# A Hardware Implementation of the Snappy Compression Algorithm

*Kyle Kovacs*

Acknowledgement

I want to especially thank Yue Dai for working closely with me on writing the RTL for the compression accelerator. This was made possible in part by her hard work.
I also greatly thank Adam Izraelevitz for being a mentor, friend, and role model to me throughout my time as a grad student at Berkeley.
Vighnesh Iyer and Paul Rigge provided a lot of needed assistance early on in the course of the project, and I am thankful for their help.
The members of the ADEPT lab Chick Markley, Alon Amid, Nathan Pemberton, Brendan Sweeney, David Bruns-Smith, Jim Lawson, Howard Mao, Edward Wang, Sagar Karandikar, David Biancolin, and Richard Lin were also instrumental in assisting me with learning how to use various tools and software quirks.
Thanks to all those listed here and others for supporting me.

# A Hardware Implementation of the *Snappy* Compression Algorithm

by

Kyle Kovacs

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Spring 2019

The thesis of Kyle Kovacs, titled A Hardware Implementation of the *Snappy* Compression Algorithm, is approved:

Chair ________________________________________ Date _______________

________________________________________ Date _______________

University of California, Berkeley

**A Hardware Implementation of the *Snappy* Compression Algorithm**

## Abstract

A Hardware Implementation of the *Snappy* Compression Algorithm

by

Kyle Kovacs

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Krste Asanović, Chair

In the exa-scale age of big data, file size reduction via compression is ever more important. This work explores the possibility of using dedicated hardware to accelerate the same general-purpose compression algorithm normally run at the warehouse-scale computer level. A working prototype of the compression accelerator is designed and programmed, then simulated to asses its speed and compression performance. Simulation results show that the hardware accelerator is capable of compressing data up to 100 times faster than software, at the cost of a slightly decreased compression ratio. The prototype also leaves room for future performance improvements, which could improve the accelerator to eliminate this slightly decreased compression ratio.

# Contents

# List of Figures

# List of Tables

## Acknowledgments

I want to especially thank **Yue Dai** for working closely with me on writing the RTL for the compression accelerator. This was made possible in part by her hard work.

I also greatly thank **Adam Izraelevitz** for being a mentor, friend, and role model to me throughout my time as a graduate student at Berkeley.

**Vighnesh Iyer** and **Paul Rigge** provided a lot of needed assistance early on in the course of the project, and I am thankful for their help.

The members of the ADEPT lab (in no particular order) **Chick Markley**, **Alon Amid**, **Nathan Pemberton**, **Brendan Sweeney**, **David Bruns-Smith**, **Jim Lawson**, **Howard Mao**, **Edward Wang**, **Sagar Karandikar**, **David Biancolin**, and **Richard Lin** were also instrumental in assisting me with learning how to use various tools, software, quirks of Chisel, and other odds and ends.

Thanks to all those listed here and others for supporting me.

# Chapter 1

# Introduction

## 1.1 Background

### Why is Compression Important?

Data compression is important across a wide range of applications, especially as hundreds, thousands, and millions of photos, videos, documents, server logs, and all manner of data streams are produced, collected, broadcast, and stored across the billions of electronic devices in the world today. Sizes and amounts of data are continuing to grow, and as they do, it is ever more essential to find ways to increase our storage capacity. While disk technology can help alleviate storage capacity needs, another option is to decrease the amount of data that needs to be stored. This is where compression can play a major role in data-heavy applications.

### What is Compression?

There are two major classes of data compression: lossy and lossless. "Lossy" compression, as the name implies, loses some of the original content through the compression process. This means that the transformation is not reversible, and so the original data cannot be recovered. Lossy compression is acceptable in a variety of domains (especially in images) because not all the original data is necessary to convey meaning. Compression that preserves the original data is called "lossless." This means that after being compressed, the data can undergo an inverse transformation and end up the same as the original bit-stream. Lossy compression works by eliminating unnecessary information, like high-frequency noise in images, whereas lossless compression works by changing the way data is represented. Lossless compression will be the subject of the remainder of this paper.

There are many and various lossless compression algorithms [11], and the field is well developed. Some algorithms offer high compression ratios (i.e. the size of the output data is very small when contrasted with the size of the input data) while others offer higher speed (i.e. the amount of time the algorithm takes to run is relatively low). Different types of data tend to be more or less compressible depending on the algorithm used in combination with the data itself.

## 1.2 This Project

### Project Overview

This project is focused around a hardware accelerator capable of running a data compression algorithm. Throughout the project's lifetime, the accelerator was designed, built, tested, and evaluated on the basis of functionality and performance. This compressor will contribute to Berkeley and the research community by taking its place in the RocketChip [2] ecosystem as a standalone, open-source implementation of a hardware accelerator. The accelerator code will be hosted on Github [13], and will be available for researchers at Berkeley and the general public to use and/or modify.

### Project Context

Because this research was conducted within the ADEPT lab at UC Berkeley, it utilizes the favorite set of tools chosen by the lab i.e. Chisel, RocketChip, FireSim, RISC-V, and others. These are described in more detail in the following sections.

# Chapter 2

# Implementation

## 2.1 Context and Structure

### RocketChip

Rocket is an in-order 5-stage scalar RISC-V processor designed at UC Berkeley in the Chisel [4] [3] hardware construction language. The Rocket core is open-source, and comes alongside an open-source SoC (System-on-Chip) generator called RocketChip [2]. The RocketChip generator was chosen for this project because it is the marquee open-source RISC-V core. It provides a useful platform on which to develop RISC-V software and associated hardware.

### The Rocket Custom Co-processor (RoCC) Interface

The RISC-V instruction set architecture [1] sets aside a large encoding space specifically for custom instructions and ISA extensions. These instructions can be specifically designed by users who wish to add extra functionality to the ISA. One such extension format designed to be used with the Rocket core is the Rocket Custom Co-processor, or RoCC interface.

RoCC instructions are formatted as follows: one 7-bit field (`opcode`), which usually specifies which co-processor the instruction will be routed to, three 5-bit register specifier fields (`rd`, `rs1`, and `rs2`) for a destination register and two sources, one 7-bit field (`funct`) for specifying an operation, and three 1-bit fields (`xd`, `xs1`, and `xs2`) to indicate whether or

| bits | 31 25 | 24 20 | 19 15 | 14 | 13 | 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|---|
| | `funct7` | `rs2` | `rs1` | `xd` | `xs1` | `xs2` | `rd` | `opcode` |
| length | 7 | 5 | 5 | 1 | 1 | 1 | 5 | 7 |

Figure 2.1: **The RoCC instruction format.** The three 5-bit fields correspond to the source registers and the destination register. The 1-bit fields act as switches to specify whether or not the register fields are to be used. The 7-bit fields cover the rest of the encoding space, acting as routing and operation selection information.

not the register fields are to be used. If either or both of the `xs*` bits are set, then the bits in the appropriate architectural registers specified by the `rs*` fields will be passed over the connection bus to the co-processor; likewise, if they are clear, then no value will be sent for that source register. Similarly, if the `xd` bit is set, then the issuing core is meant to wait for the co-processor to send a value back over the connection bus, at which point it will be stored in the register specified by `rd`.

Instructions of this format can be hard-coded as assembly sequences in user-level C code and run on a Rocket core with an appropriate co-processor attached. Modification of the compiler-assembler RISC-V toolchain would allow normal, high-level C code to be translated into these custom instructions, but that possibility will not be explored here.

## Custom RoCC API

The compressor has two basic functions: `compress` and `uncompress`. However, since each of these functions needs three arguments (source, destination, and length), an additional setup instruction – `setLength` – is required. The `setLength` instruction executes in one cycle, uses one source register, and does not use a destination register. It simply loads a value from a register in the CPU into the length register within the compressor. The `compress` instruction uses `rs1` for the source pointer and `rs2` for the destination pointer, and it uses `rd` to return the length of the compressed stream. The `uncompress` function uses the source register fields in the same way, but it uses `rd` to return a success flag instead.

Figure 2.2 shows the top-level API (Application Programming Interface) for using the compressor. Each of the three instructions has its own `funct` identifier. This, along with the `opcode` field (specified by `CHANNEL`) gets packaged up into the proper instruction bits via the `ROCC_INSTRUCTION` macros. These macros construct assembly instructions for the appropriate fields. The C functions make a `fence` call to synchronize the memory before issuing the RoCC instructions in order to ensure that memory re-ordering does not affect the functionality of the compressor.

