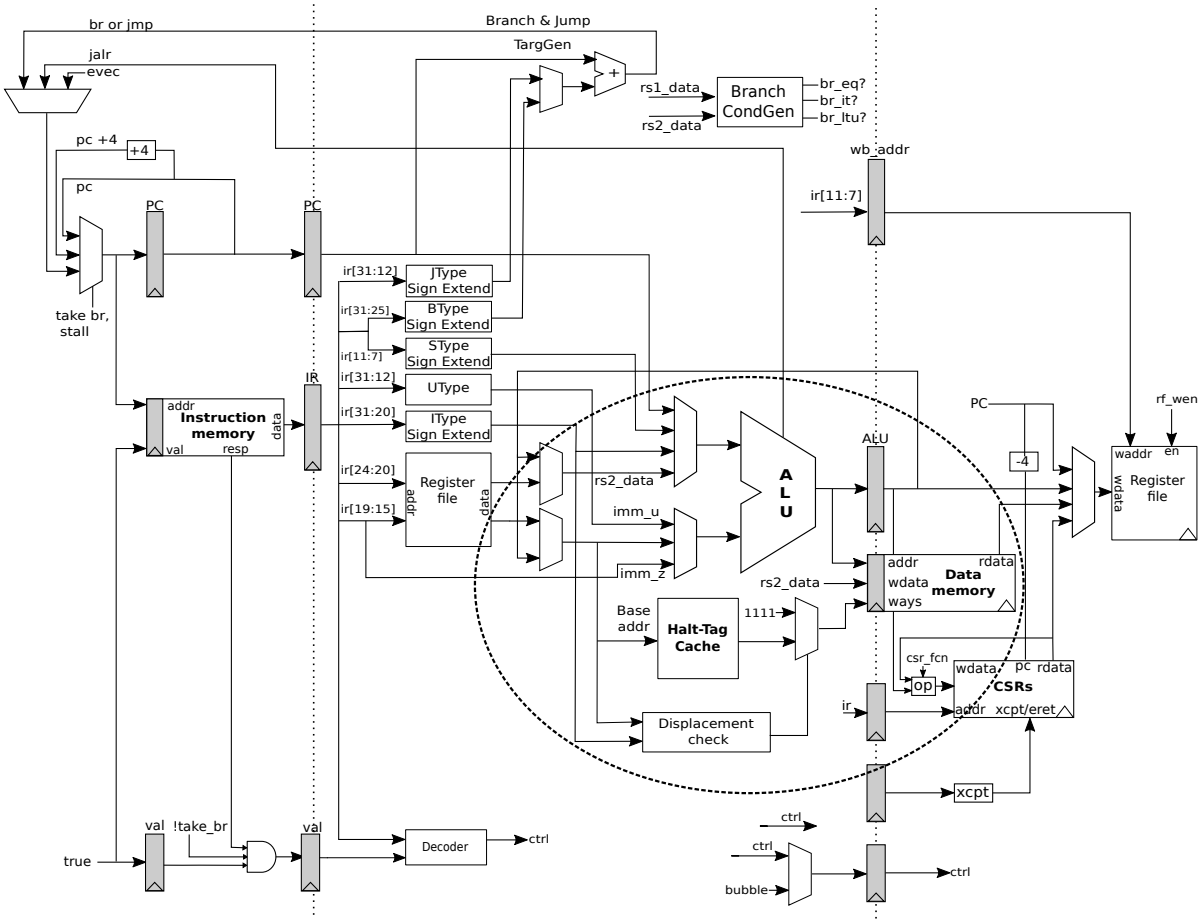

Figure 3.5: Z-Scale datapath with SHA logic implemented.

3.2 Early Load Data Dependence Detection (ELD^3)

The speculative halt-tag access (SHA) technique eliminates accesses to cache ways that cannot possibly contain the requested data when the address displacement is small. When the address displacement is too large for the SHA technique, all four ways are accessed in parallel during the load operation. To reduce the energy dissipation for load operations with too large displacement, another technique, early load data dependence detection (ELD^3), has been implemented into the SHMAC architecture, which will be presented in this section.

The early load data dependence detection (ELD^3) is an approach that can detect if the load operation has a data dependency with a following instruction that will cause pipeline stall. If there is a data dependency, the tag and data ways are accessed in parallel to avoid stall cycles. If there is no data dependency between the load instruction and the following instructions, the load operation is performed sequentially, where all tag ways are accessed but only one data way in which the data resides is accessed in the next cycle.

Figure 3.6 shows the three stages of the RISC-V Z-Scale processor. In the instruction fetch stage, the instruction cache is accessed to get the requested instruction word. During the next stage, the decoding and arithmetic operation are performed, which means that it is known before the beginning of the memory stage if it is a load instruction. The calculated output from the arhitmetic operation is used as memory address to access the data cache if it is a memory operation, before eventually writing back to the register file. When there is a load or store operation, both tag and data ways are accessed parallelly in the same cycle. Given that the data we are looking for during a load operation only can reside in one of the ways, a significant amount of energy is wasted due to the parallel access to avoid additional stalls.

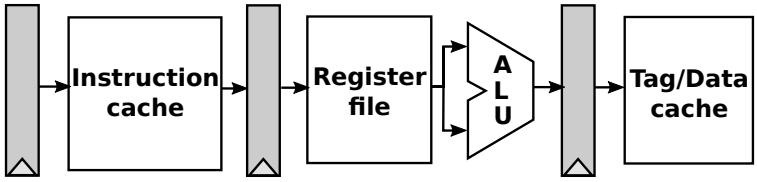

Figure 3.6: Three stages of RISC-V Z-Scale processor.

In order to access tag and data ways over two consecutive cycles, an additional pipeline must be implemented. Figure 3.7 shows the additional pipeline, such that the data accesses can be performed over two clock cycles. If the data access is perfomed parallelly due to data dependency, both tag and data ways are accessed in the first memory stage. If the data access is performed sequentially, the tag ways are accessed in the first memory stage, while a single data way is accessed in the second memory stage using the tag hit signal from the tag access stage. Store operations to data caches are usually performed sequentially. That is because the tag comparison must be performed and the tag hit signal must be known before the data can be written to the correct data way. By adding the additional pipeline

to enable sequential load accesses, the store operations can begin earlier by accessing the tags in the first memory stage instead of the second stage. The store operations still takes two cycles for most SRAMs that are synchronous, because the write to the selected data array can not be done before the tag hit signal is known, which means that the additional pipeline will not reduce energy dissipation for store operation.

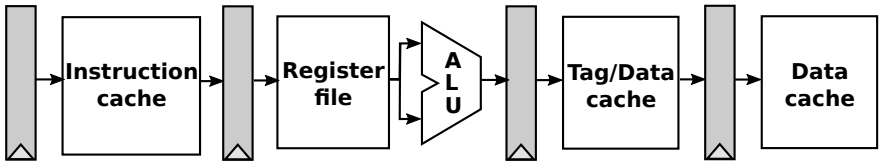

Figure 3.7: RISC-V Z-Scale pipeline with additional memory pipeline.

To implement the ELD^3 technique, the information about data dependency between the load instruction and the following instructions must be available at the time of the load operation. In a conventional in-order pipelined processor, the information must the available before the end of the address generation stage in order to decide if the tag and data ways will be accessed conventionally in parallel, or if an extra pipeline stage will be used to access the tag and data ways sequentially over two clock cycles when there is no data dependency. As mentioned in Section 2.3.3, the register specifiers always have the same position for all instruction formats in RISC-V ISA. This means that in an in-order pipeline, it is possible to compare the destination register of the load instruction with the source registers of the instructions that enter the pipeline after the load instruction to detect if there is a data dependency between the load instruction and the following instructions. However only using this method will not solve the data dependency detection required for the ELD^3 technique.

In a five stage in-order pipeline processor, usually there exist at least one instruction decode stage which is used for decoding and operations such as register file read. When a load instruction is in address generation stage, it is possibe to check for data dependency between the load instruction and the instruction that immediately follows it. However it is not directly possible to check for data dependency between the load instruction in address generation stage and the second and third upcoming instructions. In a three stage pipeline processor such as the RISC-V Z-Scale where the decoding is done together with the address generation stage, it gets even more difficult to directly check for data dependency between the load instruction in address generation stage and the upcoming instructions.

