UNIVERSITÀ
DI **PADOVA**

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Efficient Implementations of Hash Functions in AArch64 Assembly

Relatore
Prof. Silvestri Francesco

Laureando
Gasparotto Mattia
Matricola N. 2101818

ANNO ACCADEMICO 2025-2026

Data di laurea 21/07/2026

To my family, who have always helped me.

Abstract

In the field of big data processing, universal and strongly universal hash functions are essential for handling large amounts of data. This thesis focuses on implementing a set of hash functions as efficiently as possible.

After implementing each function in Rust, a modern high-level language, an inline AArch64 assembly implementation is developed. The results show that, even compared with modern compilers, hand-written assembly can still outperform compiler-generated code.

Through benchmarks, this thesis also analyzes how specific design choices affect the functions' performance. Finally, it determines whether the complexity of writing AArch64 assembly is worth the resulting performance gain.

Contents

Chapter 1

Introduction

Big Data is a term that originated in the late 1990s to describe the process of storing, analyzing, and processing massive datasets. The results of this process are then used for training smarter AI models, predicting the market, or automating smart home environments. Examples of Big Data applications include the 32 petabytes of climate observations by NASA [1], the 150 million sensors collecting data 40 million times per second in the Large Hadron Collider [2] or the decoding of the human genome.

In order to process these massive datasets, functions must be implemented as efficiently as possible. Hash functions are a widely used set of functions in the field of Big Data for quickly accessing and processing data.

1.1 Hash functions

A **hash function** $h : U \to [m]$ maps each value $x \in U$ to a random variable $h(x) \in [m]$ [3], where $[m] = \{0, 1, \ldots, m-1\}$ and $|U| \gg m$. Since the number of possible inputs is larger than the number of possible outputs, hash functions should be designed to minimize collisions. The probability of a **collision**, given two inputs $x, y \in U, x \neq y$, is $P[h(x) = h(y)]$, that is, the probability that two different inputs generate the same output.

A hash function is said to be:

1. *c-approximately universal*, or simply **universal**, if:

$$P[h(x) = h(y)] \leq \frac{c}{m}, c = \mathcal{O}(1)$$

2. **strong universal** if, for any q,$r \in [m]$:

$$P[h(x) = q \wedge h(y) = r] \leq \frac{1}{m^2}$$

A strong universal function is also universal [3].

1.2 Rust programming language

Rust is a programming language created in 2006 by Graydon Hoare during his time at Mozilla. It focuses on performance and memory efficiency while still guaranteeing memory and thread safety. Other important features are: the absence of a garbage collector, support for embedded devices, and highly informative compiler error messages [4].

The use of functions that do not guarantee memory and thread safety, and are thus deemed unsafe, are still allowed in Rust within `unsafe {}` blocks or inside functions marked with the `unsafe` keyword. An example of an unsafe operation is the `asm!()` macro. The `asm!()` macro is used to write assembly code inside a Rust program. Therefore, it is possible to write platform-specific code in order to achieve maximum performance from the processor.

1.2.1 Benchmarking and testing

`Criterion` [5] is a crate (the Rust terminology for a package) that is widely used for evaluating functions' performance. For each function, the measurements are done as follows:

1. **Warmup**: The function is executed for some time (default: 3s) to fill the device cache.
2. **Measurement**: The function is executed repeatedly and execution time is recorded.
3. **Analysis**: Criterion calculates various parameters such as mean, distribution, slope, etc.
4. **Comparison**: The performance of the current run is compared with the last run to determine if the function has improved or not.

The functions were tested using the `proptest` [6] crate. After defining the valid range of inputs and parameters, `proptest` will assert the equivalence between the implementations through hypothesis-like property-based testing and shrinking. With this approach, a wide range of inputs (including edge cases) is used and, in case of failure, shrinking is applied in order to find the minimal input that triggers an error.

1.3 ARM architecture

ARM is a RISC processor architecture developed by ARM Holdings. RISC stands for *Reduced Instruction Set Computer* and is a type of ISA (*Instruction Set Architecture*) that focuses on reducing both the number of available instructions and their complexity.

A study of the x86 architecture revealed that 25% of the instructions account for 95% of the execution time [7]. By removing unnecessary instructions, it is possible to achieve a more effective chip design and better power performance at the cost of code being 30% longer than the CISC (*Complex Instruction Set Computer*) equivalent [7] [8]. Thanks to the simplicity of the ARM ISA, the CPU is able to pipeline consecutive instructions. Over the generations, the pipeline has grown from a 3-stage pipeline in the ARM1 generation (classic *fetch-decode-execute*), to an 8-stage pipeline in the ARM11 generation (where two instructions are fetched at the same time) [8].

Due to their simplicity and energy efficiency, ARM CPUs have become widely adopted in recent years, expanding from the mobile and smart sensor markets into the computer market (with the latest Apple laptops and PCs) and the server market (with Ampere-based ARM CPUs).

The assembly code written inside the `asm!()` macro follows a slightly different syntax. To bind Rust variables inside the assembly code, it is possible to define named placeholders. With these, Rust variables can be used as input or output (or both) for the assembly code. These placeholders can be used like normal registers inside the assembly code, provided that the placeholder's name is enclosed in curly braces. The Rust compiler will then automatically map each placeholder to a physical hardware register.

1.3.1 Raspberry Pi 4

Raspberry Pi is a series of single board computers (SBCs) developed by the Raspberry Pi Foundation. For the development of this thesis, the Raspberry Pi 4 (4 GiB model) will be used.

The Raspberry Pi 4 features a Broadcom BCM2711 SoC, which has four ARM Cortex-A72 ARMv8 processors clocked at 1.5 GHz, with 32 KiB + 48 KiB of data and instruction L1 cache per core, 1 MiB shared L2 cache and 4 GiB LPDDR4-3200 SDRAM.

The operating system used is Raspberry Pi OS 64-bit version, a Debian-based distribution, with the Linux 6.12.70+rpt-rpi-v8 kernel, a fork of the Linux kernel by the Raspberry Pi Foundation optimized for their devices. The installed version of Cargo and Rust is 1.91.0.

ARITHMETIC INSTRUCTIONS	
`ADC{S} rd, rn, rm`	$rd = rn + rm + C$
`ADD{S} rd, rn, op2`	$rd = rn + op2$
`MADD rd, rn, rm, ra`	$rd = ra + rn \times rm$
`MUL rd, rn, rm`	$rd = rn \times rm$
`SBC{S} rd, rn, rm`	$rd = rn - rm - \sim C$
`SUB{S} rd, rn, op2`	$rd = rn - op2$
`UMULH Xd, Xn, Xm`	$Xd = (Xn \times Xm)_{127:64}$
LOGICAL AND MOVE INSTRUCTIONS	
`AND{S} rd, rn, op2`	$rd = rn$ & $op2$
`EXTR rd, rn, rm, #p`	$rd = rn_{p-1:0} : rm_{N-1:p}$
`LSL rd, rn, rm`	$rd = rn \ll rm$
`LSR rd, rn, rm`	$rd = rn \gg rm$
`MOV rd, #i`	$rd = i$
`ORR rd, rn, op2`	$rd = rn \vert op2$
LOAD AND STORE INSTRUCTIONS	
`LDP rt, rt2, [addr]`	$rt2 : rt = [addr]_{2N}$
`LDR rt, [addr]`	$rt = [addr]_N$
BRANCH AND CONDITIONAL INSTRUCTIONS	
`B rel`$_2$`8`	$PC = PC + rel^{\pm}_{27:2} : 0_2$
`CBZ rn, rel`$_2$`1`	if $(rn = 0)$ $PC \mathrel{+}= rel^{\emptyset}_{20:2} : 0_2$
`CMP rd, op2`	$rd - op2$ (S)
`CSEL rd, rm, rn, cc`	if(cc) $rd = rn$; else $rd = rm$
`TST rn, op2`	rd & $op2$ (S)

Table 1.1: Summary of AArch64 assembly instructions used in this thesis.

Chapter 2

Implementation of hash functions

In this chapter, the hash functions presented and analyzed in [3] will be implemented in both Rust and inline AArch64 assembly, with the goal of making them as fast as possible.

The comparison between the various implementations of the same hash function is done as follows: for integer hashing, a vector of 1024 random `u64` integers will be created and `Criterion` will measure the execution time to hash a random integer from the vector. In the case of vector hashing, `Criterion` will measure the execution time to hash a vector of `u64` with lengths increasing on a logarithmic scale.

For integers, the test will be run with different optimization levels:

1. `-O0`: No optimizations, mainly used for debugging.
2. `-O1`: Basic optimizations.
3. `-O2`: Some optimizations.
4. `-O3`: All optimizations.
5. `-O3` and `target-cpu=native`: All optimizations, the Rust compiler will also use platform-specific instructions.

One of the many optimizations that the Rust compiler can perform is **inlining**. By writing the `#[inline]` attribute before the declaration of a function, the Rust compiler will attempt to inline (embed) the called function's body inside the caller to reduce function call overhead.

Another important optimization is **constant folding**. Some hash functions' parameters can be declared as constants. In such cases, Rust can calculate constant expressions at compile time. To benefit from this optimization, the inline assembly implementations will leave the computation of possible constant expressions to Rust.

Since the effectiveness of constant folding depends heavily on how the code is written, all benchmarks are executed with constant parameters wrapped in a `black_box()`. This prevents

the compiler from making assumptions about their values, hence only the worst-case scenario will be considered. Therefore, the code could run faster in some real-world applications.

2.1 Universal hash functions

2.1.1 Multiply-mod-prime

Based on [3, Chapter 2.2] the **universal multiply-mod-prime** hash function $h_{a,b} : [u] \to [m]$ is defined as:

$$(((a \cdot x) + b) \bmod p) \bmod m \tag{2.1}$$

where:

- x is the value to be hashed, $x \in [u]$.
- a is a uniformly random integer in $[1, p-1]$.
- b is a uniformly random integer in $[0, p-1]$.
- p is a prime number greater than the universe size of x.
- m is the range of the hash function.

Since x will be a 64-bit integer, p, a, and b will be stored in `u128` integers. The product of x and a can therefore be a 192-bit integer, which does not fit in any of Rust's primitive types. A solution is to use a third-party crate to handle those numbers:

- `ethnum` [9]: This crate defines the `u256` type, among other types, as a tuple of two `u128` integers.
- `num-bigint` [10]: This crate defines two structs, `BigInt` and `BigUInt`, which store large signed and unsigned integers, respectively, inside a vector.

Of the two listed, `ethnum` is faster because `num-bigint` has significant overhead due to the number being represented as a vector. Since the product is at most 192 bits, the `u256` type from `ethnum` is sufficient.

The straightforward Rust implementation for a 64-bit integer input is:

```
1 use ethnum::u256;
2
3 #[inline]
4 pub fn multiply_mod_prime_basic(x: u64, a: u128, b: u128, m: u64, p: u128) ->
    u64 {
```

```
    let ax = u256::from(a) * u256::from(x);
    let ax_b = ax + u256::from(b);
    let mod_p = (ax_b % p).as_u128();
    let result = mod_p % (m as u128);
    result as u64
}
```

Code 2.1: Basic multiply-mod-prime implementation.