In order to use these functions properly, the calling process must maintain exclusive access to the region of memory in question. In other words, the compressor relies on software to manage memory access rights. The compressor is allowed to modify main memory between the given destination address and that address plus the maximum compressed length determined by the given input length. It is also allowed to read from the given source address up to the source address plus the given input length.

## Chisel

Chisel is a Scala-embedded hardware construction language developed at UC Berkeley. Chisel code running in the Scala runtime environment produces Flexible Intermediate Representation of Register Transfer Level (FIRRTL) code [9] through a process called elaboration. Once FIRRTL is generated, it can be either passed to a simulation environment, such as

```
// Include the ROCC_INSTRUCTION macros
#include "rocc.h"

// The RoCC channel that this accelerator lives on
#define CHANNEL 3

// Custom FUNCT fields for the accelerator
#define COMPRESS   0   // Compress data
#define UNCOMPRESS 1   // Uncompress data
#define SET_LENGTH 2   // Specify the length field

// Compresses LENGTH bytes from SOURCE into DEST.
// Returns the compressed length.
static inline unsigned long compress(char* source,
                                     int uncompressed_length,
                                     char* dest)
{
    unsigned long compressed_length;
    asm volatile ("fence");
    ROCC_INSTRUCTION_S(CHANNEL, uncompressed_length, SET_LENGTH);
    ROCC_INSTRUCTION_DSS(CHANNEL, compressed_length, source, dest, COMPRESS);
    return compressed_length;
}

// Decompresses LENGTH bytes from SOURCE into DEST.
// Returns non-zero on failure.
static inline unsigned long uncompress(char* source,
                                       int compressed_length,
                                       char* dest)
{
    unsigned long success;
    asm volatile ("fence");
    ROCC_INSTRUCTION_S(CHANNEL, compressed_length, SET_LENGTH);
    ROCC_INSTRUCTION_DSS(CHANNEL, success, source, dest, UNCOMPRESS);
    return success;
}
```

Figure 2.2: **RoCC API code snippet.** `rocc.h` provides the appropriate assembly macros. The three instructions, `compress`, `uncompress`, and `setLength` are defined. Two functions, `compress(source, uncompressed_length, dest)` and `uncompress(source, compressed_length, dest)` provide the main library API for using the compressor. Each instruction includes a `fence` to preserve memory operation ordering.

Treadle [14] or the FIRRTL Interpreter [6], or it can be used to emit Verilog code for other simulation environments like VCS [15] or Verilator [16].

Chisel provides hardware designers with powerful parameterization options and abstraction layers unavailable to programmers of classical hardware description languages like Verilog or VHDL. Chisel circuits are thought of as generators rather than instances of hardware, viz. precise types, bitwidths, and connections are not known in a Chisel circuit until after it has been elaborated. Chisel was chosen as the design language for this project because of its flexibility and because it is tightly integrated with the RocketChip ecosystem.

## 2.2 Overview

### Compression Algorithm

The compression algorithm chosen for this work is called Snappy [12]. Snappy compression is a dictionary-in-data sliding-window compression algorithm, meaning that it does not store its dictionary separate from the compressed data stream, but rather uses back-references to indicate repeated data sequences. This algorithm was chosen because of its wide use and portability. Google's open-source C++ implementation of the algorithm [7] was used to guide the development of the hardware platform.

#### Snappy Encoding Format

The first few bytes in the compressed data stream contain the length of the uncompressed data. These bytes are stored as a variable-length integer, meaning the number of bytes it takes to encode the value of the integer is proportional to the value of the integer. This is achieved by using the most significant bit of each byte as a flag which signifies whether or not there are more bytes to follow.

After the uncompressed length comes a series of "literal" and "copy" substreams, each consisting of a tag byte and possible additional length bytes. Literals encode a certain number of bytes from the input that are exactly the same in the output, while copies encode a length and a backwards offset to the reference data. Table 2.1 shows the different encodings for the Snappy substreams. Literals use `0b00` in the least significant bits of the first tag byte, whereas copies use `0b01`, `0b10`, or `0b11`, depending on the required offset length — longer back-references require more tag bytes.

To generate this stream of literal and copy substreams, a sliding-window approach is taken. Thus, there is a limit to how far back in the stream a copy can reference. Because of this, a sliding window equaling the size of the longest offset available (four bytes of offset information) is sufficient because the semantics of the compressed stream do not allow for back-references longer than this.

This project limits the length of a literal substream to 60 bytes, meaning the literal tag is always only a single byte. This choice was made because the compressor processes bytes one by one, copying into an output buffer. The literal tag is left blank and filled in later.

| Tag bits | Type | Description |
|---|---|---|
| `ssss_ss00`<br>`cccc_cccc`<br>`cccc_cccc`<br>`cccc_cccc`<br>`cccc_cccc` | Literal | The six `s` bits are used to indicate the length of the data. A length of 60, 61, 62, or 63 indicates that the following 1, 2, 3, or 4 byte(s) will specify the length, rather than the tag byte specifying the length. Hence, the `c` bits may or may not exist as part of the tag. |
| `ddds_ss01`<br>`dddd_dddd` | Copy | The `s` bits encode the length of the copy in bytes. The `d` bits encode the byte offset backwards into the data stream. |
| `ssss_ss10`<br>`dddd_dddd`<br>`dddd_dddd` | Copy | Same as above. |
| `ssss_ss11`<br>`dddd_dddd`<br>`dddd_dddd`<br>`dddd_dddd`<br>`dddd_dddd` | Copy | Same as above. |

Table 2.1: **Types of Snappy substreams.** Literal substreams can have anywhere between 1 and 5 bytes of tag data. For this implementation, the maximum literal length is 60, meaning that all literal tags are just one byte. There are three different formats for copy substreams; the choice of which copy tag format to use is made based on the offset value. Very large offsets can be encoded by up to four bytes, whereas small offsets are encoded by just eleven bits.

Because a tag longer than one byte could result in cache line crossings and the need to shift large amounts of already-copied data, it was simpler for this implementation to limit the literal length to 60. The compression ratio will not suffer by much from this simplification, as in the worst case this adds an overhead of just $\frac{1}{60}$ times the shortest possible encoding. That is, if the entire stream were comprised of literal substreams, then the longest literal tag (five bytes) could be used. This would result in five tag bytes encoding $2^{32}$ bytes of literal data. With the maximum literal length set at 60, the resultant stream would contain one tag byte for every 60 bytes, which accounts for an expansion by a factor of $\frac{1}{60}$. However, this case is extremely unlikely, and long literals may never even show up, especially with highly compressible data.

Decompressing the compressed stream is rather simple, assuming the decompressed data is read serially. When a literal tag encoding a literal of length $N$ is reached, then the next $N$ bytes simply need to be copied. When a copy tag encoding a copy of length $N$ at offset $O$ is reached, then $N$ bytes simply need to be copied from $O$ bytes behind the tag's location.

For a given input stream, the output of the Snappy compression algorithm is not unique. A data stream is considered compressed if it conforms to the substream format discussed above. There are many ways to compress a single input data stream via the Snappy algo-