3.2.1 Implementation of data dependency bit (DDB) memory

In this project, a practical approach is used that provides early detection of a dependency on a load for most of the executed load instructions. For each instruction word in the L1 instruction cache, an extra bit is used to indicate if the instruction is a load instruction and if the instruction has a dependency with any of the following two instructions. When an instruction is decoded early in the address generation stage, a check is performed to to

identify if it is a load operation. If it is a load operation, a check is performed to identify if there is a dependency between the load operation and the two upcoming instructions. If a dependency is detected, the bit corresponding to the load operation is set to indicate that the load operation should be accessed in the same cycle to avoid additional stalls. If no dependency is detected, the bit corresponding to the load operation is cleared, such that the tag and data ways are accessed sequentially over two clock cycles the next time the load operation is executed. The dependency bit corresponding to the load operations will be helpful in the decision making regarding sequential or parallel data access as long as the cache line with the load instruction is not evicted from the L1 instruction cache. If a cache line is evicted from the L1 instruction cache, the bits corresponding to the load instructions might be incorrect. However, the load operation will still be executed correctly with the correct response from the data cache even when the dependency bit is incorrect. If the dependency bit for the load operation in the evicted cache line was cleared, the load operation in the new cache line will only cause an additional stall cycle if it has a dependency. The dependency bit for the new load operation is updated after it is executed the first time. This simplifies the implementation as an access for the entire cache line is not required to update the dependency bits for the instructions.

There are several possible ways to implement the accessing of dependency bits. One way is to access the dependency bits together with the words in the instruction cache. This method only adds one extra bit per word in the instruction cache, thus results in low overhead in terms of bits. Also, the dependency information is known in the instruction fetch stage, which gives the futher stages an additional cycle to prepare for the load operation even before entering the instruction decode stage. However, the dependency bits will need to be updated independently of the instruction word which complicates the design of the instruction cache. Furthermore when the bits are stored together with the words in instruction cache, the dependency bit for all instruction will need to be read, since the instruction type is not known in the instruction fetch stage.

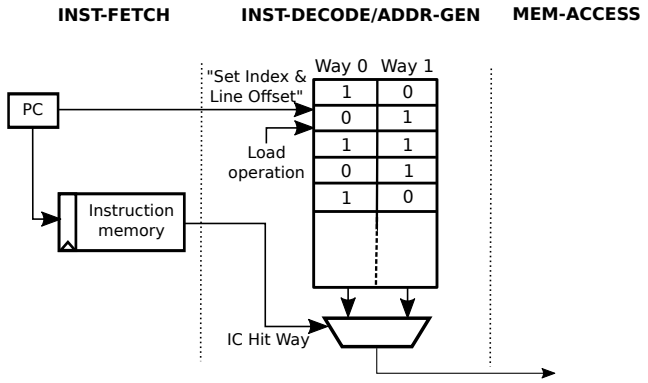

Figure 3.8: Data dependency bit (DDB) memory access

Instead, a more practical approach is to implement a data dependency bit (DDB) memory, where the bits are stored apart from the instruction cache. The advantage of separating the dependency bits is that the DDB memory is only accessed if a load instruction is detected

from the instruction decoder. The bitwidth of the DDB memory is equal to the number of ways in the L1 IC. The height of the DDB memory equals the number of sets in one L1 IC way multiplied by the number of instruction words per cache line. Figure 3.8 shows how the DDB memory is accessed. When a load operation is detected after decoding, the DDB memory is accessed using the set and line index bits of the program counter. The DDB memory outputs the dependency bits for each way of the given set and line index. For instance, the output from the DDB memory in Figure 3.8 is two bits wide, since the L1 IC is a two-way set associative cache. To get the correct dependency bit, the L1 IC hit vector is used to select the correct output from the multiplexer. To determine if there is a dependency between a load instruction and the upcoming instructions, a conventional in-order pipeline data dependency detector is used.

Figure 3.9 shows the modified datapath with an additional pipeline such that sequential access can be performed. With an additional pipeline, a few components have been moved from the original 3-stage pipeline. The Control and Status Registers (CSRs) is placed in the tag-access pipeline because the input data for CSRs are availbale in the tag-access pipeline. The forwarding logic has also been modified, because it now needs to forward data from both tag-access stage and data access stage.

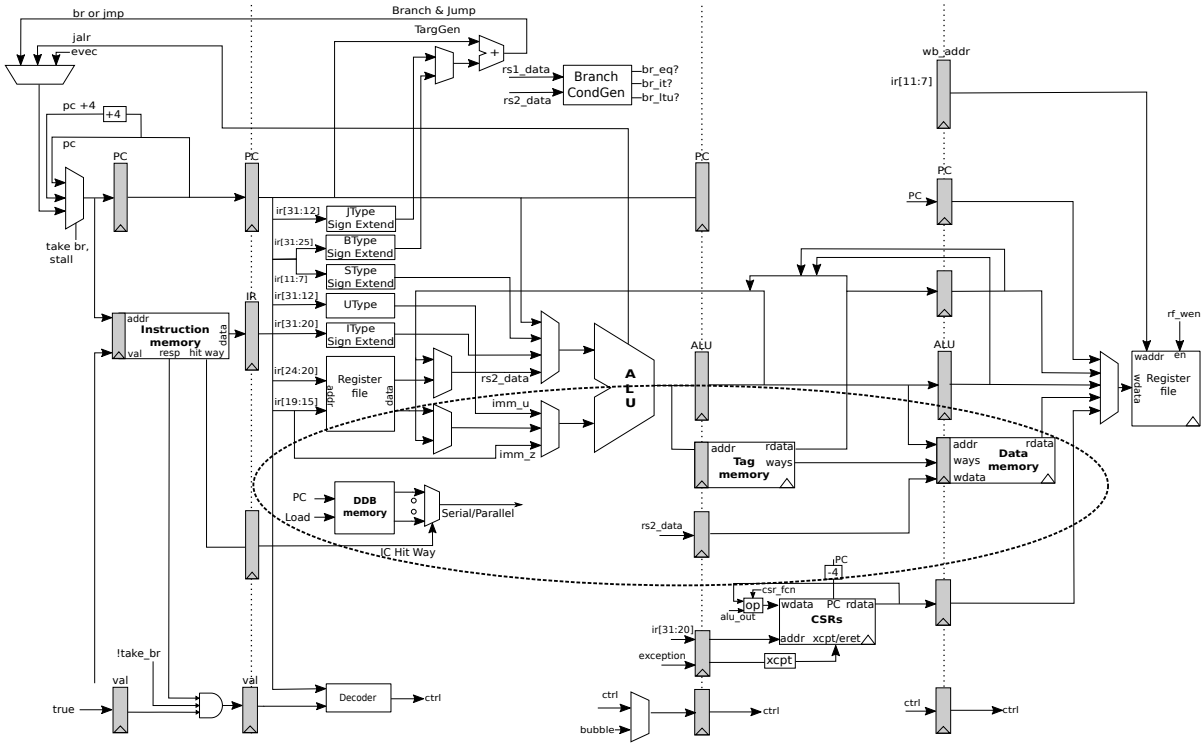

Figure 3.9: Z-Scale datapath with ELD^3 logic implemented.

3.3 SHA + ELD^3

When SHA and ELD^3 have been implemented separately, both implementations can be combined into the same core. As mentioned in Section 3.1, the SHA technique will only be beneficial when the displacement is small enough such that the base address index matches with the relative memory address index. Figure 3.10 shows how SHA and ELD^{3}are chosen for each load instruction. **1:**When the displacement is small, the halt tags are accessed but the DDB memory is not accessed. The tag and data ways are accessed in parallel, but SHA will halt the both tag and data ways using hit vector from halt tag access. **2:**When the displacement is too large for SHA, the halt tags are not accessed, but the DDB memory is accessed. The outcome of DDB memory will decide the next step taken. **2a:** If the DDB memory returns a dependency bit which is cleared, then the tag and data ways are sequentially accessed. **2b:** Or if the DDB memory returns a dependency bit which is set, the tag and data ways are accessed in parallel, such that the data can be forwarded to the following instruction to avoid a stall cycle.