If the prime p is a *Mersenne prime* $2^q - 1$, with $q \in \{89, 107, 127\}$, [3] suggests applying a *strength reduction* by replacing the modulo mod p with a bitwise AND, a shift, an addition, and an `if` condition, which are significantly faster. The resulting function is:

```
use ethnum::u256;

#[inline]
pub fn multiply_mod_prime_mersenne(x: u64, a: u128, b: u128, m: u64, q: u32) ->
    u64 {
    let p: u256 = 1u128.wrapping_shl(q).wrapping_sub(1).into(); // p = 2^q - 1
    let ax = u256::from(a) * u256::from(x);
    let ax_b = ax + u256::from(b);
    // Mersenne reduction
    let mod_p = ((ax_b & p) + (ax_b >> q)).as_u128();
    let mod_p = if mod_p >= p {
        mod_p - p.as_u128()
    } else {
        mod_p
    };
    let result = mod_p % (m as u128);
    result as u64
}
```

Code 2.2: Rust multiply-mod-prime implementation with Mersenne prime.

In the inline AArch64 assembly implementation, the product of x and a is stored in three 64-bit registers. As mentioned at the beginning of the chapter, the computation of p is left to the Rust compiler, as it can be declared as constant. Even the computation of mod m is left to Rust since there is no AArch64 assembly instruction to compute the modulo of a 128-bit integer.

With the exception of performing the product manually, the inline AArch64 assembly implementation is a straightforward translation of `multiply_mod_prime_mersenne`:

```
#[inline]
```

```
pub fn multiply_mod_prime_1(x: u64, a: u128, b: u128, m: u64, q: u32) -> u64 {
    let p: u128 = 1u128.wrapping_shl(q).wrapping_sub(1); // p = 2^q - 1
    let a_lo = a as u64;
    let a_hi = (a >> 64) as u64;
    let b_lo = b as u64;
    let b_hi = (b >> 64) as u64;
    let p_lo = p as u64;
    let p_hi = (p >> 64) as u64;
    let result_low: u64;
    let result_high: u64;
    unsafe {
        asm!(
        "MUL {yp_0}, {a_hi}, {x}", // yp_0 used for carry
        "UMULH {y_2}, {a_hi}, {x}",
        "UMULH {y_1}, {a_lo}, {x}",
        "MUL {y_0}, {a_lo}, {x}", // first 64-bit of y = a * x
        "ADDS {y_1}, {y_1}, {yp_0}", // second 64-bit of y = a * x
        "ADC {y_2}, {y_2}, xzr", // third 64-bit of y = a * x
        "ADDS {y_0}, {y_0}, {b_lo}", // first 64-bit of y += b
        "ADCS {y_1}, {y_1}, {b_hi}", // second 64-bit of y += b
        "ADC {y_2}, {y_2}, xzr", // third 64-bit of y += b

        "MOV {yp_0}, {y_0}", // first 64-bit of y&p
        "AND {yp_1}, {y_1}, {p_hi}", // second 64-bit of y&p (third is all 0s)
        "LSR {y_0}, {y_1}, {shift1}",
        "LSL {y_1}, {y_2}, {shift2}",
        "ORR {y_0}, {y_0}, {y_1}", // first 64-bit of y>>q
        "LSR {y_1}, {y_2}, {shift1}", // second 64-bit of y>>q
        "ADDS {y_0}, {yp_0}, {y_0}", // first 64-bit of y = (y&p)+(y>>q)
        "ADC {y_1}, {yp_1}, {y_1}", // second 64-bit of y = (y&p)+(y>>q)
        "CMP {y_1}, {p_hi}", // if the second part of y is > p
        "BHI 0f", // Jump to 0 forward
        "BLO 1f", // if it's lower skip to label 1
        "CMP {y_0}, {p_lo}", // else, if the first part of y is < p
        "BLO 1f", // Jump to 1 forward
        "0:", // Label 0
        "SUBS {y_0}, {y_0}, {p_lo}", // first 64-bit of y-p
        "SBC {y_1}, {y_1}, {p_hi}", // second 64-bit of y-p
        "1:", // Label 1
```

```
41
42          y_0 = out(reg) result_low, y_1 = out(reg) result_high,
43          y_2 = out(reg) _,
44          x = in(reg) x,
45          shift1 = in(reg) q as u64 - 64, shift2 = in(reg) 128 - q as u64,
46          a_lo = in(reg) a_lo, a_hi = in(reg) a_hi,
47          b_lo = in(reg) b_lo, b_hi = in(reg) b_hi,
48          p_lo = in(reg) p_lo, p_hi = in(reg) p_hi,
49          yp_0 = out(reg) _, yp_1 = out(reg) _,
50          options(pure, nomem, nostack)
51          )
52      }
53      let result = ((result_high as u128) << 64) | (result_low as u128);
54      (result % (m as u128)) as u64
55  }
```

Code 2.3: Inline AArch64 assembly first multiply-mod-prime implementation.

The function can be further improved by applying another strength reduction to mod m. If m is a power of two, such that $m = 2^{m_{bits}}$ with $m_{bits} \in \{1, \ldots, 64\}$, the mod m operation is equal to a bitwise AND between the number and $m - 1$. Additionally, by choosing a constant value for q, we can evaluate the first 64 bits of $a \cdot x + b \gg q$ with a single instruction: `EXTR { y_0}, {y_2}, {y_1}, {shift}`, whereas it was previously done in 5 instructions.

According to [11], the multiply-mod-prime hash function presented in Equation 2.1 generates fewer collisions when the ratio $\frac{a \cdot x + b}{p}$ is larger. Therefore, we chose the smallest suitable Mersenne prime, $q = 89$, defined in the code as `MIN_Q_U128`. The implementation is as follows:

```
1   #[inline]
2   pub fn multiply_mod_prime_2(x: u64, a: u128, b: u128, m_bits: u32) -> u64 {
3       let a_lo = a as u64;
4       let a_hi = (a >> 64) as u64;
5       let b_lo = b as u64;
6       let b_hi = (b >> 64) as u64;
7       let m_mask: u64 = if m_bits == 64 {
8           u64::MAX
9       } else {
10          (1u64 << m_bits) - 1
11      };
12      let result: u64;
13      unsafe {
14          asm!(
```

```
        "MUL {yp_0}, {a_hi}, {x}", // yp_0 used for carry
        "UMULH {y_2}, {a_hi}, {x}",
        "UMULH {y_1}, {a_lo}, {x}",
        "MUL {y_0}, {a_lo}, {x}", // first 64-bit of y = a * x
        "ADDS {y_1}, {y_1}, {yp_0}", // second 64-bit of y = a * x
        "ADC {y_2}, {y_2}, xzr", // third 64-bit of y = a * x
        "ADDS {y_0}, {y_0}, {b_lo}", // first 64-bit of y += b
        "ADCS {y_1}, {y_1}, {b_hi}", // second 64-bit of y += b
        "ADC {y_2}, {y_2}, xzr", // third 64-bit of y += b

        "MOV {yp_0}, {y_0}", // first 64-bit of y&p
        "AND {yp_1}, {y_1}, {p_hi}", // second 64-bit of y&p (third is all 0s)
        "EXTR {y_0}, {y_2}, {y_1}, {shift}", // first 64-bit of y>>q
        "LSR {y_1}, {y_2}, {shift}", // second 64-bit of y>>q
        "ADDS {y_0}, {yp_0}, {y_0}", // first 64-bit of y = (y&p)+(y>>q)
        "ADC {y_1}, {yp_1}, {y_1}", // second 64-bit of y = (y&p)+(y>>q)
        "CMP {y_1}, {p_hi}", // if the second part of y is > p
        "BHI 0f", // Jump to 0 forward
        "BLO 1f", // if it's lower skip to label 1
        "CMP {y_0}, {p_lo}", // else, if the first part of y is < p
        "BLO 1f", // Jump to 1 forward
        "0:", // Label 0
        "SUB {y_0}, {y_0}, {p_lo}", // first 64-bit of y-p

        "1:", // Label 1
        "AND {y_0}, {y_0}, {m_mask}", // y = y mod m

        y_0 = out(reg) result, y_1 = out(reg) _,
        y_2 = out(reg) _,
        x = in(reg) x, m_mask = in(reg) m_mask,
        shift = const MIN_Q_U128 - 64,
        a_lo = in(reg) a_lo, a_hi = in(reg) a_hi,
        b_lo = in(reg) b_lo, b_hi = in(reg) b_hi,
        p_lo = const P_LO, p_hi = in(reg) P_HI,
        yp_0 = out(reg) _, yp_1 = out(reg) _,
        options(pure, nomem, nostack)
        )
    }
    result
```

```
}
```

Code 2.4: Second inline AArch64 assembly multiply-mod-prime implementation.

The next implementation is the *branchless* version of the previous one. This design choice was made because a mispredicted branch can cause a pipeline flush, costing several CPU cycles. Instead of using a conditional jump, this version saves the value of $y - p$ in a temporary register and then, by using the conditional operations `CCMP` and `CSEL`, determines whether y needs to be updated. The previous implementation is modified as follows:

```
#[inline]
pub fn multiply_mod_prime_3(x: u64, a: u128, b: u128, m_bits: u32) -> u64 {
    // ...

    unsafe {
        asm!(
        // ...

        "ADDS {yp_0}, {y_0}, #1", // potential y_0
        "SUBS xzr, {y_0}, {p_lo}",
        "SBC xzr, {y_1}, {p_hi}",
        "CSEL {y_0}, {yp_0}, {y_0}, HS", // update if y - p >= 0

        "AND {y_0}, {y_0}, {m_mask}", // y = y mod m

        y_0 = out(reg) result, y_1 = out(reg) _, y_2 = out(reg) _,
        x = in(reg) x, m_mask = in(reg) m_mask,
        shift = const MIN_Q_U128 - 64,
        a_lo = in(reg) a_lo, a_hi = in(reg) a_hi,
        b_lo = in(reg) b_lo, b_hi = in(reg) b_hi,
        p_lo = const P_LO, p_hi = in(reg) P_HI,
        yp_0 = out(reg) _, yp_1 = out(reg) _,
        options(pure, nomem, nostack)
        )
    }
    result
}
```

Code 2.5: Edits for a branchless inline AArch64 assembly multiply-mod-prime implementation.

Figure 2.1 was generated using the following parameters: $a = 0x1\,A34F\,98C1\,2B5E\,71D0\,E4A3$, $b = 0xC51A\,2938\,B47D\,6E5F\,1029$, $m_{bits} = 20$ ($m = 1\,048\,576$), and $q = 89$ ($p = 618\,970\,019\,642\,690\,137\,449\,562\,111$). The graph shows

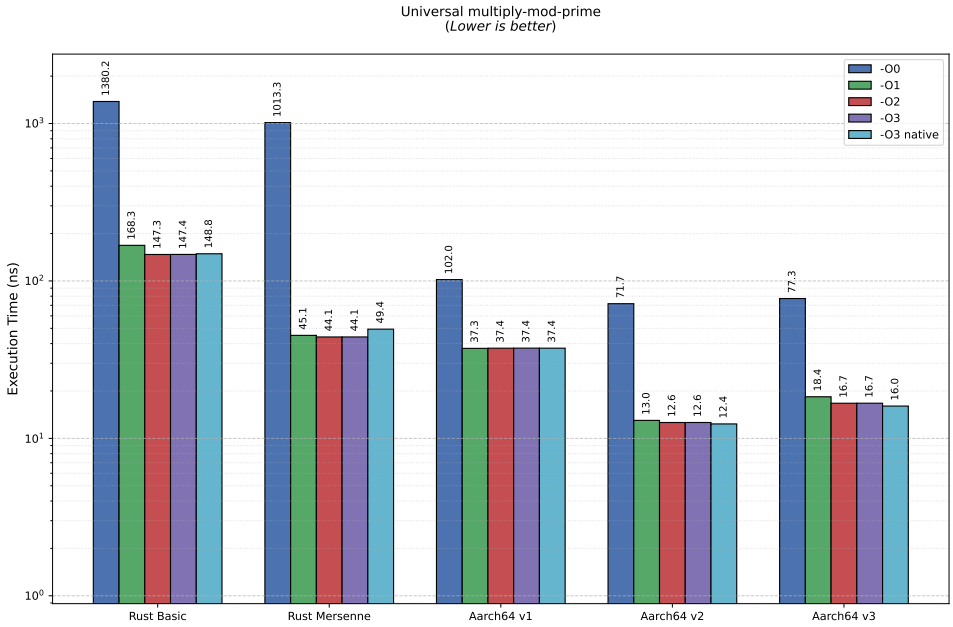

Figure 2.1: Multiply-mod-prime implementations' performance with different optimization levels.

that, with compiler optimizations enabled, the second inline AArch64 assembly implementation triples the throughput of the Rust Mersenne implementation and has ten times the throughput of the Rust basic implementation.