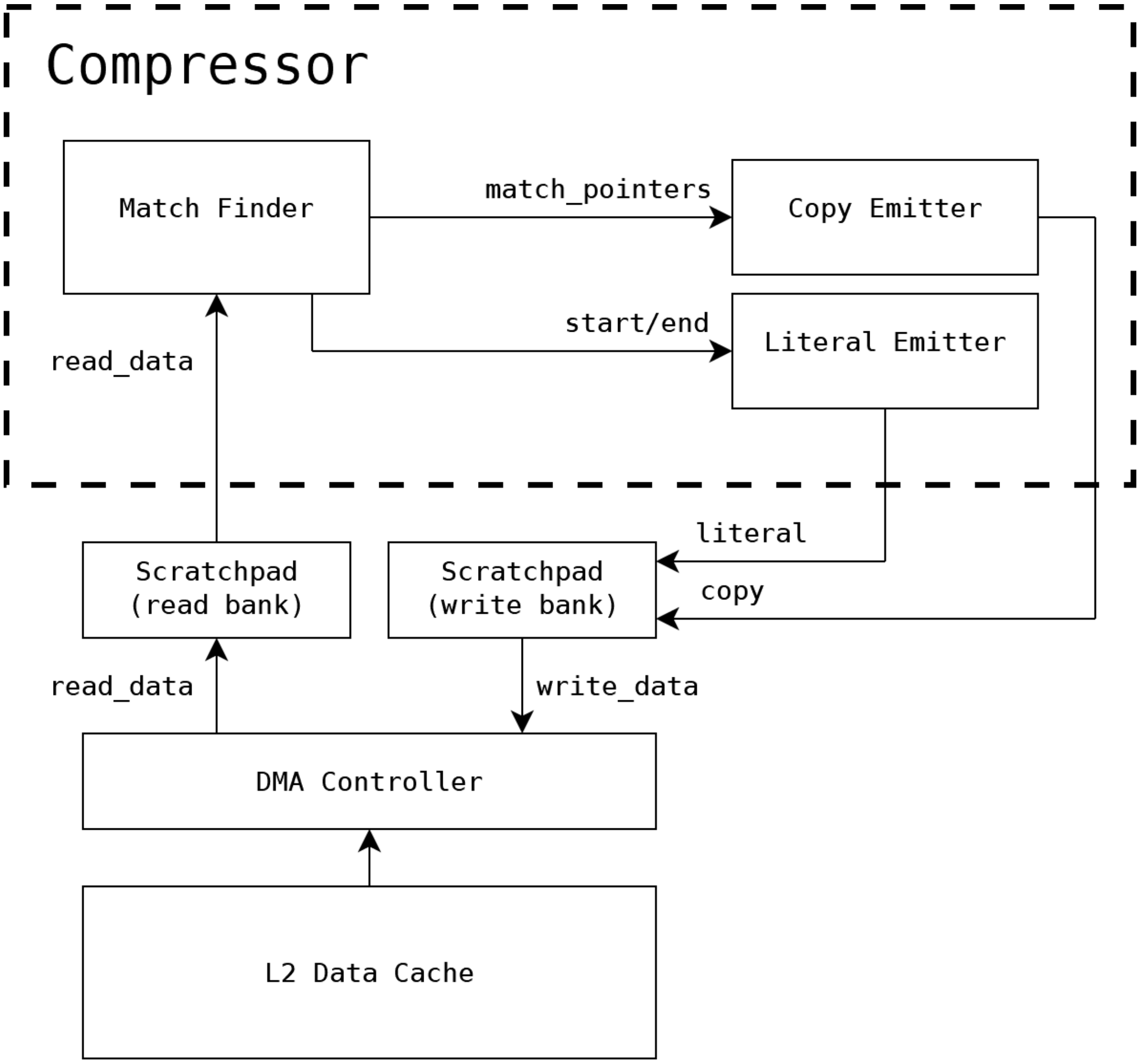


Figure 2.3: **Full system block diagram.** The L2 cache is shown at the bottom connecting to both Scratchpad banks through a DMA controller. The banks are read by the Match Finder and written to by the Copy and Literal Emitters. The box labelled "Compressor" highlights the bounds of the main control logic that governs the compression algorithm flow.

rithm, all of which are valid compressed streams as long as they are decompressible via the same decompression semantics.

### Full System

The diagram in Figure 2.3 illustrates how the basic blocks of the compressor work together. The basic structure consists of a DMA controller, which fills up a Scratchpad read bank with data from memory; a Match Finder, which scans through the data and locates pieces of it that repeat; a Literal Emitter, which writes tag bytes and streamed data from the Scratchpad read bank back into a Scratchpad write bank; and a Copy Emitter, which writes tag bytes and copy information into the Scratchpad write bank.

The DMA controller automatically requests data from the L2 cache in order to keep the read bank of the Scratchpad full so that the compression can run. It also interleaves write requests to the L2 cache so that the Scratchpad write bank can continually be flushed.

The compressor also has more control logic (omitted from the diagram for cleanliness) which coordinates data flow through the system by telling the blocks when to run and when to wait. When the Match Finder detects a match, it is the Copy Emitter's job to determine the length of the matching data. The Match Finder cannot know where to look next before the Copy Emitter determines how long the match was and whether or not more matches can be emitted immediately. Because of this dependency, it is not possible to run both the Match Finder and the Copy Emitter at the same time for a given compression job. Hence, the compressor control logic must handle this information and pause the blocks depending on the state of the system.

## 2.3 Details

### Scratchpad

In order to work with data, the compressor needs to access main system memory. Coprocessors generated with RocketChip have a connection to the CPU L1 data cache by default. However, accessing the L1 cache should be avoided for this application because serially streaming all the input data through the L1 cache would cause far too much traffic and hurt CPU performance. For this reason, main memory is accessed through a Scratchpad connected to the L2 cache over a TileLink [5] bus (RocketChip's main interconnect bus).

The Scratchpad consists of two banks, each of which can be read from or written to independently. The read bank sends read requests to the L2 cache via a DMA controller, and it fills up with the desired input data. Control logic inside the compressor keeps track of which section of the input is currently held in the Scratchpad read bank, and signals the DMA controller to overwrite old data when new data needs to be pulled in from memory. The write bank is filled up by the output side of the compressor. Each time a literal or copy is emitted, it is buffered in the Scratchpad write bank before the DMA controller flushes the bank out to memory.

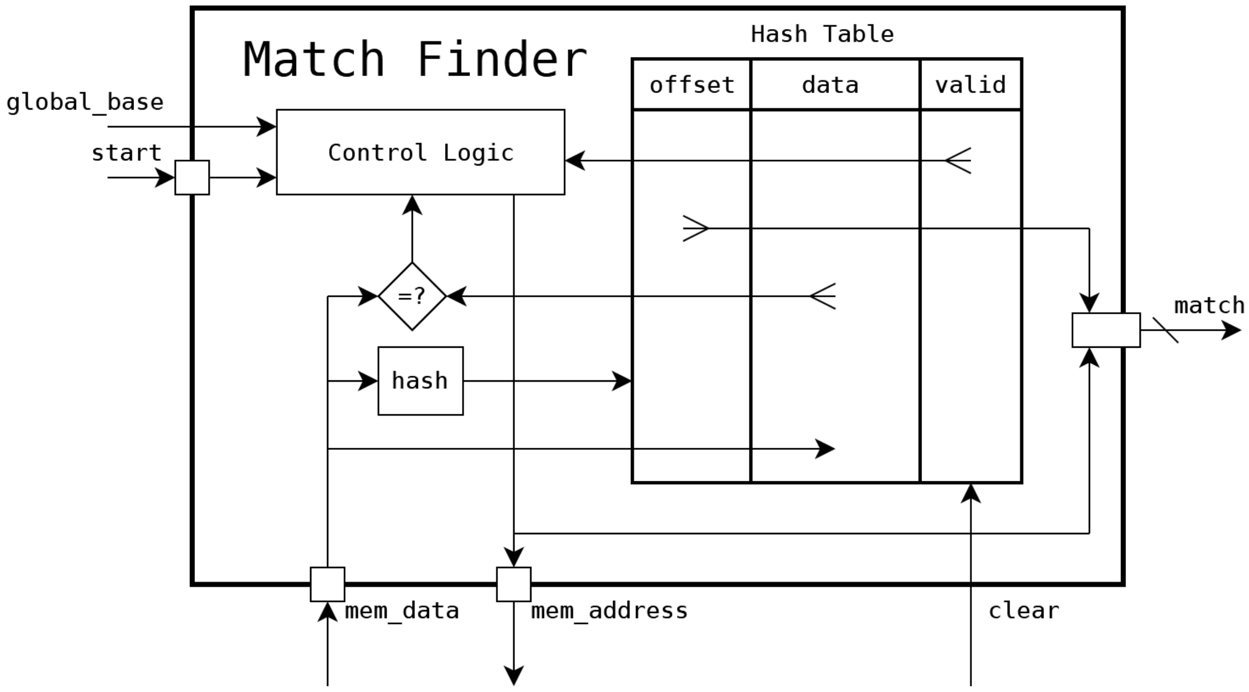


Figure 2.4: **Match Finder block diagram.** The hash table keeps track of recently-seen data. Offsets from the beginning of the stream are stored in the offset column, 32-bit data chunks are stored in the data column, and the valid bits are cleared between compression operations. A match is considered to be found when valid data from the hash table matches the input data. The matching data is the input data, and the matched data is at the offset specified by the appropriate line in the hash table. The control logic for the Match Finder handles data flow in and out of the hash table.

## Match Finder

The Match Finder is responsible for locating a 4-byte match in the input data stream. A starting address is provided to the Match Finder, and it produces pairs of addresses that point to matching 4-byte sequences in the Scratchpad. Each time the input decoupled connection fires, the Match Finder will produce a match some number of cycles later. Once the match length is determined, other pieces of external control logic decide when to start the Match Finder again.