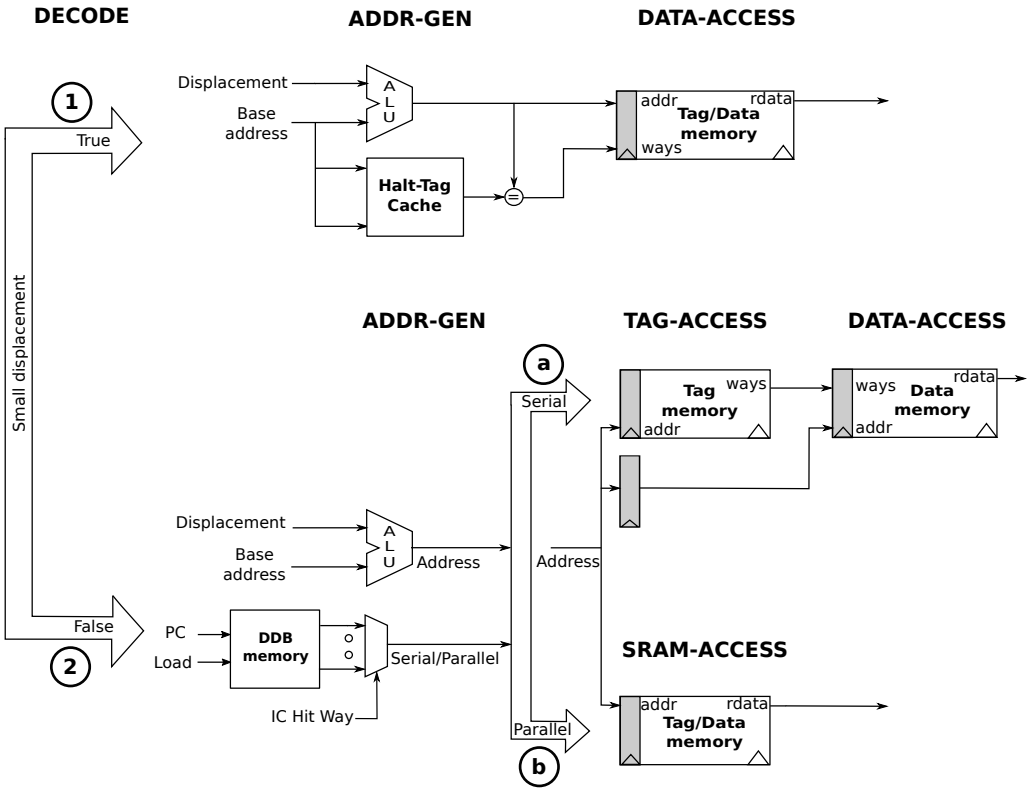

Figure 3.10: The impact of displacement when SHA and ELD^3 are combined.

Both SHA and ELD^3 are implemented as separated modules in Chisel. In order to combine both techniques and integrate the techniques into the existing SHMAC core, inheritance is used to extend classes of baseline model with SHA and ELD^3 classes. Because the implementation of ELD^3 requires several modifications of the existing baseline core, the ELD^3 code is integrated into the baseline module. This is illustrated in Code 3.1, where *ELDCore* consists of code for the baseline and ELD^3. Furthermore, the code for SHA technique exists in *SHACore* class which extends the ELDCore in order to combine both techniques together with the baseline. As mentioned in the Section 2.3.5, Chisel is used in this project, which is an embedded Domain-Specific Language (DSL) written in Scala, which introduces many new features and ways to design hardware. Scala is a multi-paradigm programming language in the sense that it supports both object-oriented and functional paradigms. One powerful feature built into Scala is *lazy val* values, which are lazy initialization pattern. When a *val* is declared as *lazy*, the definition is not executed before the first time the value is accessed. This is frequently used for shared values between the ELD^3 and SHA module in order to only evaluate the *val* once on the first access.

Code 3.1: Extending SHA and ELD^3 with the baseline

```
class ELDCoreIO extends Bundle{
  val imem = new FrontEndCpuIO().flip()
  val tmem = new CachePortIo(conf.xprlen)
  lazy val dmem = new CachePortIo(conf.xprlen)
  val ctl = new CtrlSignals().asInput
}

class ELDCore(cpuid: Integer) extends Module{
  lazy val io = new ELDCoreIO
  // Implementation of ELD
}

class SHACoreIO extends ELDCoreIO{
 override lazy val dmem = new CachePortIo(conf.xprlen)
 val htag_mem = new HTagPortIo(conf.shmacConf.d_ways)
}

class SHACore(cpuid: Integer) extends ELDCore(cpuid){
  override lazy val io = new SHACoreIO
  // Implementation of SHA
}
```

Chapter 4

Methodology

The caches in the SHMAC processor tile can be extensively configured. In this project, the configuration parameters for L1 DC and L1 IC have been modified for different cases during the evaluation process. However, unless something else is mentioned, the cache parameters are set to the values shown in Table 4.1

Data cache lines	2048
Words per line	4
Ways	4
Halt tag bitwidth	5
Tile memory words	512
Tile scratchpad words	32768

Table 4.1: Standard cache configuration for L1 DC and L1 IC

The effectiveness of the SHA and ELD^3 implementations were evaluated using the MiBench benchmark suite (Guthaus et al., 2001). The following tests, distributed across the test categories automotive, network and office, were used:

- **Bitcount (automotive):** This test counts the number of bits in an array of integers. The test includes five methods, and among those is an optimized 1-bit per loop counter.
- **Dijkstra (network):** This test constructs a graph, and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm.
- **Qsort (automotive):** This test sorts an array of strings into ascending order using the quick sort algorithm.

- **Stringsearch (office):** This test searches for given words in phrases using a case insensitive comparison algorithm.

The MiBench applications were compiled using the RISC-V GCC toolchain, which consists of GCC, Binutils, newlib and glibc ports. A similar proxy kernel as the riscv-pk was used on SHMAC to service system calls. The tests were performed on a computer running the SHMACsim simulator. No testing on hardware was performed.

The applications with relatively straightforward build processes were used, as the more complex builds were troublesome to cross-compile and run on SHMAC. In addition test applications, such as basicmath and susan, were not able to read the input files during execution. In other cases, different library files were not found, which made it difficult to compile the applications without doing changes to the test program code. Therefore, only test applications that worked out-of-the-box without considerable changes were chosen.

The implementation of SHA, ELD^3 and SHA+ELD^3 were evaluated by running MiBench application individually on SHMAC. This made it possible to compare the results between the different implementations. In addition to the performance counters for the baseline implementation, new performance counters were implemented for SHA and ELD^3 in order to evaluate the output results. These performance counters were used to evaluate the techniques and the energy savings presented in Section 5.

In order to test some of the added components, a few testbenches were created. The DDB memory for the ELD^3 technique, in addition to the dependency checker and halt-tag cache modules for the SHA technique, were tested with these testbenches where random values were loaded and stored into specific addresses.

4.0.1 L1 DC Energy

The effect of SHA, ELD^3 and SHA+ELD^3 on energy dissipation has been evaluated. The same energy evaluation method is used as in the STA paper (Bardizbanyan et al., 2013). Using Equation 4.1 and 4.2, the final energy values for load and store operations are obtained.

$$E_{read} = 4 \times E_{data\_read} + 3 \times E_{tag\_read} + E_{peripheral} \tag{4.1}$$

$$E_{write} = 1 \times E_{data\_write} + 3E_{tag\_read} + E_{peripheral} \tag{4.2}$$

Energy evaluation for SHA

The numbers for tag arrays, data arrays and peripherals are sourced from the paper on SHA (Moreau et al., 2016), and displayed in Table 4.2. These numbers form the basis for obtaining the energy dissipation values for load operations in a baseline L1 data cache and in an L1 data cache with the SHA technique implemented, down to a per-way granularity.

Component	Energy (pJ)
Read Halt	19.1
Write Halt	17.7
Read Tag	19.1
Write Tag	17.6
Read Data	26.5
Write Data	27.2
Peripheral	18.8

Table 4.2: Component energy for the different parts of an L1 data cache using SHA technique. Energy values from Moreau et al. (2016)

The energy dissipation numbers from Table 4.2 were used in the paper on SHA to quantify the reduction of the energy dissipation in an L1 data cache that implements the SHA technique. Table 4.3 shows a selection of the different data load access cases that can happen, and their associated energy dissipation. Only those cases that are relevant for the implementation of SHA in this project are included. BL is a baseline load in a standard L1 data cache, that does not implement the SHA technique. SHA0 is the case where the displacement is too large, so the halt tags are not accessed, and the load is performed without SHA. SHA2:X (gray rows) represents a successful speculative load, where X is the number of matching halt tags. This number corresponds to the number of ways in the L1 data cache that has been halted. The final case for loads is SHA3, where the displacement is low enough that a speculation is performed, but the speculation fails, so all ways have to be accessed anyway.