The significant gap between `-O0` and `-O1`, observed across all implementations for every hash function in this thesis, is primarily caused by debuggability. In `-O0`, every variable is written to and read from the stack in memory. This allows debuggers (like GDB) to inspect variable values at any point. Instead, from `-O1` to `-O3`, optimization is prioritized over debuggability, so hardware registers are used.

Although the last two inline AArch64 assembly implementations seem quite restrictive since they require m to be a power of two, it is possible to apply the method described in [3, Chapter 4] by choosing $2^{m_{bits}} > m$ to get a more general implementation.

2.1.2 Multiply-shift

Based on [3, Chapter 2.3], the **universal multiply-shift** hash function is $h_a : [2^w] \to [2^l]$ and is defined as:

$$h_a(x) = \lfloor (a \cdot x \bmod 2^w)/2^{w-l} \rfloor \tag{2.2}$$

where:

- x is a w-bit integer to be hashed.

- a is a uniformly random odd w-bit integer.
- l is the number of bits needed to represent the output space.

Compared to the multiply-mod-prime hash function in Chapter 2.1.1, the universal multiply-shift hash function is significantly faster as it only has elementary operations that are natively supported by modern CPUs.

Both the Rust and inline AArch64 assembly implementations are straightforward. The Rust implementation for a 64-bit integer input and an l-bit output is as follows:

```
#[inline]
pub fn multiply_shift(x: u64, l: u32, a: u64) -> u64 {
    a.wrapping_mul(x).wrapping_shr(64 - l)
}
```

Code 2.6: Rust multiply-shift implementation.

The AArch64 assembly implementation for a 64-bit integer input and an l-bit output is as follows:

```
#[inline]
pub fn multiply_shift(x: u64, l: u32, a: u64) -> u64 {
    let result: u64;
    unsafe {
        asm!(
        "MUL {res}, {a}, {x}", // res = a * x
        "LSR {res}, {res}, {s}", // res = res >> (64 - l)

        x = in(reg) x, a = in(reg) a, s = in(reg) 64-l as u64,
        res = inout(reg) 0u64 => result,
        options(pure, nomem, nostack, preserves_flags)
        );
    }
    result
}
```

Code 2.7: Inline AArch64 assembly multiply-shift implementation.

Figure 2.2 was generated using the following parameters: $a = 0x9E3779B97F4A7C15$ and $l = 20$. The graph shows that the two implementations have basically the same throughput, meaning that Rust code has no overhead.

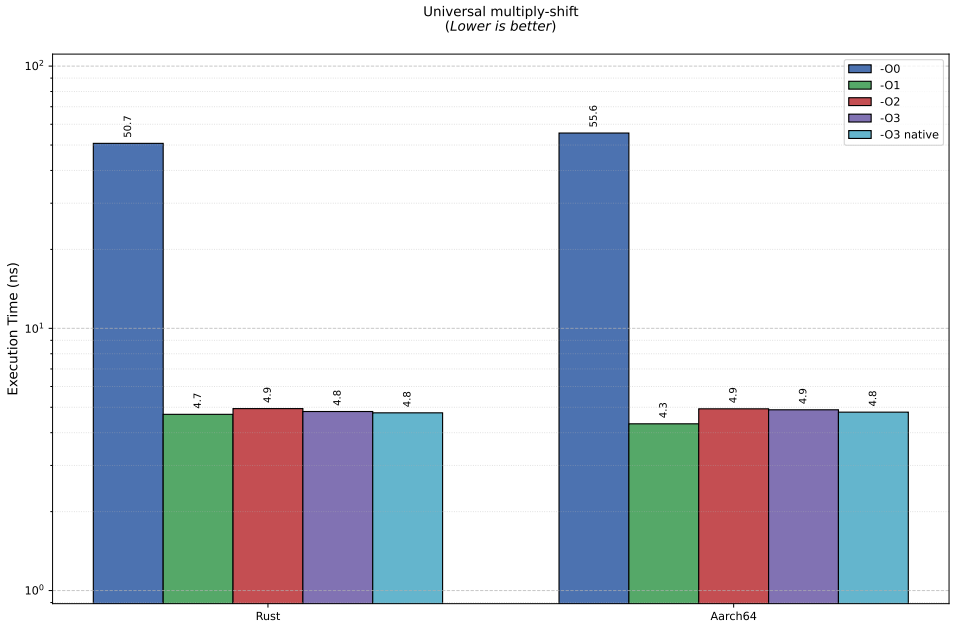

Figure 2.2: Multiply-shift implementations' performance with different optimization levels.

2.2 Strong universal hash functions

2.2.1 Multiply-mod-prime

Based on [3, Chapter 3.2], the **strong universal multiply-mod-prime** hash function is $h_{a,b}(x) : [p] \to [p]$ and is defined as:

$$h_{a,b}(x) = (ax + b) \bmod p \tag{2.3}$$

where:

- x is the value to be hashed, $x \in [p]$.
- a is a uniformly random integer in $[1, p-1]$.
- b is a uniformly random integer in $[0, p-1]$.
- p is a prime number.

As in the universal multiply-mod-prime implementations in Chapter 2.1.1, the prime number p can be either a generic prime or a Mersenne prime $p = 2^q - 1$. To be able to hash all 64-bit integers with the Mersenne implementation, the lowest available q would be 89. However, in this case, the product of x and a would no longer fit in a `u128` variable but would require a `u256` from the `ethnum` crate, which is considerably slower. Therefore, to maintain high performance, we keep p within a `u64` variable. Since the largest q for a 64-bit Mersenne prime is 61, the maximum input that can be hashed is $2^{61} - 2$.

The basic and Mersenne implementations of a strong universal multiply-mod-prime hash function that hashes most 64-bit integers are:

```
#[inline]
pub fn multiply_mod_prime_basic(x: u64, a: u64, b: u64, p: u64) -> u64 {
    let ax = (a as u128) * (x as u128);
    let ax_b = ax + (b as u128);
    (ax_b % (p as u128)) as u64
}
```

Code 2.8: Basic Rust strong universal multiply-mod-prime implementation.

```
#[inline]
pub fn multiply_mod_prime_mersenne(x: u64, a: u64, b: u64, q: u32) -> u64 {
    let p = 1u128.wrapping_shl(q).wrapping_sub(1); // p = 2^q - 1
    let ax = (a as u128) * (x as u128);
    let ax_b = ax + (b as u128);
    // Mersenne reduction
    let mut mod_p = (ax_b & p) + (ax_b >> (q as u128));
    if mod_p >= p {
        mod_p -= p;
    }
    mod_p as u64
}
```

Code 2.9: Mersenne Rust strong universal multiply-mod-prime implementation.

The implementation in inline AArch64 assembly is the straightforward translation of `multiply_mod_prime_mersenne` into AArch64 assembly instructions. As in Chapter 2.1.1, a branchless implementation is also possible:

```
#[inline]
pub fn multiply_mod_prime_1(x: u64, a: u64, b: u64, q: u32) -> u64 {
    let p = 1u64.wrapping_shl(q).wrapping_sub(1); // p = 2^q - 1
    let result: u64;
    unsafe {
        asm!(
        "MUL {y_lo}, {a}, {x}", // LSB 64-bit of y = a*x
        "UMULH {y_hi}, {a}, {x}", // MSB 64-bit of y = a*x
        "ADDS {y_lo}, {y_lo}, {b}", // LSB 64-bit of y += b
        "ADC {y_hi}, {y_hi}, xzr", // MSB 64-bit of y += b

```

```
12          "AND {yp}, {y_lo}, {p}", // LSB of y&p (MSB not necessary)
13          "LSR {yq_lo}, {y_lo}, {q}",
14          "LSL {yq_hi}, {y_hi}, {shift}",
15          "ORR {yq_lo}, {yq_hi}, {yq_lo}", // LSB of y>>q
16          "LSR {yq_hi}, {y_hi}, {q}", // MSB of y>>q
17          "ADD {y_lo}, {yp}, {yq_lo}", // LSB of y = (y&p)+(y>>q)
18          "CMP {y_lo}, {p}", // if the LSB of y is >= p
19          "BHS 0f", // Jump to 0 forward
20          "B 1f", // Skip to label 1 if everything is false
21
22          "0:", // Label 0
23          "SUB {y_lo}, {y_lo}, {p}", // LSB of y-p
24
25          "1:", // Label 1
26
27          y_lo = inout(reg) 0u64 => result, y_hi = out(reg) _,
28          x = in(reg) x, q = in(reg) q as u64,
29          a = in(reg) a, b = in(reg) b,
30          p = in(reg) p, shift = in(reg) 64-q as u64,
31          yp = out(reg) _, yq_lo = out(reg) _, yq_hi = out(reg) _,
32          options(pure, nomem, nostack)
33          )
34      }
35      result
36  }
```

Code 2.10: Inline AArch64 assembly strong universal multiply-mod-prime implementation.

```
#[inline]
pub fn multiply_mod_prime_2(x: u64, a: u64, b: u64, q: u32) -> u64 {
    // ...

    unsafe {
        asm!(
        // ...

        "SUBS {tmp_lo}, {y_lo}, {p}", // LSB of y if y>=p is true
        "CSEL {y_lo}, {tmp_lo}, {y_lo}, hs", // update if needed

        y_lo = inout(reg) 0u64 => result, y_hi = out(reg) _,
        x = in(reg) x, q = in(reg) q as u64,
        a = in(reg) a, b = in(reg) b,
        p = in(reg) p, shift = in(reg) 64-q as u64,
        yp = out(reg) _, yq_lo = out(reg) _, yq_hi = out(reg) _,
        tmp_lo = out(reg) _,
        options(pure, nomem, nostack)
        )
    }
    result
}
```

Code 2.11: Edits for a branchless inline AArch64 assembly strong universal multiply-mod-prime implementation.

Figure 2.3 was generated using the following parameters: $a = 0x9E37\,79B9\,7F4A\,7C15$, $b = 0x27BB\,2EE6\,87B0\,B0FD$, and $q = 61$ (hence $p = 2\,305\,843\,009\,213\,693\,951$).