To find a match, the Match Finder scans through the input, taking the hash of each 4-byte chunk and storing its offset from a global base pointer in a hash table. The hash table has three columns. The first column holds the offset (between 0 and the maximum window size) of the data, the second column holds the data itself (32-bit words), and the third column holds a valid bit, which is set the first time data is entered into that hash table row. All valid bits in the hash table are cleared when a new compression operation begins. For each address scanned, the 4-byte chunk stored at that address will hash to some value,

$H$. Row $H$ of the hash table is then populated with the data and appropriate offset, and its valid bit is set. If the valid bit was already set, the control logic checks to see if the previous data stored in the row $H$ matches the new data. If it does, then a match has been found. The new memory address and the offset from row $H$ of the hash table are the pointers to the two matching segments.

In addition to checking the valid bits from the hash table, the control logic also keeps track of a few bits of state such as the current read pointer into the Scratchpad, and whether a match is being sought or not. It also handles all of the ready and valid signals for the decoupled inputs and outputs.

The hash function used here is simply the input bytes multiplied by `0x1e35a7bd` and shifted right by the size of the hash table. This is the same function used in the open-source C++ algorithm. It is meant to minimize collisions.

## Memory Read Aligner

Data in the Scratchpad is stored in 64-bit entries, and the Match Finder needs to access the data in 32-bit byte-aligned chunks. This poses a problem, as it is possible to have reads that span across a 64-bit entry boundary.

It is the Memory Read Aligner's job to allow the Scratchpad to be read with the desired data size at the desired alignment. In the simplest solution, the memory could be read in two cycles: one to read the lower entry and one to read the upper entry. However, adding an extra cycle of unpipelined latency for every read effectively doubles the time it takes to do anything, so this is unacceptable for this application.

To circumvent this limitation, a decoupled interface is adopted for the memory through the Memory Read Aligner, pushing through all non-boundary-crossing reads in one cycle each, and taking two cycles to complete the boundary-crossing ones. For sequential access, this allows for five of every eight reads to complete in one cycle, while three reads take two cycles for a total throughput of $\frac{8}{11}$ reads per cycle.

By caching the most recent 64-bit data and its address in a pair of registers, and using the cached data to form part of the output, the memory can achieve full throughput of one read per cycle for sequential access. Under the worst-case access pattern (all non-sequential, boundary-crossing reads), the throughput drops to $\frac{1}{2}$ reads per cycle. This is unavoidable without a more complex caching scheme, but it is also not a concern due to the fact that the predominant memory access pattern in this application is purely sequential.

Figure 2.5 shows the internals of the Memory Read Aligner. The left side is a decoupled address-data interface, and the right side is a continuous address-data interface. Control logic (not shown) handles the ready and valid signals, and selects which bytes from the aggregated 128-bit data should be passed out of the block.

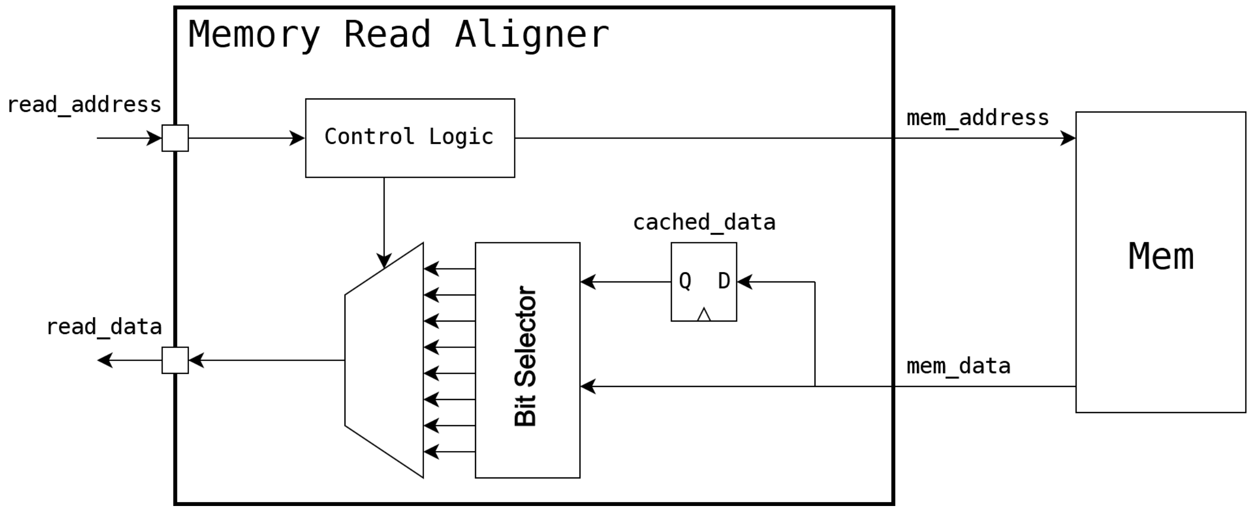


Figure 2.5: **Memory Read Aligner block diagram.** Data comes out of the memory in 64-bit chunks, but needs to be read in byte-aligned 32-bit chunks. The Read Aligner uses simple caching to concatenate the memory data properly and output byte-aligned 32-bit data through a decoupled interface. When sequential addresses are provided, there is no additional delay introduced by the Read Aligner.

## 2.4 Integration

Figure 2.6 is a schematic of how the accelerator would connect to RocketChip. The shared TileLink crossbar manages memory access and consistency such that it is abstracted away from the compressor itself. The Rocket core and the compressor can then both access the same L2 cache backed by the same physical memory.

## 2.5 Dataflow

When a compression job starts, an input and output pointer are specified by the RoCC instruction. The DMA controller keeps track of these values, and produces and consumes requests to and from the L2 cache to facilitate data transfer between the Scratchpad and the cache itself. The Scratchpad's two banks operate in concert to stream data byte-by-byte through the compressor. Both banks are managed as circular buffers, meaning that they evict old data when they run out of space. The first data loaded into each bank will be the first data evicted from that bank. The Scratchpad manager keeps track of which range of data is held in a bank at any given time in order to ensure data is valid when it is read.

Before compression can begin, the read bank must fill up with input data (see Figure 2.7a). As it fills up, its tail pointer advances until it reaches the same line as the head pointer. At this point the read bank is full.

Once the read bank has data in it, the Match Finder can scan the data for matches.