Case	Read Halt	Read Tag	Read Data	Peripheral	Energy (pJ)
BL		3	4	1	182.1
SHA0	0	4	4	1	201.2
SHA2:0	1	0	0	1	37.9
SHA2:1	1	1	1	1	83.5
SHA2:2	1	2	2	1	129.1
SHA2:3	1	3	3	1	174.7
SHA2:4	1	4	4	1	220.3
SHA3	1	4	4	1	220.3

Table 4.3: Components accessed for each load case, and the total energy dissipation. Adapted from Moreau et al. (2016)

Energy evaluation of ELD^3

Table 4.4 shows the energy for different components of the L1 DC when using the ELD^3 technique. The load and store operations include energy dissipated in the L1 DC peripheral circuit, which consists of the least recently used (LRU) replacement unit, the cache controller and the remaining multiplexers. The energy for a four-way tag read is 57.3 pJ, while the energy for a L1 DC four-way L1 DC data read is 106 pJ. When a sequential tag-data access with the ELD^3 is performed, the energy corresponding to three data arrays are not considered. That is because only a single data way is accessed during a sequential load. This results in a energy reduction of 79.5 pJ. If a cache miss is occured during a sequential load, no data way is accessed since the miss is detected earlier. In this case, there is a energy reduction of 106 pJ because the energy is saved for all the data way. For each load operation, the overhead of accessing the data dependency bit (DDB) memory is also evaluated.

Component	Energy (pJ)
L1 DC load	182.1
L1 DC store	103.3
L1 DC four-way tag read	57.3
L1 DC four-way data read	106.0
DDB memory read	8.6
DDB memory write	8.9

Table 4.4: Component energy for the different parts of an L1 data cache using ELD^3 technique. Energy values from Moreau et al. (2016)

Chapter 5

Results

In this section, all the results of the execution from the MiBench applications are presented. The results are either shown in tabulated form or in a graph. Section 5.1 will cover the results for only when SHA is implemented. Section 5.2 will cover the results from the ELD^3 implementation, before presenting the results from SHA+ELD^3 in Section 5.3. Note that both SHA and ELD^3 are implemented as individual components, which makes it possible to run them either individually or combined.

5.1 Results for SHA

Application	Small displacement(%)	Large displacement (%)
Bitcount	98.5	1.5
Dijkstra	38.3	61.7
Qsort	64.4	35.6
Stringsearch	35.9	64.1
Average	59.3	40.7

Table 5.1: Ratio of small and large displacements.

Table 5.1 shows the displacement ratio for load instructions when executing different MiBench applications. There is a speculation success when the displacement is small, and the halt tags are accessed from the halt-tag cache. The higher the speculation success rate is, the more useful is the SHA technique to reduce energy dissipation. From the table we can see that bitcount has the largest success rate of 98.5%, followed by qsort with a success rate of 64.4%. Dijkstra and stringsearch have large amount of loads with too large

displacement of 61.7% and 64.1% respectively. Across all benchmarks, about 60% of the loads have a small displacement, that the SHA technique can take advantage of.

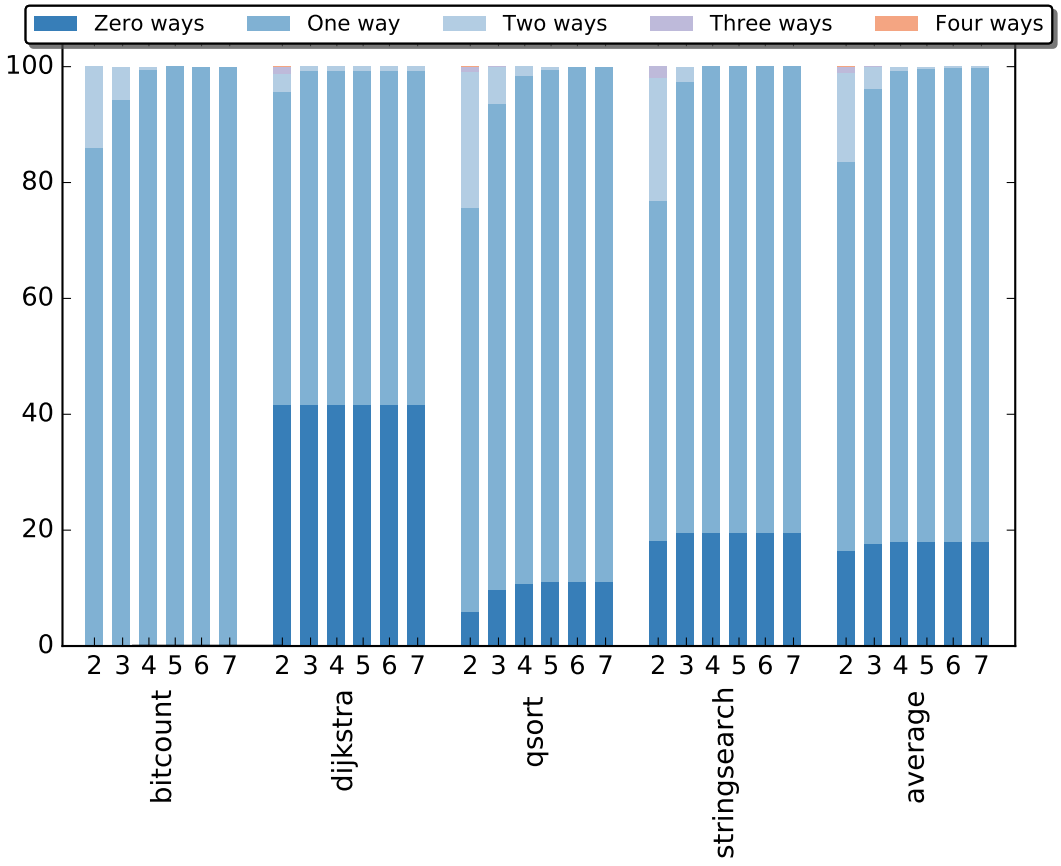

Figure 5.1: L1 DC ways accessed depending on the bit-width of the halt tags

The halt-tag cache is set up with the flexibility to adjust the width of the halt tag. The diplacement ratio shown in Table 5.1 does not affect the choice of halt-tag width. The halt-tag bit-width can be defined from 1 bit up to the bit-width of the tag stored in the data cache. However the halt-tag bit-width should be as small as possible in order to avoid the extra energy overhead. Figure 5.1 shows the speculation accuracy when running different MiBench applications and using widths between two and seven bits for the halt tag. We can see that a larger bit-width performs speculation with higher accuracy than a smaller bit-width. For instance, when 2 bit wide halt tag is used while running the qsort, 69% of the loads with small displacement, results in only one L1 DC way accessed, compared to 89% when using seven bit wide halt tag. The reason for the higher speculation accuracy when using wider halt tags is that a decreasing number of ways are accessed as a larger halt tag is

less likely to match with the tag. Also for bitcount and dijkstra, higher accuracy is shown when using larger halt tag bit-width. For dijkstra, 96,2% of load operations with small displacement results in one way access or zero way when there is a cache miss. Thus, the SHA technique can be used to detect cache misses earlier and also avoid accessing L1 DC ways as the requested data can not possibly reside in the L1 DC. This results in considerable amount of energy savings. On average, up to 98% of load operations with small displacement resulted in zero or one way accesses.