Whilst in Chapter 2.1.1 the performance improvement from using a Mersenne prime was about 100ns (Figure 2.1), in the strong universal multiply-mod-prime implementation the improvement was approximately 40ns. If 128-bit integers were used for this implementation, the performance improvement would be around the same. Unlike in Chapter 2.1.1, the branchless implementation is 2ns faster than the first inline AArch64 assembly implementation.

2.2.2 Multiply-shift

Based on [3, Chapter 3.3], the **strong universal multiply-shift** hash function $h_{a,b} : [2^w] \to [2^l]$ is defined as:

$$h_{a,b}(x) = (ax + b)[\bar{w} - l, \bar{w}) \tag{2.4}$$

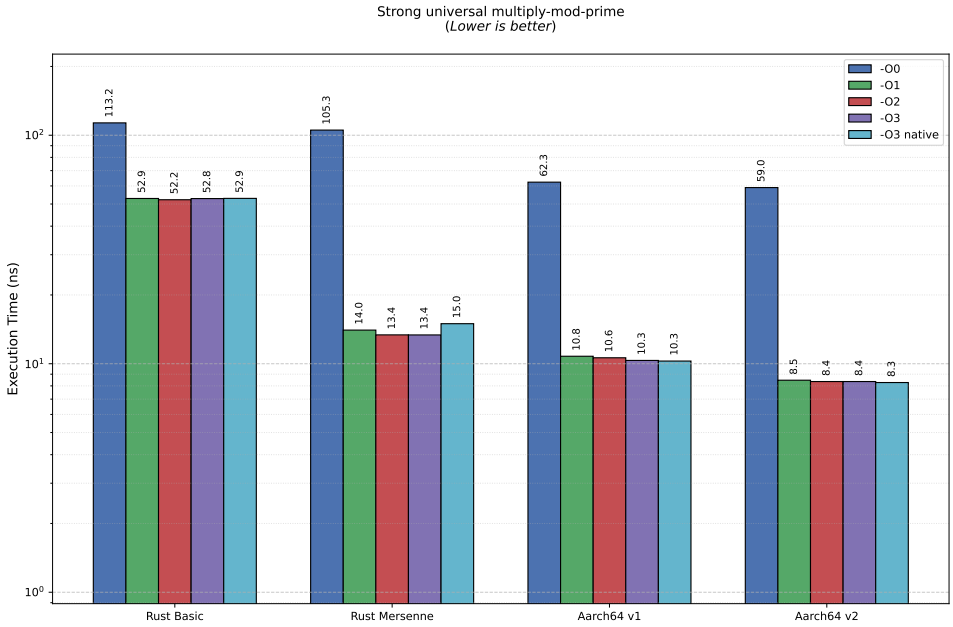

Figure 2.3: Strong universal multiply-mod-prime implementation's performance with different optimization levels.

where:

- x is a w-bit integer to be hashed.
- a and b are a pair of uniformly random $\bar{w}$-bit integers.
- l is the number of bits needed to represent the output universe.

For the function to be strong universal, it is required that $\bar{w} \geq w + l - 1$. Similar to the universal multiply-shift in Chapter 2.1.2, by setting $\bar{w} = 128$ we can exploit the automatic discarding of overflow during multiplication and keep the result of ax within a `u128` variable.

The Rust implementation is as follows:

```
#[inline]
pub fn multiply_shift(x: u64, l: u32, a: u128, b: u128) -> u64 {
    a.wrapping_mul(x as u128).wrapping_add(b).wrapping_shr(128 - l) as u64
}
```

Code 2.12: Basic strong universal multiply-shift implementation.

The inline AArch64 assembly implementation follows directly from the Rust implementation as the instructions used are elementary:

```
#[inline]
pub fn multiply_shift(x: u64, l: u32, a: u128, b: u128) -> u64 {
```

```
    let a_lo = a as u64;
    let a_hi = (a >> 64) as u64;
    let b_lo = b as u64;
    let b_hi = (b >> 64) as u64;
    let result: u64;
    unsafe {
        asm!(
        "MUL {y_lo}, {a_lo}, {x}", // LSB of y = a*x
        "UMULH {y_hi}, {a_lo}, {x}",
        "MADD {y_hi}, {a_hi}, {x}, {y_hi}", // MSB of y = a*x
        "ADDS {y_lo}, {y_lo}, {b_lo}", // LSB of y += b
        "ADC {y_hi}, {y_hi}, {b_hi}", // MSB of y += b
        "LSR {y_lo}, {y_hi}, {shift}", // y >>= (64 - l)

        y_lo = out(reg) result, y_hi = out(reg) _,
        x = in(reg) x, shift = in(reg) 128 - l as u64,
        a_lo = in(reg) a_lo, a_hi = in(reg) a_hi,
        b_lo = in(reg) b_lo, b_hi = in(reg) b_hi,
        options(pure, nomem, nostack)
        );
    }
    result
}
```

Code 2.13: Strong universal multiply-shift inline AArch64 assembly implementation.

Figure 2.4 was generated using the following parameters: $a = 0x1\,A34F\,98C1\,2B5E\,71D0\,E4A3$, $b = 0xC51A\,2938\,B47D\,6E5F\,1029$, and $l = 20$.

2.2.3 Vector multiply-shift

Based on [3, Chapter 3.4], the **strong universal vector multiply-shift** hash function $h_{a_0,\ldots,a_{d-1},b} : [2^w]^d \to [2^l]$ is defined as:

$$h_{a_0,\ldots,a_{d-1},b}(x_0,\ldots,x_{d-1}) = \left(\left(\sum_{i\in[d]} a_i x_i\right) + b\right)[\bar{w}-l,\bar{w}) \tag{2.5}$$

where:

- $x_0,\ldots,x_{d-1}$ is a vector of w-bit integers to be hashed.

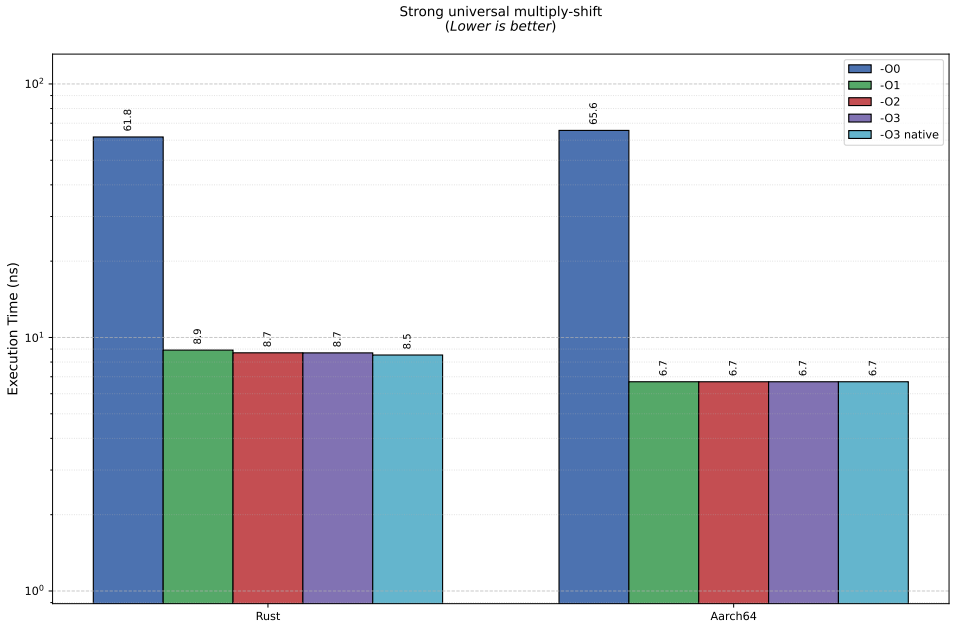

Figure 2.4: Strong universal multiply-shift implementations' performance with different optimization levels.

- $a_0, \ldots, a_{d-1}$ is a vector of independent uniform w-bit integers.
- b is an independent uniform w-bit integer.

For the function to be strong universal, it is required that $\bar{w} \geq w + l - 1$. Similar to the strong universal multiply-shift presented in Chapter 2.2.2, to be able to hash vectors of 64-bit integers ($w = 64$) into integers of at most 64 bits ($l \leq 64$), we set $\bar{w} = 128$.

The Rust implementation is as follows:

```
#[inline]
pub fn vector_multiply_shift(x: &[u64], l: u32, a: &[u128], b: u128) -> u64 {
    let mut ax_sum = b;
    for (xi, ai) in x.iter().zip(a.iter()) {
        ax_sum = ax_sum.wrapping_add(ai.wrapping_mul(*xi as u128));
    }
    ax_sum.wrapping_shr(128 - l) as u64
}
```

Code 2.14: Rust strong universal vector multiply-shift implementation.

The inline AArch64 assembly implementation is directly derived from the Rust version:

```
#[inline]
pub fn vector_multiply_shift(x: &[u64], l: u32, a: &[u128], b: u128) -> u64 {
    let b_lo = b as u64;
```

```
    let b_hi = (b >> 64) as u64;
    let result: u64;
    unsafe {
        asm!(
        "CBZ {len}, 1f", // If len == 0, skip to label 1
        "0:", // loop label
        "LDR {xi}, [{x_ptr}], #8", // Load xi
        "LDP {ai_lo}, {ai_hi}, [{a_ptr}], #16", // Load LSB and MSB of ai
        "UMULH {aixi_hi}, {ai_lo}, {xi}",
        "MUL {aixi_lo}, {ai_lo}, {xi}", // LSB of ai * xi
        "MADD {aixi_hi}, {ai_hi}, {xi}, {aixi_hi}", // MSB of ai * xi
        "ADDS {ax_sum_lo}, {ax_sum_lo}, {aixi_lo}", // LSB of ax_sum += ai * xi
        "ADC {ax_sum_hi}, {ax_sum_hi}, {aixi_hi}", // MSB of ax_sum += ai * xi
        "SUBS {len}, {len}, #1", // len -= 1
        "BNE 0b", // If len != 0, repeat loop

        "1:", // Label 1
        "LSR {ax_sum_lo}, {ax_sum_hi}, {shift}", // ax_sum >> (128 - l)

        ax_sum_lo = inout(reg) b_lo => result,
        ax_sum_hi = inout(reg) b_hi => _,
        x_ptr = inout(reg) x.as_ptr() => _, a_ptr = inout(reg) a.as_ptr() => _,
        len = inout(reg) x.len() => _, shift = in(reg) 128 - l as u64,
        xi = out(reg) _, ai_lo = out(reg) _, ai_hi = out(reg) _,
        aixi_lo = out(reg) _, aixi_hi = out(reg) _,
        options(readonly, nostack)
        )
    }
    result
}
```

Code 2.15: Inline AArch64 assembly strong universal vector multiply-shift implementation.

Figures 2.5 and 2.6 were generated using the following parameters: a = vector of $0x1\,A34F\,98C1\,2B5E\,71D0\,E4A3$, $b = 0xC51A\,2938\,B47D\,6E5F\,1029$, and $l = 20$ with `opt-level = 3`. The graphs show that the benefits of using an AArch64 assembly implementation are quickly dominated by the time needed to access memory.