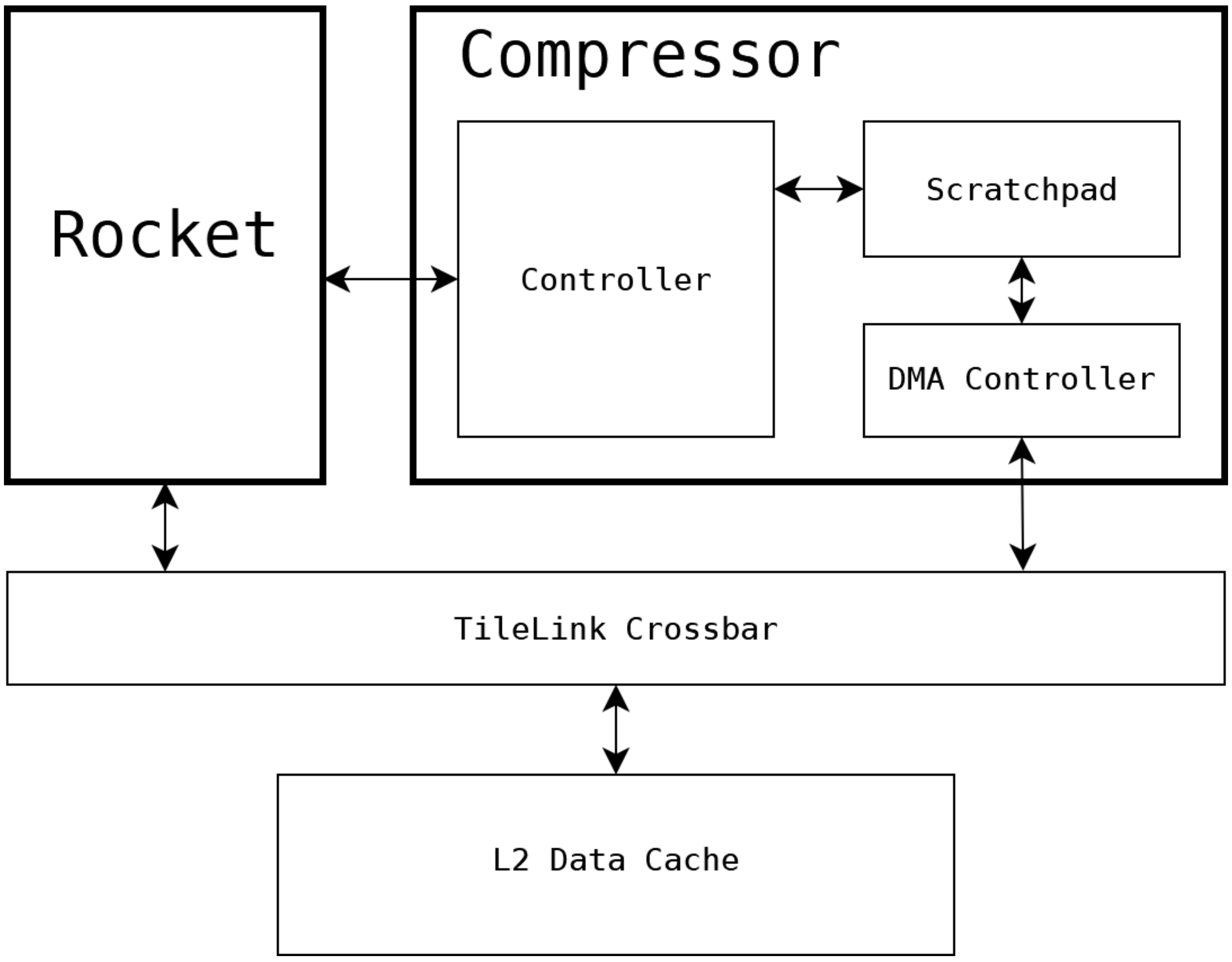
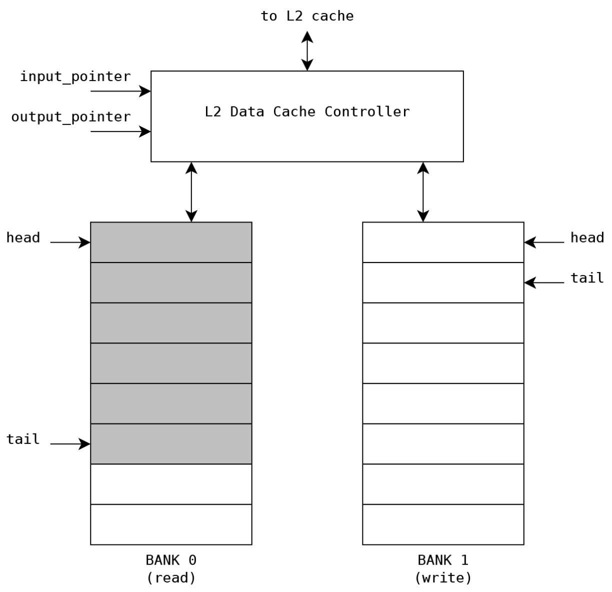
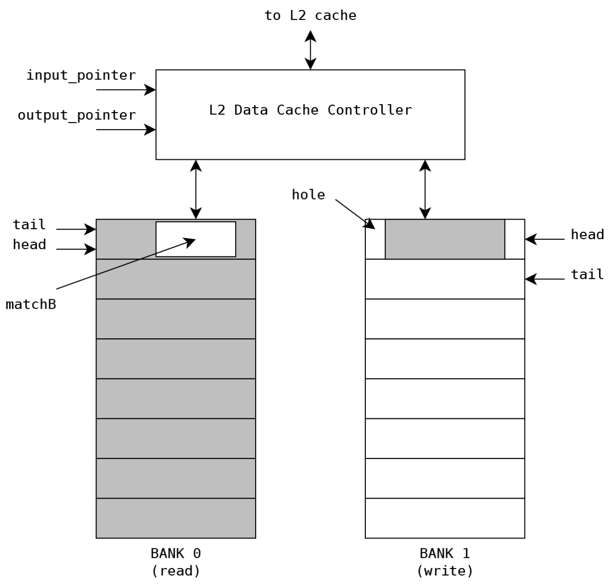


Figure 2.6: **Top-level RocketChip Integration** The Rocket core is attached to the L2 data cache through a TileLink crossbar. The accelerator also connects to the crossbar and accessed the same L2 cache. Memory consistency is handled by the rest of the system, and the accelerator only needs to see the L2 cache in order to operate.

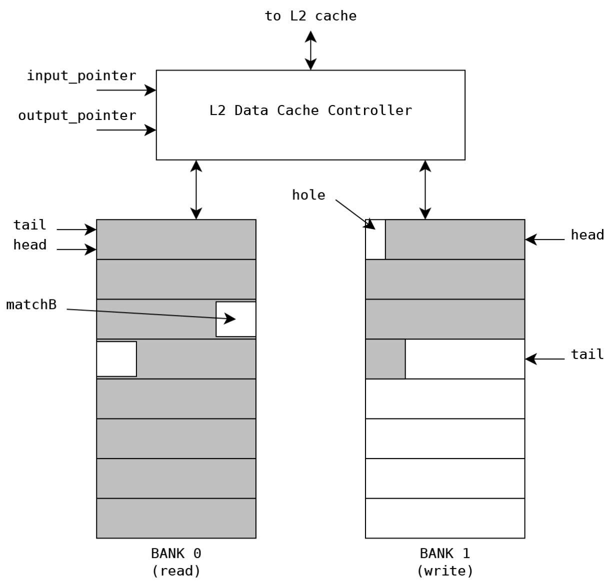


(a) To start the compression, the read bank of the Scratchpad must first fill up with data from the L2 cache, starting at the input pointer. Once the tail reaches the head, the read bank is full of valid data.

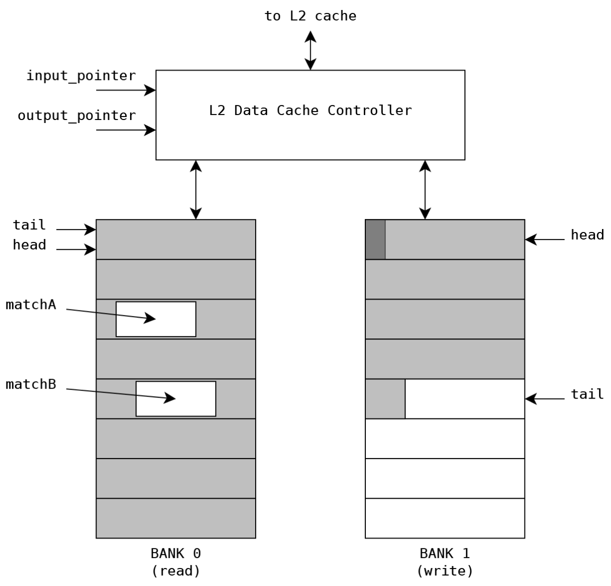


(b) When there is valid data in the read bank, the Match Finder begins scanning the input data for matches, populating the hash table while doing so. As bytes are scanned, they are copied into the write bank of the Scratchpad. A hole is left at the start of the write bank to hold the literal tag. Once a match is found, all data prior to the matching data is considered a literal and the hole is filled in. If 60 bytes are scanned before a match is found, then a literal of length 60 will be emitted by force.

Figure 2.7: **Accelerator dataflow.** These six figures depict the flow of data in and out of the compressor. Input data is stored in the L2 cache at the location indicated by the input pointer, and the output is produced in the L2 cache at the location specified by the output pointer. The size of the Scratchpad banks is configurable, and it determines the sliding window length for the algorithm. A longer window allows for longer copy back-references, and a shorter window requires less hardware.

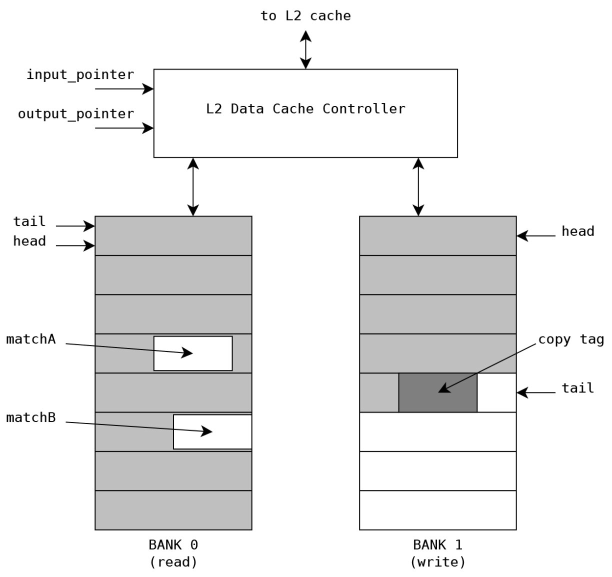


(c) The Match Finder continues to scan through the input data, crossing line boundaries as it scans. The Read Aligner handles this boundary crossing. Meanwhile, unmatched data is still being copied into the write bank.