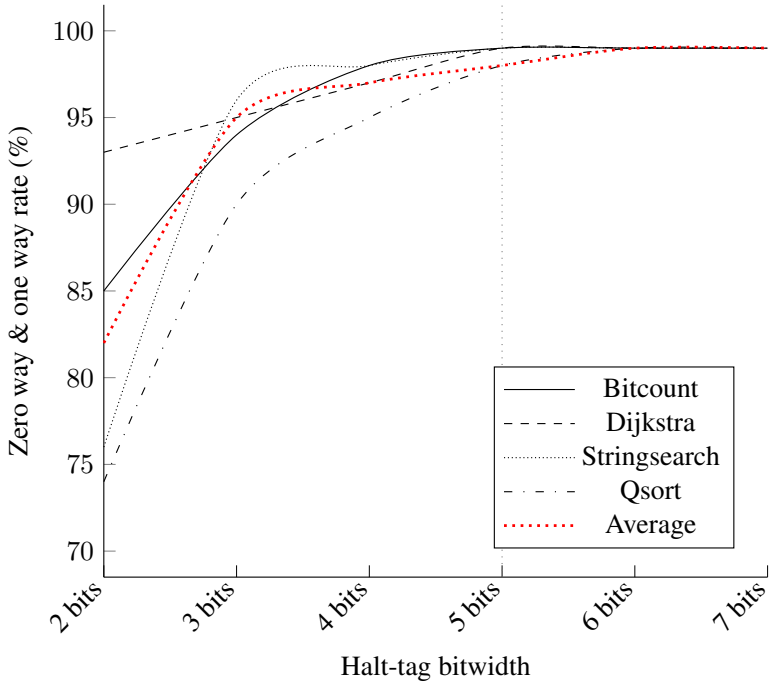

Figure 5.2: Zero and one way access rate depending on bit-width of the halt-tag

The better the speculation accuracy is, the more tag and data arrays will halt, thus more energy will be saved. However, we also need to keep in mind that a larger halt tag has less energy benefits because the energy cost of accessing and storing the halt tags will increase. From Figure 5.2 we can see that there is no gain in speculation accuracy when using a halt tag larger than 5 bits, except for qsort which have a slightly better speculation accuraty when using a 6 bit halt tag bitwidth. On average, 98% of loads with small displacement results in zero or one way accesses. The optimal halt-tag bit-width may depend on the application that is running. When a significant amount of tags have identical lower bits, it might be beneficial to store wider halt tags in order to filter out the ways that does not reside the requested data.

The L1 DC has also been evaluated with different cache configurations. When the halt tag bitwidth was set to two bits, there was slight difference in speculation accuracy between 2-way and 4-way set associative L1 where 2-way L1 DC accessed less amount of ways

for loads. The reason behind this is that more ways need to be compared with the halt tag when using four ways, and since only two bit halt tags were used for the tag comparison, the speculation accuracy decreased. Also, when the number of cache lines were increased, both 2-way and 4-way L1 DCs performed with better speculation accuracy. When 64 cache lines was used with four words per line, about 14% of total ways were accessed when using 2-way L1 DC compared to 12% when 2048 cache lines were used. It would also be interesting to change the number of words per line when using the SHA technique, because more words would share the same tag. Unfortunately, it was not possible to identify this in SHMAC, as the code for creating the cache is developed in such way that only four words per line can be used for L1 DCs.

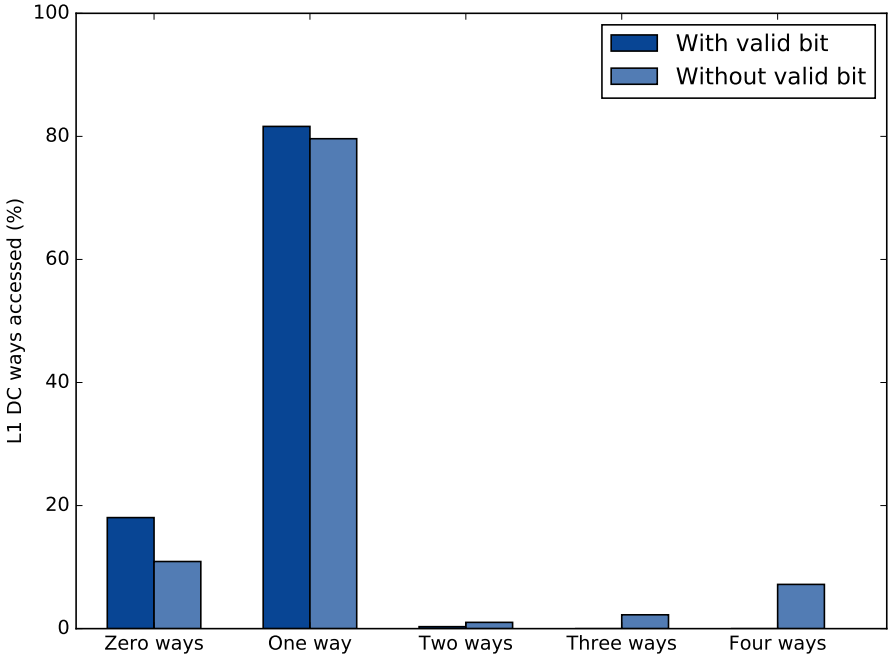

Figure 5.3: Difference in speculation accuracy when using valid bit for halt tag

As shown previously in Figure 3.2, we also store a valid bit in addition to the halt tag. Figure 5.3 shows the speculation accuracy when using a valid bit. We can see that there is a difference in the speculation accuracy, where 82% of the loads for one way is accessed when we check for validity of the halt tags, compared to 79% when we do not use a valid bit. The real difference is shown for four-way accesses, where no four-way access is performed when a valid bit is used. For instance, when we do not use a valid bit and the lower bits of the tag consist of all zeroes, it will actually be a halt-tag match if the cache line is empty. This incurs that unnecessary ways are accessed in the next cycle, that cannot possibly contain the requested data. By using a valid bit we are able to exclude empty cache lines when accessing the halt tags.

Figure 5.4 shows that the SHA technique reduces the load energy for all applications. In this case, the halt tag bit-width is set to 5 bits. The benchmark with least energy savings compared to the baseline cache are stringsearch and dijkstra with 19% energy saved, whereas the benchmark with the most energy saved is bitcount with an energy dissipation reduction of 58%. The main reason for the significant energy reduction is that the majority of load instructions for bitcount and qsort have small displacement, meaning that the SHA technique can reduce the energy dissipation for these instructions. For load instructions with too large displacement for SHA, a conventional load operation is performed, with all ways accessed, which gives the SHA technique no possibility to reduce the energy dissipation. The average load energy saved is 40%, compared to the load energy baseline cache without the SHA technique implemented.

The SHA technique has proven to be an effective approach to reduce the energy dissipation for more than half of the total loads across all applications. Given that there is no performance penalty, because all ways are accessed in the next cycle in case of a speculation misprediction, this technique is well suited to be implemented in a L1 DC where performance is an important factor.

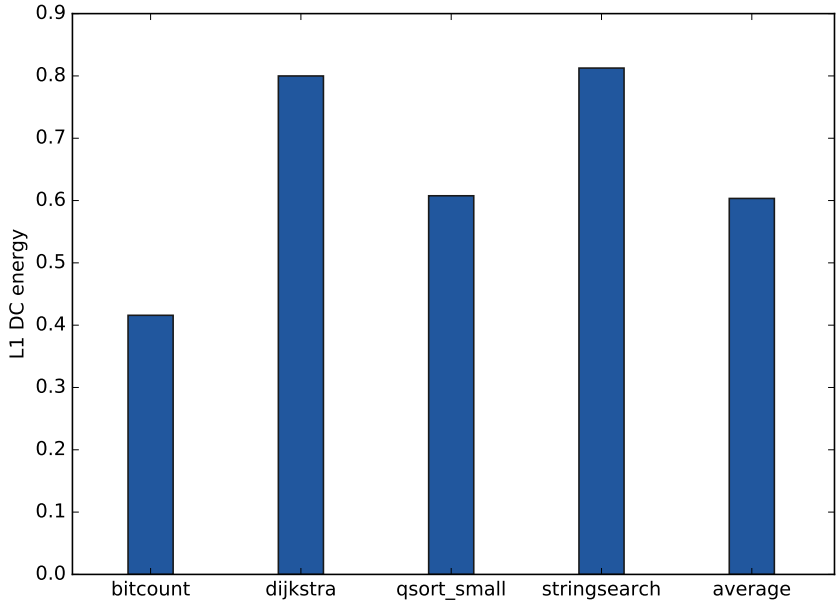

Figure 5.4: Total SHA load energy

5.2 Results for ELD^3

The SHA technique has shown to be very efficient for load operations when the displacement is small. However, when the displacement is too large for the SHA technique, all ways are accessed because the halt tags are not accessed in the address generation stage. The ELD^3 attempts to reduce the energy disspiation differently. The effect of the ELD^3 technique is depending on the number of data dependencies, rather than the displacements. This Section presents the results only for the ELD^3 implementation, while in Section 5.3 the results for the SHA+ELD^3 is presented.