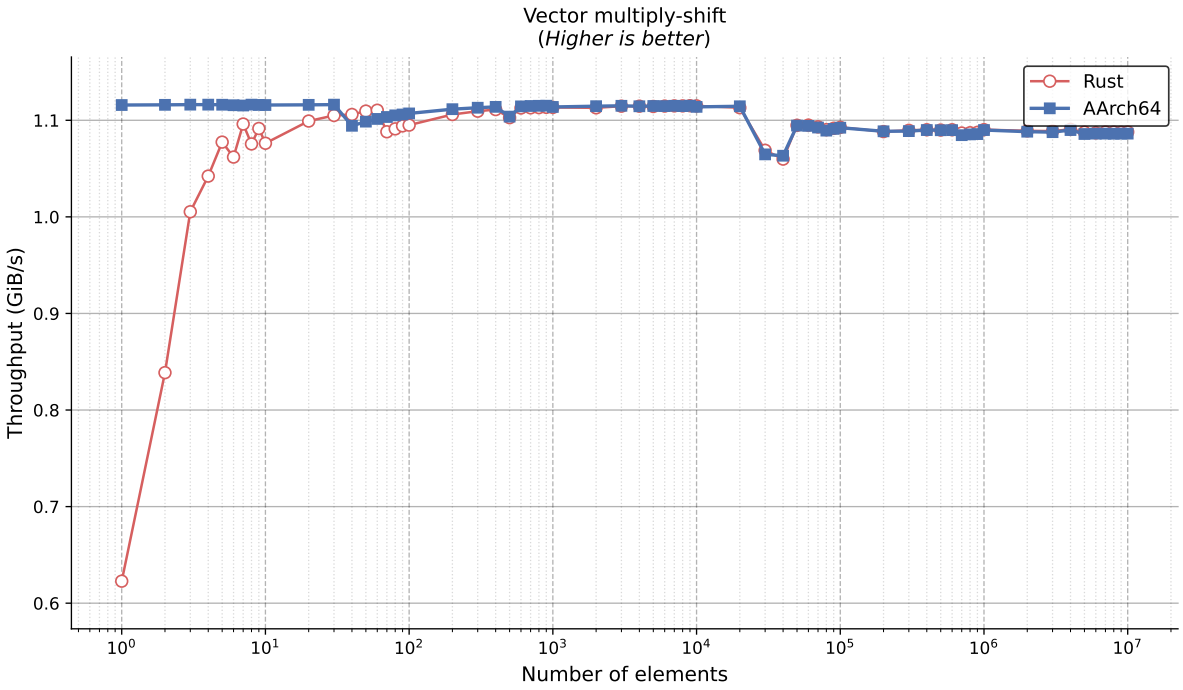

Figure 2.5: Strong universal vector multiply-shift implementations' throughput.

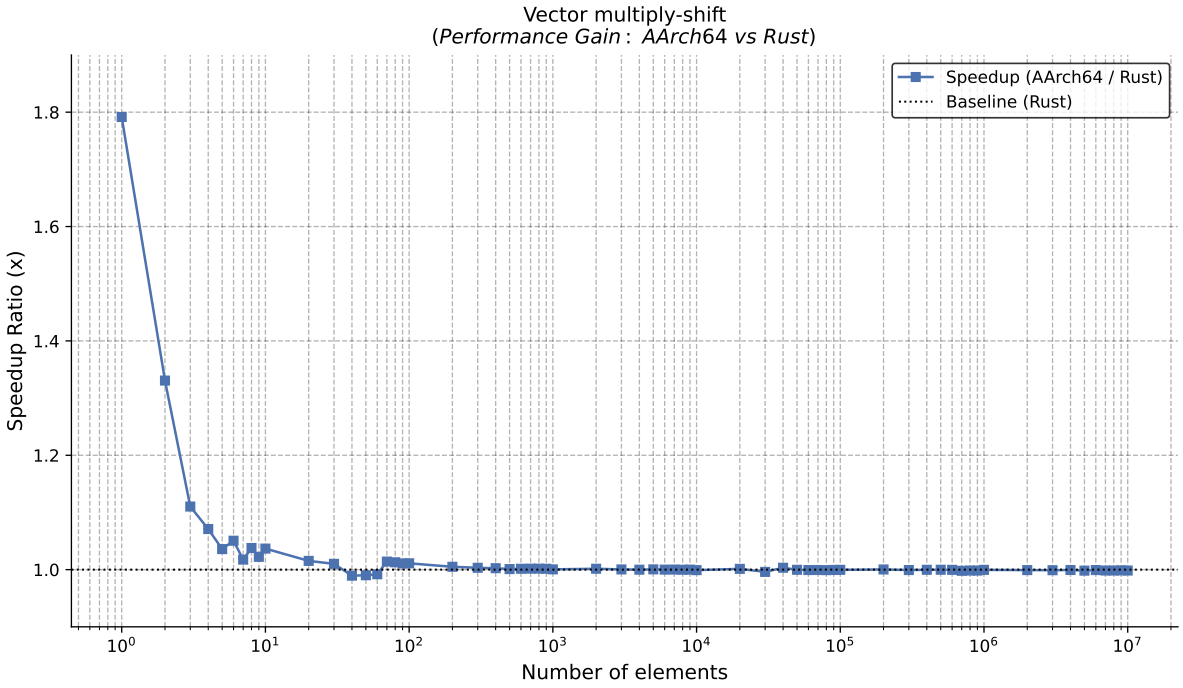

Figure 2.6: Strong universal vector multiply-shift implementations' delta graph.

2.2.4 Pair-multiply-shift

Based on [3, Chapter 3.5], the **strong universal pair multiply-shift** hash function is $h_{a_0,\ldots,a_{d-1},b} : [2^w]^d \to [2^l]$ and is defined as:

$$h_{a_0,\ldots,a_{d-1},b}(x_0,\ldots,x_{d-1}) = \left(\left(\sum_{i\in[d/2]} (a_{2i} + x_{2i+1}) \cdot (a_{2i+1} + x_{2i})\right) + b\right) [\bar{w} - l, \bar{w}) \quad (2.6)$$

Again, strong universality is achieved with $\bar{w} \geq w + l - 1$, therefore $\bar{w} = 128$. When d is odd, the last element is added to the hash by adding $a_{d-1}x_{d-1}$.

This function takes the same parameters as vector multiply-shift presented in Chapter 2.2.3. However, for a vector of n elements (n even), instead of performing n multiplications and $n-1$ additions, it performs $n + (\frac{n}{2} - 1)$ additions and $\frac{n}{2}$ multiplications. Counting one clock cycle per addition and three clock cycles per multiplication, the pair-multiply-shift hash function is expected to be $\lim_{n\to+\infty} \frac{4n-1}{3n-1} = \frac{4}{3} \approx 1.33$ times faster than the vector multiply-shift.

The Rust implementation is as follows:

```
#[inline]
pub fn pair_multiply_shift(x: &[u64], l: u32, a: &[u128], b: u128) -> u64 {
    let pairs = x.len() / 2;
    let mut sum = b;
    for (a_pair, x_pair) in a.chunks_exact(2).zip(x.chunks_exact(2)).take(pairs
    ) {
        let ax_even = a_pair[0].wrapping_add(x_pair[1] as u128);
        let ax_odd = a_pair[1].wrapping_add(x_pair[0] as u128);
        sum = sum.wrapping_add(ax_even.wrapping_mul(ax_odd));
    }
    if x.len() % 2 == 1 {
        let last = (a[x.len() - 1], x[x.len() - 1]);
        sum = sum.wrapping_add(last.0.wrapping_mul(last.1 as u128));
    }
    sum.wrapping_shr(128 - l) as u64
}
```

Code 2.16: Rust strong universal pair multiply-shift implementation.

The inline AArch64 assembly implementation is as follows:

```
#[inline]
pub fn pair_multiply_shift(x: &[u64], l: u32, a: &[u128], b: u128) -> u64 {
    let b_lo = b as u64;
    let b_hi = (b >> 64) as u64;
```

```
5       let result: u64;
6       unsafe {
7           asm!(
8           "CBZ {len}, 2f", // If len == 0, skip to label 2
9           "LSR {pairs}, {len}, #1", // pairs = len / 2
10          "CBZ {pairs}, 1f", // If pairs == 0, skip to label 1
11          "0:", // loop label
12          "LDP {ax_even_lo}, {ax_even_hi}, [{a_ptr}], #16", // Load LSB and MSB
    of a_even
13          "LDP {x_even}, {x_odd}, [{x_ptr}], #16", // Load x_even and x_odd
14          "LDP {ax_odd_lo}, {ax_odd_hi}, [{a_ptr}], #16", // Load LSB and MSB of
    a_odd
15          "ADDS {ax_even_lo}, {ax_even_lo}, {x_odd}", // LSB of ax_even = a_even
    + x_odd
16          "ADC {ax_even_hi}, {ax_even_hi}, xzr", // MSB of ax_even = a_even +
    x_odd
17          "ADDS {ax_odd_lo}, {ax_odd_lo}, {x_even}", // LSB of ax_odd = a_odd +
    x_even
18          "ADC {ax_odd_hi}, {ax_odd_hi}, xzr", // MSB of ax_odd = a_odd + x_even
19          "UMULH {tmp1}, {ax_odd_lo}, {ax_even_lo}",
20          "MADD {tmp1}, {ax_odd_hi}, {ax_even_lo}, {tmp1}",
21          "MUL {ax_even_lo}, {ax_odd_lo}, {ax_even_lo}", // LSB of (a_even +
    x_odd) * (a_odd + x_even)
22          "MADD {ax_even_hi}, {ax_odd_lo}, {ax_even_hi}, {tmp1}", // MSB of (
    a_even + x_odd) * (a_odd + x_even)
23          "ADDS {ax_sum_lo}, {ax_sum_lo}, {ax_even_lo}", // LSB of ax_sum += (
    a_even + x_odd) * (a_odd + x_even)
24          "ADC {ax_sum_hi}, {ax_sum_hi}, {ax_even_hi}", // MSB of ax_sum += (
    a_even + x_odd) * (a_odd + x_even)
25          "SUBS {pairs}, {pairs}, #1", // Decrement pairs
26          "BNE 0b", // If pairs != 0, repeat loop
27
28          "1:", // Label 1
29          "TST {len}, #1", // Check if len is odd
30          "BEQ 2f", // If len is even, skip to label 2
31          "LDP {ax_even_lo}, {ax_even_hi}, [{a_ptr}]", // Load LSB and MSB of a[
    len-1]
32          "LDR {x_even}, [{x_ptr}]", // Load x[len-1]
33          "UMULH {ax_odd_lo}, {ax_even_lo}, {x_even}", // ax_odd_lo for carry
```

```
34          "MUL {ax_even_lo}, {ax_even_lo}, {x_even}", // LSB of ax_even = a[len
    -1] * x[len-1]
35          "MADD {ax_even_hi}, {ax_even_hi}, {x_even}, {ax_odd_lo}", // MSB of
    ax_even = a[len-1] * x[len-1]
36          "ADDS {ax_sum_lo}, {ax_sum_lo}, {ax_even_lo}", // LSB of ax_sum += a[
    len-1] * x[len-1]
37          "ADC {ax_sum_hi}, {ax_sum_hi}, {ax_even_hi}", // MSB of ax_sum += a[len
    -1] * x[len-1]
38
39          "2:", // Label 2
40          "LSR {ax_sum_lo}, {ax_sum_hi}, {shift}", // ax_sum >> (128 - l)
41
42          ax_sum_lo = inout(reg) b_lo => result,
43          ax_sum_hi = inout(reg) b_hi => _,
44          x_ptr = inout(reg) x.as_ptr() => _, a_ptr = inout(reg) a.as_ptr() => _,
45          len = in(reg) x.len(), shift = in(reg) 128 - l as u64,
46          pairs = out(reg) _,
47          x_even = out(reg) _, x_odd = out(reg) _,
48          ax_even_lo = out(reg) _, ax_even_hi = out(reg) _,
49          ax_odd_lo = out(reg) _, ax_odd_hi = out(reg) _,
50          tmp1 = out(reg) _,
51          options(readonly, nostack)
52          )
53      }
54      result
55  }
```

Code 2.17: Inline AArch64 assembly strong universal pair multiply-shift implementation.