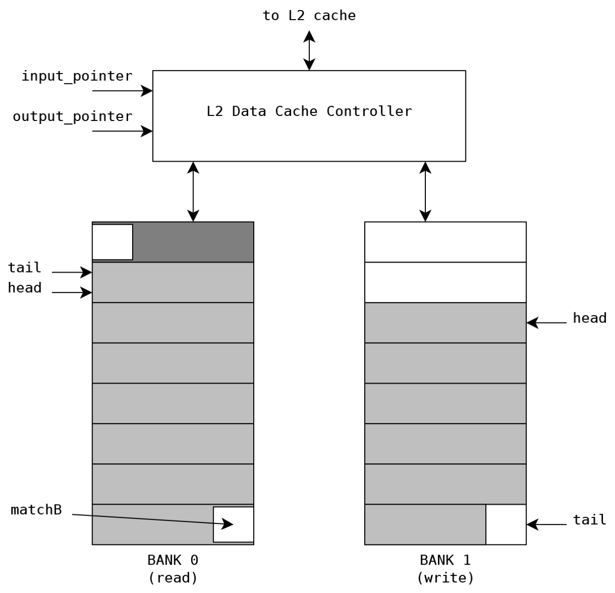


(d) When a match is found, the Copy Emitter will scan through the data to determine the length of the match. Nothing will be put into the write bank until the length is determined, but the match pointers will advance until their data no longer matches. Note that the length of the literal is now known, so the hole is filled in.

Figure 2.7: **Accelerator dataflow.** These six figures depict the flow of data in and out of the compressor. Input data is stored in the L2 cache at the location indicated by the input pointer, and the output is produced in the L2 cache at the location specified by the output pointer. The size of the Scratchpad banks is configurable, and it determines the sliding window length for the algorithm. A longer window allows for longer copy back-references, and a shorter window requires less hardware.



(e) Once the end of the match is detected, the appropriate copy tag will be emitted into the write bank, and the Match Finder will continue looking for more matches. If the subsequent data is unmatched, a literal will be produced. If the subsequent data matches, another match will be produced.



(f) When the Match Finder gets to the end of the Scratchpad read bank, the Scratchpad will evict the oldest line in the bank and replace it with new data from the L2 cache. The maximum look-back distance for matches is therefore limited by the size of the Scratchpad. `matchB` is shown wrapping around back to the beginning of the read bank. Meanwhile, the write bank can begin to flush data out to the L2 cache at the location specified by the output pointer.

Figure 2.7: **Accelerator dataflow.** These six figures depict the flow of data in and out of the compressor. Input data is stored in the L2 cache at the location indicated by the input pointer, and the output is produced in the L2 cache at the location specified by the output pointer. The size of the Scratchpad banks is configurable, and it determines the sliding window length for the algorithm. A longer window allows for longer copy back-references, and a shorter window requires less hardware.

The `matchB` pointer advances one byte every cycle, as depicted in Figure 2.7b, allowing the Match Finder to check a 4-byte word composed of three old bytes and one new byte. That 4-byte word is hashed and the appropriate line of the hash table is updated.

According to the Snappy algorithm, when a match is found (in this case by the Match Finder), all preceding data up until that pointed to by the seeking pointer (in this case `matchB`) is considered a literal and is emitted as such. In order to do this effectively, the data is actually emitted one Scratchpad line at a time as the input is being scanned. This is done to keep intermediate buffer size and extra copy time to a minimum. The consequence of this is that a hole needs to be left in the output at the beginning of the literal. This hole will eventually contain the length tag for the literal, but until the length of the literal is known, the value of the length tag byte cannot be. This is the reason for the imposed 60-byte literal length limit — without this limit, the literal data would have to be aligned in accordance with the literal tag length. This represents a trade-off in which compression ratio is slightly sacrificed in order to provide faster compression runtime. The hole is shown in Figures 2.7b and 2.7c.

Upon finding a match, the literal length becomes known, and so the hole left can be filled in with the appropriate literal tag (Figure 2.7d). At that point, the control logic takes note that all data up until the last byte of the literal is theoretically safe to evict from the read bank of the Scratchpad by updating a pointer. However, this data should be kept in the scratchpad as long as possible in order to allow for a longer look-back window for match detection. The Match Finder cannot look back at data that has already been evicted from the Scratchpad read bank, so it is beneficial to only evict and overwrite the data when space is needed.

The Copy Emitter uses the match information (the `matchB` pointer that was used to scan for matches and the `matchA` pointer that is calculated based on the offset column of the hash table) to determine the length of the match. Both `matchA` and `matchB` advance in lockstep until the end of the match is found, at which point the copy can be emitted into the write bank. Figures 2.7d and 2.7e show the match pointers advancing. A copy is emitted in Figure 2.7e.

Once a copy is emitted, the process can continue as before, with matching data producing more copies and non-matching data producing more literals. As the write bank fills up, its tail pointer advances. Once data has been written into the write bank, it can be flushed out to the L2 cache. It is written to the L2 cache at the location specified by the RoCC instruction output pointer. If the write bank becomes full, or if the input runs out, then the write bank is forcibly flushed. Because there is only one port to the L2 cache, read and write requests cannot be made on the same cycle. For this reason, the accelerator alternates between filling the read bank and flushing the write bank. Figure 2.7f shows new data in the read bank evicting old data, as well as data being flushed from the write bank out to the L2 cache.

When the entire compression job is complete, the hash table will be marked invalid and all the pointers will reset. Any remaining data in the write bank needs to be flushed out to the L2 cache before the output is considered valid for the application to consume.

# Chapter 3

# Results

## 3.1 Software Performance

Google's open-source implementation of the Snappy algorithm [7] was evaluated to provide a performance comparison metric against which to measure. This was done by natively compiling the open-source Snappy library using RISC-V Fedora running on QEMU, and then running code utilizing the library on RocketChip instances on FireSim [10], a cloud-based FPGA-accelerated simulation platform.

Figure 3.1 shows the timing mechanism used to evaluate the compression speed of the software implementation. RISC-V provides an instruction, `rdcycle`, which can be called to obtain the processor cycle number, usually for timing purposes. In this case, it is used before and after the library call to determine the number of cycles it takes to compress a string.

To form a point of comparison between the software and hardware implementations, both were run against the same set of inputs. Three different datasets were generated and tested, and Table 3.1 contains the results of the software implementation tests. The first dataset (called "Random") was completely random characters (integers between 32 and 126 from a random number generator interpreted as characters). This data should be incompressible because there are no repeating patterns, and the compression ratios gathered support this. The second set (called "Real") was taken from a corpus of text generated from random *Magic: The Gathering* cards. This corpus was chosen because it represents highly repetitive English text, and is therefore highly compressible, as the compression ratios also support. The third dataset (called "Repeating") was simply the letter "a" over and over again. This represents the most compressible stream possible — the same byte repeated throughout the input stream.

The timing code from Figure 3.1 was used to measure cycle counts for each compression job. Counts are shown normalized by the input size to demonstrate the implementation's efficiency. Software cycle counts were averaged over 20 runs. Because there is significant overhead in running the program, the listed efficiency is very low for very small files.