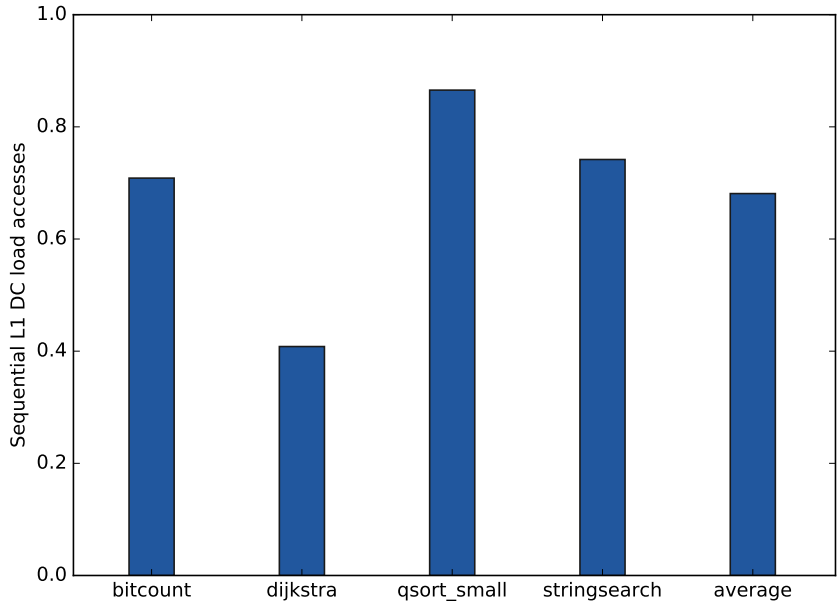

Figure 5.5: Loads operation that only access one L1 DC data way using ELD^3

ELD^3 is an effective technique for reducing the L1 DC energy dissipation by only accessing a single L1 DC data way when there is no data dependency with an upcoming instruction. Figure 5.5 shows the amount of load operations to the L1 DC that are accessed sequentially due to the ELD^3 technique. The application with highest ratio of sequential loads is qsort with 86%. Most of the load operations for stringserach and bitcount are also done sequentially. When running dijkstra, only 41% of the load operations are done sequentially, meaning that for most load operations, a data dependency with a following instruction is detected such that a pipeline stall would occur if the load operation was performed sequentially. On average, 67% load operations are done sequentially which indicates that ELD^3 can reduce energy disspiation for most load operations. The energy estimation is presented at the end of this section. Initially when the application is

executed, all dependency bits inside the DDB memory are cleared, meaning that the load operations are performed sequentially the very first time. The reason behind this is the higher possiblity that the load operation can be performed sequentially.

When the load operation is performed sequentially, only one data way is accessed in the next cycle if the data exist in the L1 DC. Likewise with SHA being able to detect a cache miss earlier, the ELD^3 technique is also able to detect a cache miss in the tag-access stage when the load operation is performed sequentially. This is done by checking if the hit vector bits for tag access are all zeros, indicating that there is no tag match. Note that the cache miss is known at the same time when performing the load operation parallelly with both tag and data ways accessed in same cycle, but a significant amount of energy is wasted because the data ways are accessed even when there is cache miss.

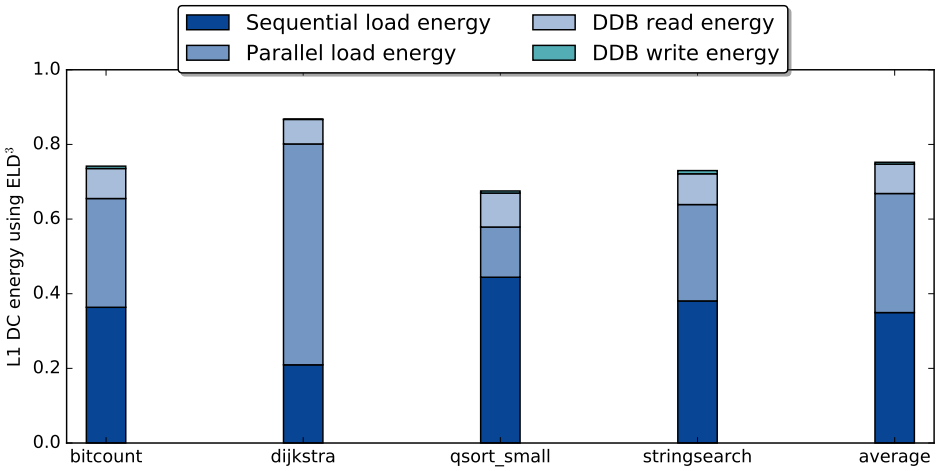

Figure 5.6: Total L1 DC energy when using ELD^3 relative to energy usage for baseline

Figure 5.6 shows the L1 DC energy disspiation for load operations. As previously shown in Figure 5.5, most of load operations for qsort are sequentially performed where only one data way is accessed, which reduces the energy dissipation significantly. We can see that the energy dissipation for qsort is reduced by 31%. The ELD^3 technique also reduces the energy disspiation for bitcount and stringsearch by 26% and 25% respectively. Since, most of the load operations for dijkstra are performed parallelly, the ELD^3 technique is only able to reduce the energy disspiation by 13%. We can also see from Figure 5.6 that there is an overhead of checking the DDB memory for each load operation and for updating the DDB memory during writeback. For each load operation, the energy overhead of checking the DDB memory is 8.6 pJ, whereas the energy overhead for updating the DDB memory is 8.9%. Since the DDB memory is accessed on each load operation, this overhead applies for both parallel and sequential loads. This means that the total energy disspiation can increase when using the ELD^3 technique if the amount of parallel load operation is dominant. On average, 25.7% of the overall energy disspiation is reduced compared to the baseline.

By having two pipelines for L1 DC accesses, there is an increased possibility for structural hazards as mentioned in Section 2.2. Structural hazards occur when there is a resource conflict where more than one instruction tries to access the same resource. This occurs when there are two consecutive load operations, where the first load is sequentially performed, and the second instruction has a data dependency, thus needs to access both tag and data ways in parallel. This cause a stall cycle such that the sequential load is completed before beginning with the parallel load operation. On average 3.9% of loads cause a stall cycle due to this kind of structural hazard, where qsort is the biggest sufferer with 5.8% of loads causing a stall cycle.

5.3 Results for SHA and ELD^3

The results for SHA and ELD^3 techniques running separately have shown that both techniques are effective in reducing the energy disspiation for L1 DC when it is possible. However, for both techniques running separately, there are load operations where it is not possible to do way halting (too large displacement) when using SHA or sequential loads (data dependency) when using ELD^3. Since SHA and ELD^3 techniques can capture different categories of loads, a combined SHA+ELD^3 can increase the number of loads that lead to data accesses with reduced energy dissipation. In this section, the results for SHA+ELD^3 will be presented.

When the displacement is small, the SHA technique is always selected. This is is mainly because about 60% of all load operations on average have small displacement such that SHA can be used to reduce the energy disspiation. In addition, entire tags are accessed to create the hit vector of tag comparison when using ELD^3, compared to only accessing halt-tags when using SHA. Also, the SHA technique ensure that there is no performance penalty when there is a speculation failure. Only an overhead of the halt-tag access will incure. Whereas for ELD^3, a performance penalty occurs when wrong dependency information is stored inside DDB memory and the load operation is sequentially accessed. We can see from Table 5.2 that SHA+ELD^3 suits well for dijkstra, qsort and stringsearch as these applications have considerable amount of loads with too large displacment for the SHA technique.

Application	**SHA loads (%)**	**ELD^3 loads loads (%)**
Bitcount	98.5	1.5
Dijkstra	38.3	61.7
Qsort	64.4	35.6
Stringsearch	35.9	64.1
Average	59.3	40.7

Table 5.2: Load distribution between SHA and ELD^3

Figure 5.7 shows the load distribution between SHA and ELD^3. We can see that SHA is very efficient for bitcount where over 98% of loads have small displacement. Since the dependency bit is not accessed for loads with small displacements, the DDB memory will not incure any energy overhead due to DDB memory read or update. For dijkstra, the ELD^3 is able to capture 24% of loads with large displacement. However, due to data dependencies, the majority of loads with large displacement are parallelly accessed. For qsort and stringsearch, 29% and 57% of the loads with large displacement are sequentially accessed. On average, the ELD^3 technique is able to reduce energy dissipation for 27% of total loads.

Because the majority of loads are performed by the SHA technique, the overall impact of the ELD^3 techniuqe reduces in SHA+ELD^3 compared to when only ELD^3 is used. Looking at Figure 5.5 again, we can see that the ELD^3 techniuqe is able to perform sequential loads for most loads when running bitcount. However, when the SHA+ELD^3 technique

is used, the impact of the ELD^3 technique is minimal for bitcount, because most of the load instructions have small displacement. This means that there is an overlap for loads where both SHA and ELD^3 can be applied. In best case, the ELD^3 technique would be able to reduce energy disspiation for all loads with large displacement instead of the loads where it overlaps with the SHA technique. Since the DDB memory is not read before the displacement speculation for SHA is performed, the DDB memory is not adding any overhead by reading the dependency bit for instructions where SHA will be used.