Figures 2.7 and 2.8 were generated using the same parameters used for Figures 2.5 and 2.6: $a =$ vector of $0x1\,A34F\,98C1\,2B5E\,71D0\,E4A3$, $b = 0xC51A\,2938\,B47D\,6E5F\,1029$, and $l = 20$.

Again, like Chapter 2.2.3, the benefits of using an AArch64 inline assembly implementation are dominated by the time needed to access the memory. In addition, there is a significant degradation of performance at around 10^5 elements. Since the size of the L2 cache of the BCM2711 chip is 1MB, it can hold a vector of $1.25\,10^5$ `u64` integers.

The zig-zag pattern in Figure 2.7 for short vectors is the effect of the last element in odd vectors.

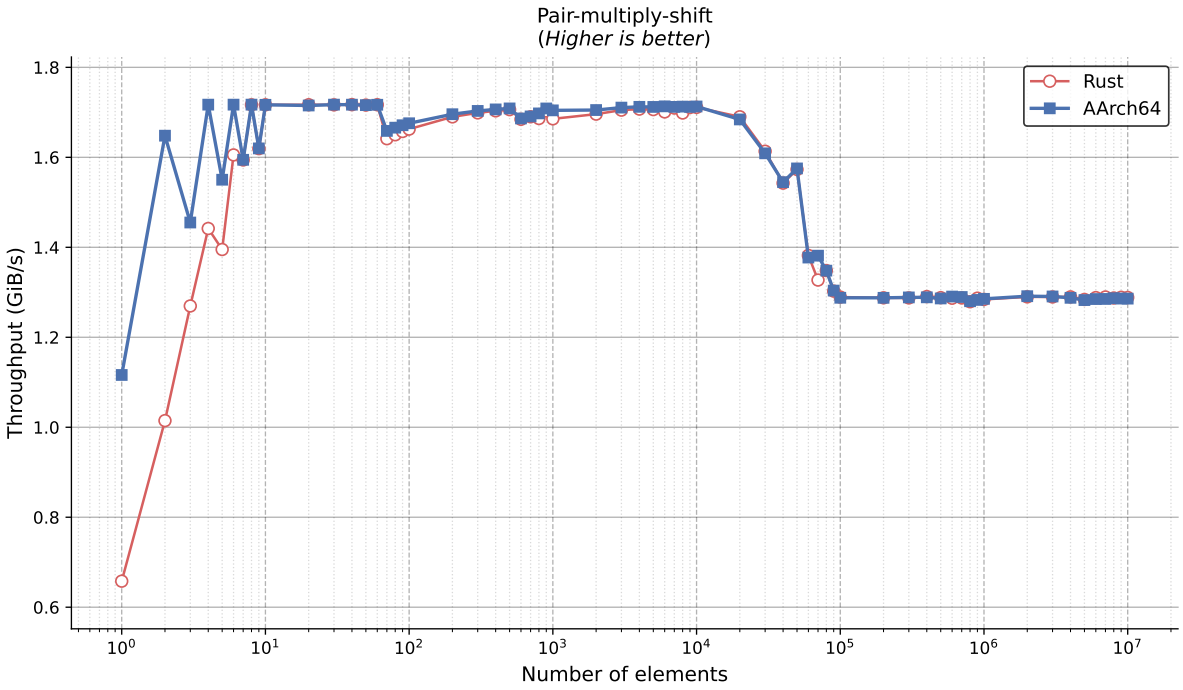

Figure 2.7: Strong universal pair-multiply-shift implementations' throughput.

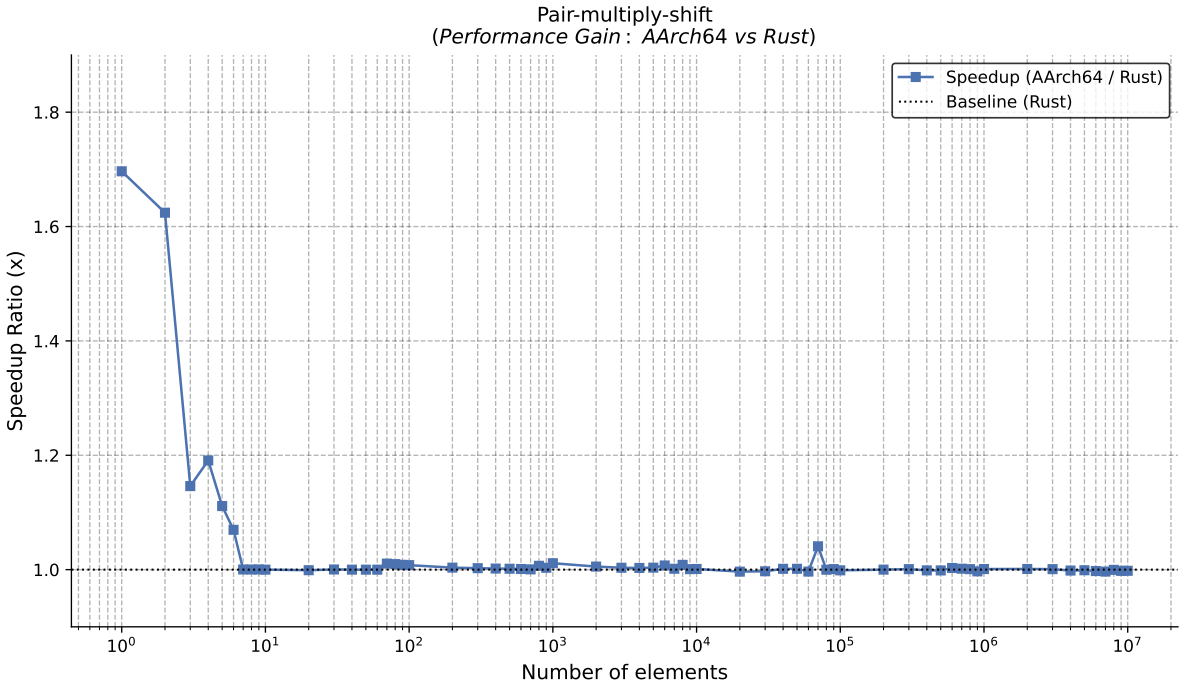

Figure 2.8: Strong universal pair-multiply-shift implementations' delta graph.

2.3 String hash functions

2.3.1 Bounded length

Based on [3, Chapter 5.2], the first hash function for strings is a variation of pair-multiply-shift presented in Chapter 2.2.4:

$$h_{a_0,\ldots,a_D}(x_0,\ldots,x_{d-1}) = \left(\left(\sum_{i\in[d/2]}(a_{2i}+x_{2i+1})(a_{2i+1}+x_{2i})\right)+a_d\right)[\bar{w}-l,\bar{w}) \qquad (2.7)$$

In the equation 2.7, the elements of the vector x are not single characters (8-bit integers) as it would be too expensive to compute the product for every character pair. Instead, the bytes are grouped into 32-bit integers and therefore $d = \lceil \frac{c}{4} \rceil$, where $c < 256$ is the number of characters of the input string. The constant `MAX_LENGTH` is used to store the maximum number of characters considered by the hash function.

With x a vector of 32-bit integers, the equation 2.7 produces a 32-bit integer as a result. By computing two separate 32-bit hashes, we can obtain a final 64-bit integer through concatenation. Consequently, the function will require two vectors a of at least $2 \cdot d$ 64-bit integers (capped at $256/4 + 1$), or 2 elements if the string provided is empty.

In the Rust implementation the generation of the vector x from the input string is done as follows: the first 256 characters of the string are copied to the beginning of an array initialized with $256/4$ 32-bit zeros. This operation is done with the unsafe operator `std::ptr::copy_nonoverlapping`, an equivalent of C's `memcpy`, which copies the 8-bit integers in x as 32-bit integers. Instead of calling the function twice to get the 64-bit hash, the function uses a **loop fusion** technique to access the x vector only once.

The Rust implementation is as follows:

```
#[inline]
pub fn bounded_length_hash(s: &[u8], l: u32, a: (&[u64], &[u64])) -> u64 {
    let l_1 = l >> 1; // l_1 = l / 2
    let l_2 = l - l_1; // l_2 = l - l_1
    let d = (s.len() + 3) >> 2; // d = ceil(s.len() / 4)

    let mut x = [0u32; MAX_LENGTH / 4];
    unsafe {
        // memcpy equivalent
        std::ptr::copy_nonoverlapping(
        s.as_ptr(),
        x.as_mut_ptr() as *mut u8,
```

```
        std::cmp::min(s.len(), MAX_LENGTH),
        );
    }

    let d = std::cmp::min(d, MAX_LENGTH / 4);
    let mut sum1: u64 = 0;
    let mut sum2: u64 = 0;
    for i in 0..=(d >> 1).saturating_sub(1) {
        let term11 = a.0[2 * i].wrapping_add(x[2 * i + 1] as u64);
        let term12 = a.0[2 * i + 1].wrapping_add(x[2 * i] as u64);
        sum1 = sum1.wrapping_add(term11.wrapping_mul(term12));
        let term21 = a.1[2 * i + 1].wrapping_add(x[2 * i] as u64);
        let term22 = a.1[2 * i].wrapping_add(x[2 * i + 1] as u64);
        sum2 = sum2.wrapping_add(term21.wrapping_mul(term22));
    }
    let sum1 = if l_1 == 0 {
        0
    } else {
        sum1.wrapping_add(a.0[d]).wrapping_shr(64 - l_1) as u32
    };
    let sum2 = sum2.wrapping_add(a.1[d]).wrapping_shr(64 - l_2) as u32;
    ((sum2 as u64) << l_1) | (sum1 as u64)
}
```

Code 2.18: Rust strong universal bounded length string hash function implementation.

In assembly, numbers are manipulated as byte streams rather than integers. Therefore, `std::ptr::copy_nonoverlapping` is not required, because x can be an array of 256 8-bit zeros. Later, in the `asm!()` macro, the array will be read in 32-bit chunks.