For Random data (See Table 3.2), the compression ratio does not depend on the input

```
#include <stdio.h>          // Import printf
#include <string.h>         // Import strings
#include "snappy/snappy.h"  // Import Snappy library

// Calls the RISC-V 'rdcycle' instruction and returns the result
// Compile with -O3 to inline this function in 1 assembly instruction
static inline unsigned long read_cycles(void)
{
    unsigned long cycles;
    asm volatile ("rdcycle %0" : "=r" (cycles));
    return cycles;
}

// Compress a string and print the number of cycles it took
int main(int argc, char* argv[])
{
    // Example string
    const std::string input = "If Tid the thistle sifter lifts his sieve"
        "of sifted thistles, then he'll shift the thistles in his sifting"
        "sieve to live unsifted.";

    // Allocate space
    std::string output;
    unsigned long start, end;

    // Read start time, run compression, read end time
    start = read_cycles();
    snappy::Compress(input.data(), input.size(), &output);
    end = read_cycles();

    // Print result and return
    printf("Ran compression. Took %lu cycles.\n", end - start);
    return 0;
}
```

Figure 3.1: **Cylce timing code.** This code represents an example of running the Snappy software compression with timing. The result will tell how many cycles it took the processor to compress the input data. The `rdcycle` instruction is used to obtain timing information, and it can be run in just one assembly instruction with the `-O3` compiler flag.

| | **Random** | | **Real** | | **Repeating** | |
|---|---|---|---|---|---|---|
| Bytes | Ratio | Cycles/Byte | Ratio | Cycles/Byte | Ratio | Cycles/Byte |
| 10 | 0.8333 | 800.900 | 0.8333 | 805.300 | 0.8333 | 836.500 |
| 20 | 0.9091 | 447.800 | 0.9091 | 446.550 | 3.3333 | 456.950 |
| 50 | 0.9615 | 471.600 | 0.9615 | 469.920 | 8.3333 | 207.980 |
| 100 | 0.9709 | 344.950 | 0.9901 | 459.940 | 11.1111 | 129.630 |
| 200 | 0.9804 | 229.400 | 1.0000 | 349.690 | 13.3333 | 88.185 |
| 500 | 0.9901 | 124.732 | 1.1390 | 349.724 | 17.8571 | 66.194 |
| 1000 | 0.9950 | 75.164 | 1.2438 | 372.609 | 19.2308 | 59.738 |
| 2000 | 0.9975 | 45.526 | 1.3966 | 371.086 | 20.0000 | 55.692 |
| 5000 | 0.9990 | 27.664 | 1.8748 | 313.538 | 20.8333 | 56.786 |
| 10000 | 0.9995 | 19.645 | 2.6724 | 239.555 | 21.0526 | 56.313 |
| 20000 | 0.9997 | 12.147 | 2.4450 | 265.388 | 21.1864 | 53.785 |
| 50000 | 0.9999 | 8.105 | 2.4044 | 274.950 | 21.2675 | 53.071 |

Table 3.1: **Software compression results.** The software compression library was run on a simulated Rocket core using FireSim. For each input size, compression ratio and number of cycles were measured. Cycle counts were obtained via the RISC-V `rdcycle` instruction, and compression ratios were obtained via the library itself. For an input length $N$, the "Random" dataset consists of $N$ random bytes, the "Real" dataset consists of the first $N$ bytes of *MTGCorpus*, and the "Repeating" dataset consists of the letter "a" $N$ times.

| | **Random** | | **Real** | | **Repeating** | |
|---|---|---|---|---|---|---|
| Bytes | Ratio | Cycles/Byte | Ratio | Cycles/Byte | Ratio | Cycles/Byte |
| 10 | 0.7692 | 7.000 | 0.7692 | 7.000 | 2.0000 | 4.600 |
| 20 | 0.8696 | 5.200 | 0.8696 | 5.200 | 3.3333 | 3.050 |
| 50 | 0.9434 | 4.600 | 0.9804 | 4.640 | 8.3333 | 2.420 |
| 100 | 0.9615 | 4.240 | 0.9901 | 4.260 | 11.1111 | 2.390 |
| 200 | 0.9662 | 4.115 | 1.0309 | 4.110 | 13.3333 | 2.125 |
| 500 | 0.9766 | 4.070 | 1.1601 | 3.892 | 17.8571 | 2.058 |
| 1000 | 0.9804 | 4.045 | 1.2361 | 3.769 | 19.2308 | 2.017 |
| 2000 | 0.6810 | 4.034 | 1.2862 | 3.750 | 20.0000 | 2.003 |
| 5000 | 0.9829 | 4.028 | 1.4680 | 3.547 | 20.1613 | 1.994 |
| 10000 | 0.9833 | 4.027 | 1.7343 | 3.296 | 21.0526 | 1.990 |
| 20000 | 0.9834 | 4.025 | 1.6226 | 3.409 | 21.1864 | 1.988 |
| 50000 | 0.9835 | 4.025 | 1.5298 | 3.503 | 21.2675 | 1.988 |

Table 3.2: **Hardware compression results.** The same input was run through the hardware accelerator in simulation. Efficiency is higher by up to 200-fold, but compression ratio is lower by a factor of about 2 for compressible data. The software expands incompressible data rather badly, but the hardware does not pay that penalty.

size. This is because more and more random data stays equally incompressible. However, for Real and Repeating data, the compression ratio grows as the input grows. With these datasets, more data means more opportunities to find matches and emit copy substreams. For Repeating data, however, the ratio and efficiency both tend to level off once the input becomes large enough. After a certain point, the algorithm is simply emitting maximum-length copies again and again, utilizing only one line of the hash table and taking the same amount of time to emit each substream.

Note that for Random data, the efficiency increases with the size of the input. This is because of an optimization in the software implementation which begins to "realize" that the data is incompressible and start skipping further and further along.

## 3.2 Hardware Performance

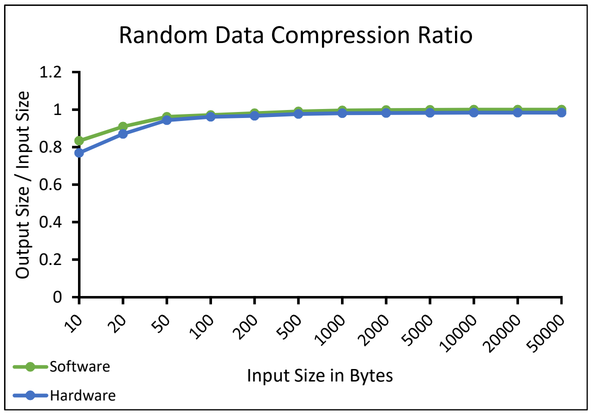
For hardware compression, efficiency is much higher. This matches what is expected because there is far less overhead in running the algorithm on dedicated hardware. Compression ratios suffer slightly, but this could be alleviated with some optimizations (see Chapter 4). The hardware accelerator does not update the hash table when matches are found. This was a non-optimal design decision that was not implemented, but that would improve compression ratios. The literal length was limited to 60 bytes, as discussed in section 2.2. This creates a non-trivial amount of encoding overhead in the output, harming the resultant compression ratio.

Figure 3.2 shows a comparison between the software and hardware implementations. Each graph is plotted on a log-pseudolog scale. It is clear to see from the graphs that the hardware implementation trades some compression ratio for efficiency.

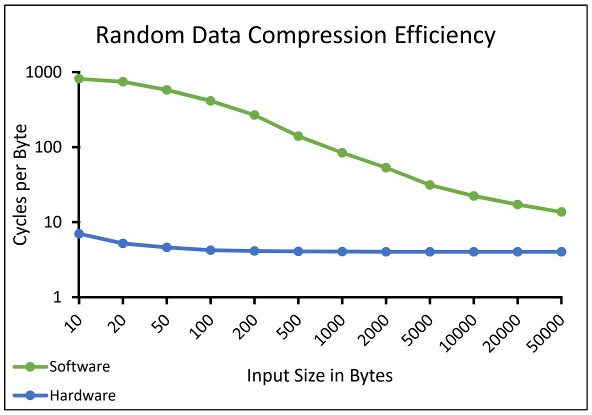
Another factor to consider is that the size of the hash table can be tuned. The software algorithm uses a hash table of 4096 lines. These hardware compression ratio results were taken with a hash table size of 512 lines. The yellow line on the plot in Figure 3.2c shows a run with the hash table size set to 2048 lines. When the hash table was adjusted to 2048 lines instead of 512, it improved the compression ratio for larger files (see Table 3.3.

## 3.3 Synthesis Results

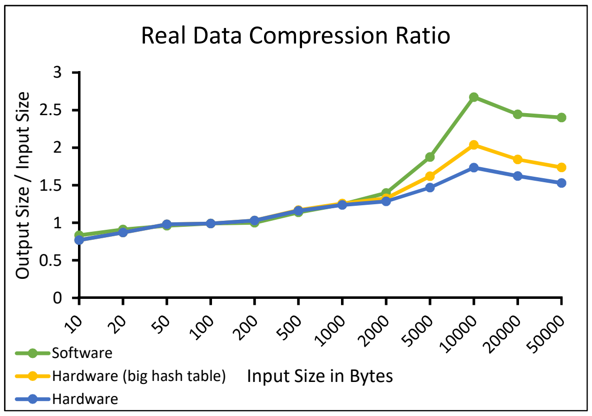
The hardware design was passed through synthesis using Hammer [8] to obtain the critical path. Results show a critical path of 3.5 nanoseconds, which corresponds to an unpipelined clock frequency of 286 MHz. RocketChip is estimated to run close to 1 GHz, so this means that the hardware clock is 3.5 times slower than the software clock. In this case, a factor of 100 improvement in cycle count is equivalent to a factor of about 29 improvement in speed. The most critical section is the hash table read path, which could be improved by pipelining the hash table or changing it from a combinational memory to a synchronous memory.