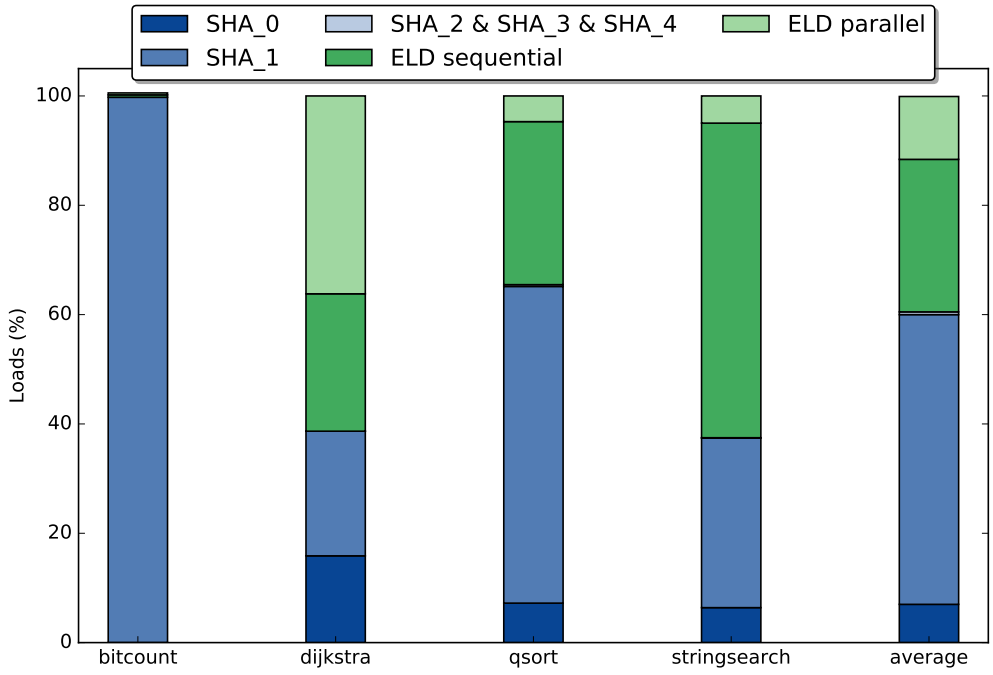

Figure 5.7: Load distribution between SHA and ELD^3.

Figure 5.8 shows the L1 DC energy disspiation for load operations when SHA+ELD^3 is running. For bitcount, SHA is able to capture almost all loads, which reduces the overall energy disspiation by 58%. And because the ELD^3 technique is used only for a small number of loads for bitcount, the DDB memory overhead is minimal. For dijkstra, the ELD^3 technique is able to use some of the loads with large displacement and access these loads sequentially. However, most of the loads are still parallelly accessed due to large displacment and high amount of data dependency, which results in an overall energy reduction of 35%. For qsort and stringserach, the ELD^3 technique is able perform sequential loads for most of the loads with large displacement. The overall energy reduction for qsort and stringsearch are 54% and 49% respectively. On average, the overall energy disspiation for is reduced by 49%.

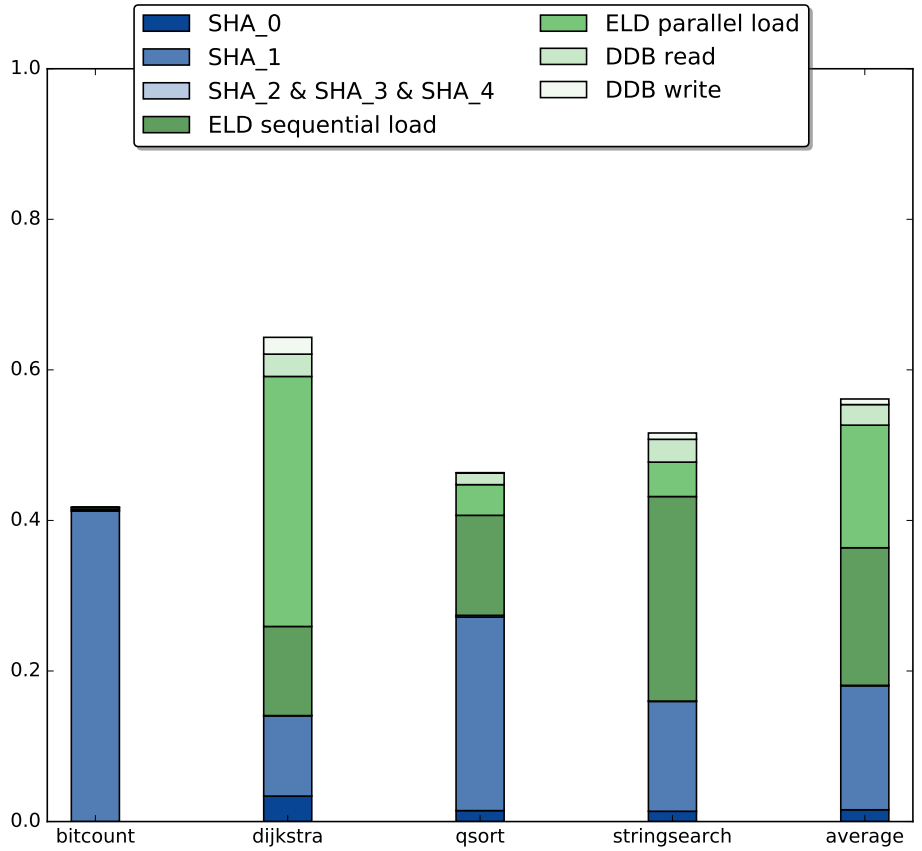

Figure 5.8: L1 DC energy for SHA+ELD^3

Chapter 6

Related work

In this Section, some of the relevant work to the SHA and ELD^3 technique will be presented. The relative merits of the techniques will also be compared.

6.1 Way-Halting Cache (WHC)

The idea of halting one or more ways in a cache is not unknown. The WHC technique was proposed to accomplish just that, by performing a fully associative halt-tag check in parallel with the decoding of the L1 data cache index Zhang et al. (2005). However, an implementation of this technique may be impractical. To reduce word line lengths, memories are often banked, and the data and tags are stored separately. Since the halt-tag memory is fully associative, it would either need to route its signals to all the different banks and memories, or be duplicated for each bank. Respectively, this would either introduce delays or a higher use of energy and area. Another impracticality of this technique is that it would require a custom SRAM implementation, which is not easily available and would be costly. The SHA technique, on the other hand, does not require a custom SRAM implementation, as the halt tags are accessed in the address generation stage. This means that the tags and data that need to be accessed is already determined by the memory access stage.

6.2 Data Filter Cache (DFC)

Data Filter Cache is based on the idea that a small cache at the top of the memory hiearchy can help with reducing the number of accesses to the L1 cache. The advantage with DFCs is that the it reduces energy since the dynamic energy to access the DFC is mucc less

than the L1 cache. However, conventional DFCs have high miss rate, which increases the execution time. Instead, a more energy efficient way is to implement a practical DFC that can be accessed early in the pipeline and then transfer a line over multiple cycles(Alen Bardizbanyan, 2013). A speculative access of DFC is performed in the address generation stage when the displacement is not too large. On a speculation success, the value is obtained one clock cycle earlier, and on a speculation failure the L1 DC can be accessed in the next cycle. This results in improved performance and eliminates accesses to DTLB and L1 DC on DFC load hits. This technique share the same core idea as SHA, that speculation access can help in eliminating L1 DC accesses, thus reduces the energy dissipation.

6.3 Partial Tag Comparison (PTC)

Another technique is the PTC, where the tags and data of the L1 data cache would be accessed after the partial tag comparison is completed Min et al. (2004). Because the partial tags are compared after the index is decoded, the signals would have to be connected to all banks and memories, which is the same issue as the WHC technique has. Similarly, the PTC technique would also be dependent on a custom SRAM implementation, which the SHA technique does not need.

6.4 Speculative Tag Access (STA)

The STA technique works by accessing the L1 data cache tags during the address generation stage Bardizbanyan et al. (2013). If there is a hit in the cache (the speculation succeeded), the data in the L1 data cache will be accessed in the memory access stage. While all tags will still be accessed in parallel, the STA technique reduces the number of accessed data arrays. This will however only positively impact the read energy, but not the store energy, as a store will only access a single data array anyway.