The inline AArch64 assembly implementation is as follows:

```
#[inline]
pub fn bounded_length_hash(s: &[u8], l: u32, a: (&[u64], &[u64])) -> u64 {
    let l_1 = l >> 1; // l_1 = l / 2
    let l_2 = l - l_1; // l_2 = l - l_1
    let d = (s.len() + 3) >> 2; // d = ceil(s.len() / 4)

    let mut x_bytes = [0u8; MAX_LENGTH];
    // Copy s_bytes into x_bytes
    x_bytes[..(std::cmp::min(s.len(), MAX_LENGTH))]
    .copy_from_slice(&s[..(std::cmp::min(s.len(), MAX_LENGTH))]);
```

```
    let d = std::cmp::min(d, MAX_LENGTH / 4);
    let result: u64;
    unsafe {
        asm!(
        "LSR {i}, {d}, #1",
        "SUBS {i}, {i}, #1",
        "CSEL {i}, xzr, {i}, mi", // i = d >> 1 - 1
        "0:", // loop label
        "LDR {char:w}, [{x_ptr}], #4", // char = x[2i]
        "LDP {term11}, {term12}, [{a1_ptr}], #16", // Load a.0[2i] in term11, a.0[2i + 1] in term12
        "ADD {term12}, {term12}, {char}", // term12 = a.0[2i+1] + x[2i]
        "LDP {term21}, {term22}, [{a2_ptr}], #16", // Load a.1[2i] in term21, a.1[2i + 1] in term22
        "ADD {term22}, {term22}, {char}", // term22 = a.1[2i+1] + x[2i]
        "LDR {char:w}, [{x_ptr}], #4", // char = x[2i+1]
        "ADD {term11}, {term11}, {char}", // term11 = a.0[2i] + x[2i+1]
        "ADD {term21}, {term21}, {char}", // term21 = a.1[2i] + x[2i+1]
        "MADD {sum1}, {term11}, {term12}, {sum1}", // sum1 += term11 * term12
        "MADD {sum2}, {term21}, {term22}, {sum2}", // sum2 += term21 * term22
        "SUBS {i}, {i}, #1", // Decrement d
        "BGE 0b", // If i >= 0, repeat the loop

        "1:", // Label 1
        "ADD {sum1}, {sum1}, {ad1}", // sum1 += a.0[2d]
        "ADD {sum2}, {sum2}, {ad2}", // sum2 += a.1[2d]
        "LSR {sum1}, {sum1}, {shift1}", // sum1 >>= (64 - l_1)
        "CMP {l_1}, #0", // Check if l_1 is 0
        "CSEL {sum1}, {sum1}, XZR, NE", // If l_1 != 0 sum1 >>= (64 - l_1) else sum1 = 0
        "LSR {sum2}, {sum2}, {shift2}", // sum2 >> (64 - l_2)
        "LSL {sum2}, {sum2}, {l_1}", // sum2 << l_1
        "ORR {result}, {sum2}, {sum1}", // result = (sum2 << l_1) | sum1

        result = out(reg) result,
        x_ptr = inout(reg) x_bytes.as_ptr() => _,
        a1_ptr = inout(reg) (a.0).as_ptr() => _,
        a2_ptr = inout(reg) (a.1).as_ptr() => _,
        d = inout(reg) d => _, shift1 = in(reg) 64 - l_1 as u64,
```

```
47          shift2 = in(reg) 64 - l_2 as u64, l_1 = in(reg) l_1 as u64,
48          ad1 = in(reg) a.0[d], ad2 = in(reg) a.1[d],
49          sum1 = inout(reg) 0u64 => _, sum2 = inout(reg) 0u64 => _,
50          char = out(reg) _, i = out(reg) _,
51          term11 = out(reg) _, term12 = out(reg) _, term21 = out(reg) _,
52          term22 = out(reg) _,
53          options(readonly, nostack)
54          )
55      }
56      result
57  }
```

Code 2.19: Inline AArch64 assembly strong universal bounded length string hash function implementation.

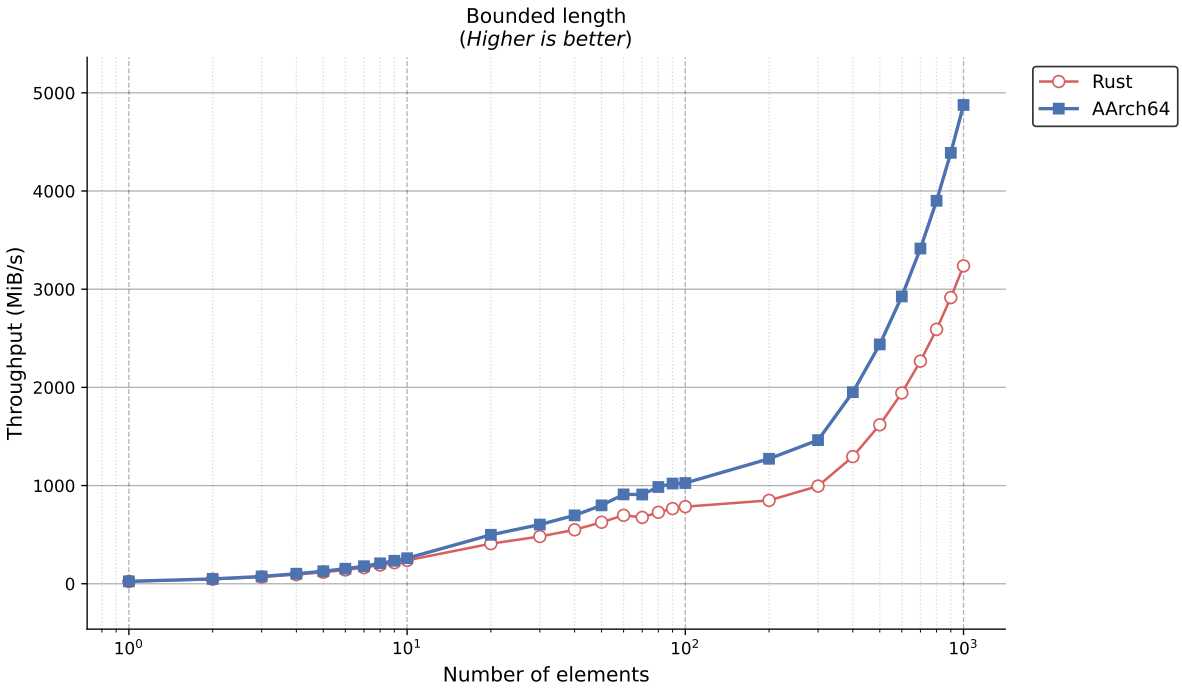

Figure 2.9: Strong universal bounded length implementations' throughput.

Figures 2.9 and 2.10 were generated using the following parameters: both a = vector of $0x9E3779B97F4A7C15$ and $m_{bits} = 20$ with `opt-level = 3`.

For strings longer than 256 characters, the function's performance can be seen increasing in Figure 2.9. The reason for this is that the function only considers the first 256 characters and the execution time remains constant. Additionally, the delta in Figure 2.10 grows steadily until $c = 256$, after which it becomes constant.

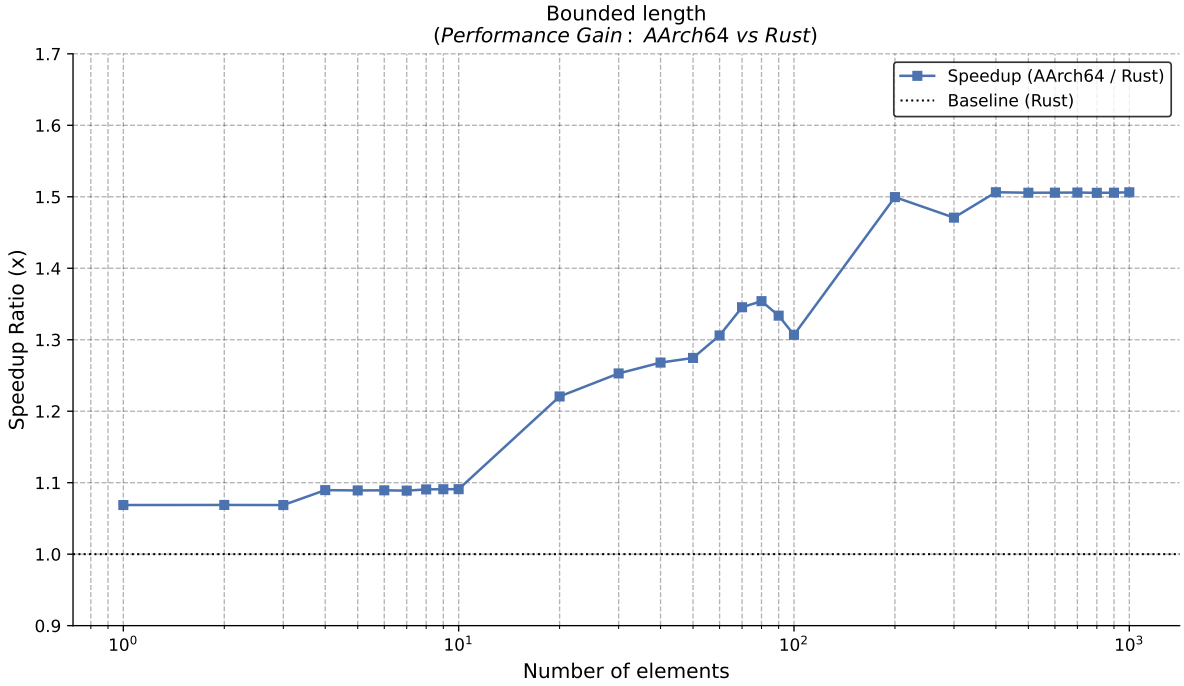

Figure 2.10: Strong universal bounded length implementations' delta graph.

2.3.2 Variable length

Based on [3, Chapter 5.3], the second hash function for strings is a composition of a polynomial computed using Horner's rule and a universal multiply-mod-prime scheme:

$$h_{a,b,c}(x_0, \ldots, x_d) = \left(\left(a \left(\sum_{i=0}^{d} x_{d-i} c^i \right) + b \right) \bmod p \right) \bmod m \tag{2.8}$$

Unlike the bounded length hash function presented in Chapter 2.3.1, this function can hash strings of practically any length, although it is computationally more expensive. p is a Mersenne prime and a, b, and c are three integers in $[p]$. m is the output range of the function.

Since the polynomial, computed with Horner's method, is composed with a universal hash function that hashes $[p] \to [m]$, a Mersenne reduction is required after every step of the loop to ensure that the result remains within $[p]$.

In the first Rust implementation, as in Chapter 2.3.1, x is a vector obtained by grouping the characters of the input string into 64-bit integers.

Since the collision probability of the function is $\frac{d}{p}$, we require $m \leq \frac{p}{d}$. To obtain 64-bit integers from the hash function, we choose $p \geq 2^{89} - 1$ so that it is possible to hash strings with $d = \left\lceil \frac{c}{8} \right\rceil \leq 2^{25}$. The crate `ethnum` is used to handle the operations with 128-bit operands.

The first Rust implementation is as follows:

```rust
#[inline]
pub fn variable_length_hash_1(s: &[u8], a: u128, b: u128, c: u128, q: u32, m_bits: u32) -> u64 {
    let p = u256::from(1u128.wrapping_shl(q).wrapping_sub(1)); // p = 2^q - 1
    let m_mask = 1u128.wrapping_shl(m_bits).wrapping_sub(1);

    let mut sum = u256::new(0);
    let a = u256::from(a);
    let b = u256::from(b);
    let c = u256::from(c);
    let chunks = s.chunks_exact(8); // Process 8 bytes at a time as u64
    let remainder = chunks.remainder(); // Remaining bytes after processing chunks of 8
    for chunk in chunks {
        let val = u64::from_ne_bytes(chunk.try_into().unwrap());
        // Horner's method
        sum = sum.wrapping_mul(c).wrapping_add(val.into());
        // Mersenne reduction
        sum = (sum & p) + (sum >> q);
        sum = if sum >= p { sum - p } else { sum };
    }
    if !remainder.is_empty() {
        // Process remaining bytes
        let mut word = [0u8; 8];
        word[..remainder.len()].copy_from_slice(remainder);
        let val = u64::from_ne_bytes(word);
        sum = sum.wrapping_mul(c).wrapping_add(val.into());
        sum = (sum & p) + (sum >> q);
        sum = if sum >= p { sum - p } else { sum };
    }

    sum = sum.wrapping_mul(a).wrapping_add(b);
    // Mersenne reduction
    sum = (sum & p) + (sum >> q);
    sum = if sum >= p { sum - p } else { sum };
    (sum.as_u128() & m_mask) as u64
}
```

Code 2.20: First Rust strong universal variable length string hash function implementation.