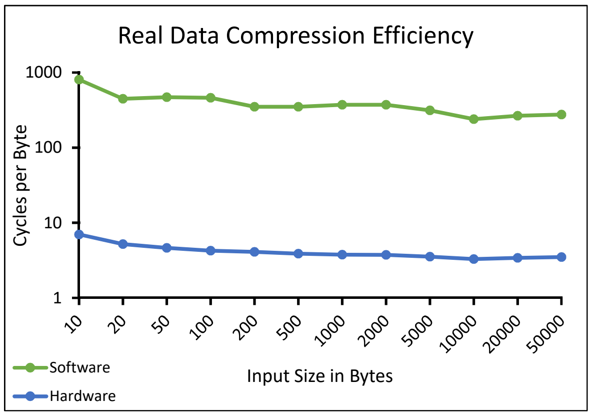
(a)



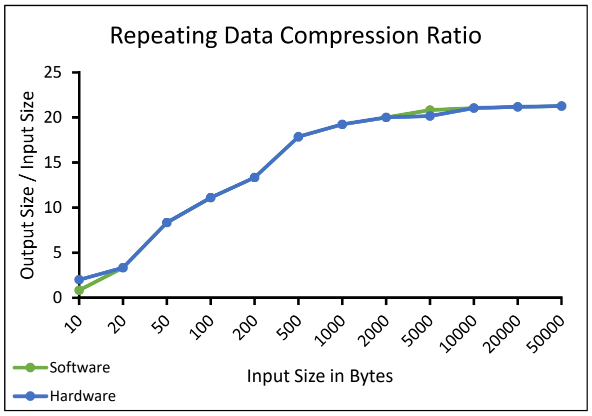
(b)



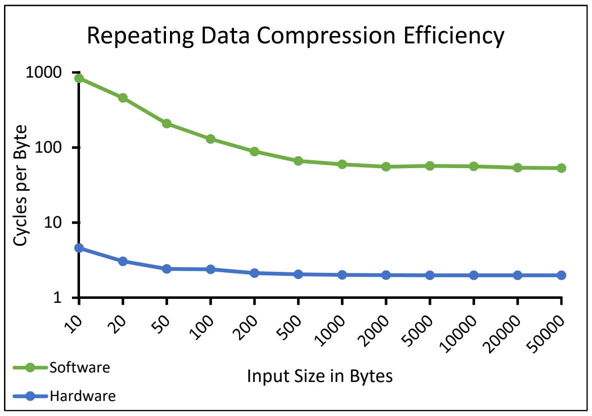
(c)



(d)



(e)



(f)

Figure 3.2: **Compression results.** Plotted are the measurements taken for compression ratio and efficiency (cycles per byte). The green lines denote the software implementation; the blue lines denote the hardware accelerated implementation. Figures 3.2a, 3.2c, and 3.2e compare compression ratios, while Figures 3.2b, 3.2d, and 3.2f compare efficiency.

| | **Real** | |
|---|---|---|
| Bytes | Ratio | Cycles/Byte |
| 10 | 1.0309 | 4.110 |
| 20 | 2.0354 | 3.124 |
| 50 | 0.9804 | 4.640 |
| 100 | 0.9901 | 4.260 |
| 200 | 0.7692 | 7.000 |
| 500 | 1.1682 | 3.886 |
| 1000 | 1.2563 | 3.758 |
| 2000 | 1.3245 | 3.736 |
| 5000 | 1.6202 | 3.436 |
| 10000 | 0.8696 | 5.200 |
| 20000 | 1.8440 | 3.277 |
| 50000 | 1.7374 | 3.376 |

Table 3.3: **Hardware compression with larger hash table.** Here, the hash table size was increased to 2048 lines instead of 512. The compression ratios for real data are the only ones that change significantly, because random data is not compressible anyway and repeating data does not benefit from a large hash table. Efficiency in terms of number of cycles is not impacted.

# Chapter 4

# Conclusion

This work successfully designed and built a compression hardware accelerator capable of compressing data in accordance with the Snappy data compression algorithm. Simulation results show that this accelerator prototype can achieve much greater efficiency than an equivalent software implementation. Although compression ratio performance is lower, there are more optimizations to be made for this prototype.

## 4.1 Future Work

### Hash Table Updates

In its current form, the accelerator does not update the hash table when matches are found. This can lead to data falling out of the look-back range when it shouldn't. This was a design oversight that led to a tangible decrease in compression ratio, but it is fixable with some architectural changes. Namely, the Match Finder would have to allow the main controller access to the hash table, or it would have to have additional ports to send updates to the hash table during copy length detection.

### Literal Length Limit

Limiting the literal length to 60 was a good way to decrease complexity for the prototype, but the compression ratio does suffer because of it. A good way to circumvent the problem of having to shift all the previous output data once the literal length is known would be to add an additional translation layer between the scratchpad write bank and the L2 cache. This would allow for a variable-sized "hole" and thus, literals longer than 60. This possibility was not explored in this work due to its complexity.

## Code Optimization

The Chisel code was not heavily optimized. Functionality was important to establish first, and optimization has not been investigated. For example, when the accelerator first starts, there is a waiting period while the scratchpad completely fills before data begins to be processed. With more careful inspection and tuning, the code can be made more efficient.

## RocketChip Simulation

In this work, simulations of the stand-alone accelerator were run using the Chisel test framework. It would be interesting to simulate the entire Rocket core and use FireSim to run the hardware tests. This would allow for more interesting analysis. For example, a major benefit of the hardware accelerator is that the CPU can do other useful work while the accelerator runs. This concurrency could be exercised in a simulation environment that allows for both hardware and software to run on a Rocket core with the accelerator attached as a co-processor.

## Other Compression Algorithms

Although this prototype was built to comply with the Snappy compression algorithm, other algorithms could be considered. Many sliding-window compression algorithms are similar, and a general framework for compression could be considered in the future. In Chisel, the algorithm used could be structured as a top-level generator parameter, which would allow for versatility of use.

# Bibliography

[1] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, May 2017.

[2] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html`.

[3] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: ACM, 2012, pp. 1216–1225. ISBN: 978-1-4503-1199-1. DOI: `10.1145/2228360.2228584`. URL: `http://doi.acm.org/10.1145/2228360.2228584`.

[4] *Chisel*. `https://chisel.eecs.berkeley.edu/`. Accessed: 2019-5-6.

[5] Henry Cook. *Productive design of extensible on-chip memory hierarchies*. Tech. rep. UCB/EECS-2016-89. EECS Department, University of California, Berkeley, May 2016. URL: `https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-89.pdf`.

[6] *FIRRTL Interpreter*. `https://github.com/freechipsproject/firrtl-interpreter`. Accessed: 2019-5-6.

[7] *Google Snappy*. `https://github.com/google/snappy`. Accessed: 2019-5-6.

[8] *Hammer: Highly Agile Masks Made Effortlessly from RTL*. `https://github.com/ucb-bar/hammer`. Accessed: 2019-8-6.

[9] Adam Izraelevitz et al. "Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations". In: *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD '17. Irvine, California: IEEE Press, 2017, pp. 209–216. URL: `http://dl.acm.org/citation.cfm?id=3199700.3199728`.

[10] Sagar Karandikar et al. "Firesim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: `10.1109/ISCA.2018.00014`. URL: `https://doi.org/10.1109/ISCA.2018.00014`.

[11] David Salomon. *Data Compression: The Complete Reference*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 1846286026.

[12] *Snappy*. `https://google.github.io/snappy/`. Accessed: 2019-4-18.

[13] *Snappy Compression Accelerator on Github*. `https://github.com/nullromo/compression-accelerator`. Accessed: 2019-5-6.

[14] *Treadle*. `https://github.com/freechipsproject/treadle`. Accessed: 2019-5-6.

[15] *VCS*. `https://www.synopsys.com/verification/simulation/vcs.html`. Accessed: 2019-6-6.

[16] *Verilator*. `https://www.veripool.org/wiki/verilator`. Accessed: 2019-6-6.