Because the STA technique need to access the data translation lookaside buffer (DTLB) in both the address generation and the memory access stage, the input signals to the DTLB has to be routed from both these stages. This increases the complexity compared to a conventional L1 data cache. In addition, forwarding logic is used to produce the input from the address generation stage. This could be a source of increased delay, and may even put the DTLB on the critical path. The SHA technique avoids putting the DTLB on the critical path by accessing the halt tags in the address generation stage, and only accessing the DTLB in the memory access stage.

6.5 Way-Prediction

Way-prediction is a technique that is used to predict the matching way of a cache access, and then only access that way Powell et al. (2001). This reduces the energy needed for the cache accesses. Way-prediction is dependent on a good prediction source for the predictions, and this prediction source has to be available early in the pipeline. The prediction source can be the load instruction program counter (PC), for instance, or an approximate address formed by XORing the load's source register with the load offset. Different prediction sources have different trade-offs between early availability in the pipeline and prediction accuracy. The PC-based prediction is available very early, but the XOR-based prediction is more accurate.

The way-prediction technique can retain the performance of a parallel cache access, while achieving an energy-delay of a sequential cache access. Relative to the SHA technique, the way-prediction technique accesses only one way if the prediction succeeds, but all ways if the prediction fails. The SHA technique on the other hand, speculatively determines which ways not to access, and may thereby avoid accessing any number of ways when the speculation succeeds.

Chapter 7

Conclusion and Future Work

Improving the energy-efficiency of computing is an important area of research, and there is potential for reducing the energy dissipation in caches. In this thesis, it is shown how accesses to a set-associative L1 DC can be made significantly more energy efficient by implementing two techniques. Speculative Halt-Tag Access (SHA) technique represents a practical way-halting approach to eliminate ways that cannot possibly contain the requested data. SHA can be performed when the displacement for address calculations is small, by accessing the low-order tag bits for each way in the address generation stage that act as a filter for which tag and data ways of the set-associative L1 DC to access. The use of SHA does not incur any performance penalties, thus suits well for L1 DCs where performance is important. The approach gives an average combined zero-way and one-way access rate of at least 83% with a two bit halt tag for loads with small displacement. This indicates that the SHA technique can work well, even with halt tags as small as two bits. When the halt tag bitwidth is set to 5 bits, up to 99% of load operations with small displacement results in zero or one way accesses as larger halt tag is less likely to match with the tag. On average, the overall energy disspiation is reduced by 40% for load operations with small displacement. The second technique, Early Load Data Dependence Detection (ELD^3) , represents an effective approach with low overhead, that can detect if the load operation has a data dependency with a following instruction that will cause pipeline a stall. This can efficiently be done by storing the dependency information of L1 IC instruction in a Data Dependency Bit (DDB) memory. The DDB memory can then be accessed in the address generation stage when load instruction is detected. If there is no data dependency, the tags and data ways are accessed in sequential such that only one data way is accessed. The ELD^3 technique gives an overall energy reduction of 26%.

It is also shown in this thesis, how accesses to L1 DC can be significantly more energy-efficient by combining both techniques. When a load operation is detected in the address generation stage, the SHA technique is used if the displacement is small. If the displacement is too large for the SHA technique, the ELD^3 technique is used to access tag and

data ways sequentially if there is no data dependency with the upcoming intructions. This combination of techniques gives an overall energy reduction of 43%.

7.0.1 Future Work

There are several ways in which this work may be extended or improved. Some of the suggestions are presented below.

Reduce the size of Data Dependency Bit (DDB) memory

The DDB memory is used to store dependency information for instructions in L1 IC as described in Section 3.2.1. The size of the DDB memory is depending on the size of L1 IC, since one bit is assigned for each word inside the cache. This means that the DDB memory can get considerable large when large L1 IC is used. One way of reducing the size of DDB memory, is by applying one bit for multiple words in L1 IC. For example can two words share the same dependency bit, and then have a dependency update policy for when the dependency bit will be updated if the dependency information is not correct for the corresponding instruction. This can be done by having a small separate memory beside the DDB memory, where one bit corresponds to one set in DDB memory.

Efficient accesses to the scratchpad memory

As the SHMAC processor tile also can consist of other memories on the processor tile such as the scrachpad memory, it can be possible to performed energy efficient memory accesses for these memories in addition to the L1 DC. An analysis should be done of how frequently these memories are accessed, in order to decide if the implementation of SHA and ELD^3 can reduce the energy dissipation.

Bibliography

Alen Bardizbanyan, Magnus Sjalander, D. W. P. L.-E., 2013. Designing a Practical Data Filter Cacheto Improve Both Energy Efficiency and Performance. In: ACM Transactions on Architecture and Code Optimization (TACO). ACM.

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: Design Automation Conference.

Bardizbanyan, A., Själander, M., Whalley, D., Larsson-Edefors, P., 2013. Speculative Tag Access for Reduced Energy Dissipation in Set-Associative L1 Data Caches. In: 2013 IEEE 31st International Conference on Computer Design (ICCD). IEEE, pp. 302–308.

Borkar, S., Chien, A. A., 2011. The Future of Microprocessors. Communications of the ACM 54 (5), 67–77.

Carvalho, C., January 2002. The Gap between Processor and Memory Speeds. Tech. rep., Departamento de Informática, Universidade do Minho.

Celio, C., Love, E., May 2014. About The Sodor Processor Collection. `https://github.com/ucb-bar/riscv-sodor`, visited 2016-11-29.

Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., LeBlanc, A. R., 1974. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. IEEE Journal of Solid-State Circuits 9 (5), 256–268.

Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., Burger, D., 2011. Dark Silicon and the End of Multicore Scaling. In: Computer Architecture (ISCA), 2011 38th Annual International Symposium on. IEEE, pp. 365–376.

Ghoshal, S., 2011. Computer Architecture and Organization: From 8085 to core2Duo & beyond. In: Computer Architecture and Organization: From 8085 to core2Duo & beyond. Pearson Education India, p. 207.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., Brown, R. B., 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. IEEE, pp. 3–14.

Lefsaker, S., December 2014. Prototyping a RISC-V based SHMAC with Chisel. https://github.com/sondrele/Chisel-SHMAC/blob/master/doc/project-report.pdf.

Min, R., Xu, Z., Hu, Y., Jone, W.-B., 2004. Partial Tag Comparison: A New Technology for Power-Efficient Set-Associative Cache Designs. In: VLSI Design, 2004. Proceedings. 17th International Conference on. IEEE, pp. 183–188.

Moreau, D., Bardizbanyan, A., Själander, M., Whalley, D., Larsson-Edefors, P., March 2016. Practical Way Halting by Speculatively Accessing Halt Tags. In: Proceedings of the IEEE International Conference on Design, Automation, and Test in Europe (DATE).

Nevine AbouGhazaleh, Bruce Childers, D. M., Melhem, R., December 2005. Energy Conservation in Memory Hierarchies using Power-Aware Cached-DRAM. Tech. rep., Department of Computer Science, University of Pittsburgh.

Oh, S.-H., Monroe, D., Hergenrother, J., 2000. Analytic Description of Short-Channel Effects in Fully-Depleted Double-Gate and Cylindrical, Surrounding-Gate MOSFETs. IEEE electron device letters 21 (9), 445–447.

Powell, M. D., Agarwal, A., Vijaykumar, T., Falsafi, B., Roy, K., 2001. Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping. In: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture. IEEE Computer Society, pp. 54–65.

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

Taur, Y., Nowak, E. J., 1997. CMOS Devices Below 0.1/spl mu/m: How High Will Performance Go? In: Electron Devices Meeting, 1997. IEDM'97. Technical Digest., International. IEEE, pp. 215–218.

Waterman, A., Lee, Y., Patterson, D. A., Asanović, K., May 2016. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Tech. Rep. UCB/EECS-2016-118, EECS Department, University of California, Berkeley.
URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

Wulf, W. A., McKee, S. A., December 1994. Hitting the Memory Wall: Implications of the Obvious. Tech. rep., Department of Computer Science, University of Virginia.

Zhang, C., Vahid, F., Yang, J., Najjar, W., 2005. A Way-Halting Cache for Low-Energy High-Performance Systems. ACM Transactions on Architecture and Code Optimization (TACO) 2 (1), 34–54.

Zhang, T., May 2014. A study of DRAM optimization to break the memory wall. Tech. rep., The Pennsylvania State University.