This implementation is expected to be relatively slow, as it relies on 256-bit operations. The 64-bit result can be computed as two separate 32-bit integers [3, Chapter 4.1]. This allows p to be reduced to a 64-bit integer, fixed at $p = 2^{61} - 1$, at the cost of requiring a pair of (a,b,c) triplets instead of one. This implementation of the function can hash strings with $d = \left\lceil \frac{c}{4} \right\rceil \leq 2^{29}$. The composition of the 64-bit integer result is achieved through loop fusion, similar to the bounded length implementations.

For reasons explained in Chapter 3.1, the function has two helper functions: one for computing the polynomial with Horner's method and one for applying the universal hash function. There is no penalty in doing this as the functions will be inlined.

The second Rust implementation is as follows:

```
#[inline]
pub fn variable_length_hash_2(
    s: &[u8],
    a: (u64, u64),
    b: (u64, u64),
    c: (u64, u64),
    m_bits: u32,
) -> u64 {
    let (sum1, sum2) = variable_length_horner((0, 0), s, c);
    variable_length_final((sum1, sum2), a, b, m_bits)
}

#[inline]
pub fn variable_length_horner(sum: (u128, u128), s: &[u8], c: (u64, u64)) -> (
    u128, u128) {
    let mut sum1 = sum.0;
    let mut sum2 = sum.1;

    let chunks = s.chunks_exact(4); // Process 4 bytes at a time as u32
    let remainder = chunks.remainder(); // Remaining bytes after grouping
    for chunk in chunks {
        let val = u32::from_ne_bytes(chunk.try_into().unwrap());
        // Horner's method
        sum1 = sum1.wrapping_mul(c.0 as u128).wrapping_add(val as u128);
        sum2 = sum2.wrapping_mul(c.1 as u128).wrapping_add(val as u128);
        // Mersenne reduction
        sum1 = (sum1 & P) + (sum1 >> MAX_Q_U64);
        if sum1 >= P {
```

```
            sum1 -= P;
        }
        sum2 = (sum2 & P) + (sum2 >> MAX_Q_U64);
        if sum2 >= P {
            sum2 -= P;
        }
    }
    if !remainder.is_empty() {
        // Process remaining bytes
        let mut word = [0u8; 4];
        word[..remainder.len()].copy_from_slice(remainder);
        let val = u32::from_ne_bytes(word);
        sum1 = sum1.wrapping_mul(c.0 as u128).wrapping_add(val as u128);
        sum2 = sum2.wrapping_mul(c.1 as u128).wrapping_add(val as u128);
        // Mersenne reduction
        sum1 = (sum1 & P) + (sum1 >> MAX_Q_U64);
        if sum1 >= P {
            sum1 -= P;
        }
        sum2 = (sum2 & P) + (sum2 >> MAX_Q_U64);
        if sum2 >= P {
            sum2 -= P;
        }
    }
    (sum1, sum2)
}

#[inline]
pub fn variable_length_final(sum: (u128, u128), a: (u64, u64), b: (u64, u64),
    m_bits: u32) -> u64 {
    let mut sum1 = sum.0.wrapping_mul(a.0 as u128).wrapping_add(b.0 as u128);
    let mut sum2 = sum.1.wrapping_mul(a.1 as u128).wrapping_add(b.1 as u128);
    // Mersenne reduction
    sum1 = (sum1 & P) + (sum1 >> MAX_Q_U64);
    if sum1 >= P {
        sum1 -= P;
    }
    sum2 = (sum2 & P) + (sum2 >> MAX_Q_U64);
    if sum2 >= P {
```

```
66          sum2 -= P;
67      }
68
69      let m_bits_1 = m_bits >> 1;
70      let m_bits_2 = m_bits - m_bits_1;
71      let m_mask_1 = 1u64.wrapping_shl(m_bits_1).wrapping_sub(1);
72      let m_mask_2 = 1u64.wrapping_shl(m_bits_2).wrapping_sub(1);
73      ((sum1 as u64 & m_mask_1) << m_bits_2) | (sum2 as u64 & m_mask_2)
74  }
```

Code 2.21: Second Rust strong universal variable length string hash function implementation.

While the Rust implementations use the `chunks_exact` and `remainder` functions to group the characters into 32-bit or 64-bit integers, the assembly implementation can skip the grouping, and the remainder is managed by slicing the input string, prior to the `asm!()` macro.

The computation of the two 32-bit integers that make up the 64-bit result is performed by interleaving the assembly instructions. This approach reduces data dependencies between adjacent instructions, allowing the processor to exploit instruction-level parallelism within its pipeline.

Unlike the previous approach, the hash function implementation is not done by chaining the two helper functions. This is because the Rust compiler is not able to inline the `asm!()` macro. The two helper functions will be later used in Chapter 3.1.

The inline AArch64 assembly implementation is as follows:

```
1   #[inline]
2   pub fn variable_length_hash(
3       s: &[u8],
4       a: (u64, u64),
5       b: (u64, u64),
6       c: (u64, u64),
7       m_bits: u32,
8   ) -> u64 {
9       let mut buf: [u8; 4] = [0; 4]; // Remainder
10      buf[0..s.len() & 3].copy_from_slice(&s[s.len() - (s.len() & 3)..]);
11      let m_bits_1 = m_bits >> 1;
12      let m_bits_2 = m_bits - m_bits_1;
13      let m_mask_1 = 1u64.wrapping_shl(m_bits_1).wrapping_sub(1);
14      let m_mask_2 = 1u64.wrapping_shl(m_bits_2).wrapping_sub(1);
15      let result: u64;
16      unsafe {
17          asm!(
```

```
18          "LSR {i}, {len}, #2", // i = len >> 2
19          "CBZ {i}, 1f", // If i = 0, skip to label 1
20          "0:", // loop label
21          "LDR {char:w}, [{s_ptr}], #4", // Load 4 bytes
22          "UMULH {sum1_hi}, {sum1_lo}, {c1}", // MSB sum1 *= c.0 (MSB = 0)
23          "MUL {sum1_lo}, {sum1_lo}, {c1}", // LSB sum1 *= c.0
24          "UMULH {sum2_hi}, {sum2_lo}, {c2}", // MSB sum2 *= c.0 (MSB = 0)
25          "MUL {sum2_lo}, {sum2_lo}, {c2}", // LSB sum2 *= c.0
26          "ADDS {sum1_lo}, {sum1_lo}, {char}", // LSB sum1 += char
27          "ADC {sum1_hi}, {sum1_hi}, xzr", // MSB sum1 += char
28          "ADDS {sum2_lo}, {sum2_lo}, {char}", // LSB sum2 += char
29          "ADC {sum2_hi}, {sum2_hi}, xzr", // MSB sum2 += char
30          "AND {sump}, {sum1_lo}, {p}", // LSB of sum1&p (MSB not necessary)
31          "EXTR {sum1_lo}, {sum1_hi}, {sum1_lo}, {q}", // LSB of sum1>>q
32          "LSR {sum1_hi}, {sum1_hi}, {q}", // MSB of sum1>>q
33          "ADDS {sum1_lo}, {sump}, {sum1_lo}", // LSB of sum1 = (sum1&p)+(sum1>>q
    )
34          "ADC {sum1_hi}, {sum1_hi}, xzr", // MSB of sum1 = (sum1&p)+(sum1>>q)
35          "AND {sump}, {sum2_lo}, {p}", // LSB of sum2&p (MSB not necessary)
36          "EXTR {sum2_lo}, {sum2_hi}, {sum2_lo}, {q}", // LSB of sum2>>q
37          "LSR {sum2_hi}, {sum2_hi}, {q}", // MSB of sum2>>q
38          "ADDS {sum2_lo}, {sump}, {sum2_lo}", // LSB of sum2 = (sum2&p)+(sum2>>q
    )
39          "ADC {sum2_hi}, {sum2_hi}, xzr", // MSB of sum2 = (sum2&p)+(sum2>>q)
40          "CMP {sum1_lo}, {p}",
41          "CSEL {tmp}, {p}, xzr, HS", // tmp = p if sum1_lo >= p, else 0
42          "SUB {sum1_lo}, {sum1_lo}, {tmp}", // sum1_lo -= tmp
43          "CMP {sum2_lo}, {p}",
44          "CSEL {tmp}, {p}, xzr, HS", // tmp = p if sum2_lo >= p, else 0
45          "SUB {sum2_lo}, {sum2_lo}, {tmp}", // sum2_lo -= tmp
46          "SUBS {i}, {i}, #1", // i--
47          "BGT 0b", // If i > 0, repeat the loop
48          "1:", // Label 1
49          "TST {len}, #3", // Test if there are remaining bytes (len & 3)
50          "BEQ 2f", // If no remaining bytes, skip to label 2
51          "LDR {char:w}, [{rem_ptr}]", // Load remaining bytes
52          "UMULH {sum1_hi}, {sum1_lo}, {c1}", // MSB sum1 *= c.0 (MSB = 0)
53          "MUL {sum1_lo}, {sum1_lo}, {c1}", // LSB sum1 *= c.0
54          "UMULH {sum2_hi}, {sum2_lo}, {c2}", // MSB sum2 *= c.0 (MSB = 0)
```

```
55          "MUL {sum2_lo}, {sum2_lo}, {c2}", // LSB sum2 *= c.0
56          "ADDS {sum1_lo}, {sum1_lo}, {char}", // LSB sum1 += remaining bytes
57          "ADC {sum1_hi}, {sum1_hi}, xzr", // MSB sum1 += remaining bytes
58          "ADDS {sum2_lo}, {sum2_lo}, {char}", // LSB sum2 += remaining bytes
59          "ADC {sum2_hi}, {sum2_hi}, xzr", // MSB sum2 += remaining bytes
60          "AND {sump}, {sum1_lo}, {p}", // LSB of sum1&p (MSB not necessary)
61          "EXTR {sum1_lo}, {sum1_hi}, {sum1_lo}, {q}", // LSB of sum1>>q
62          "LSR {sum1_hi}, {sum1_hi}, {q}", // MSB of sum1>>q
63          "ADDS {sum1_lo}, {sump}, {sum1_lo}", // LSB of sum1 = (sum1&p)+(sum1>>q
    )
64          "ADC {sum1_hi}, {sum1_hi}, xzr", // MSB of sum1 = (sum1&p)+(sum1>>q)
65          "AND {sump}, {sum2_lo}, {p}", // LSB of sum2&p (MSB not necessary)
66          "EXTR {sum2_lo}, {sum2_hi}, {sum2_lo}, {q}", // LSB of sum2>>q
67          "LSR {sum2_hi}, {sum2_hi}, {q}", // MSB of sum2>>q
68          "ADDS {sum2_lo}, {sump}, {sum2_lo}", // LSB of sum2 = (sum2&p)+(sum2>>q
    )
69          "ADC {sum2_hi}, {sum2_hi}, xzr", // MSB of sum2 = (sum2&p)+(sum2>>q)
70          "CMP {sum1_lo}, {p}",
71          "CSEL {tmp}, {p}, xzr, HS", // tmp = p if sum1_lo >= p, else 0
72          "SUB {sum1_lo}, {sum1_lo}, {tmp}", // sum1_lo -= tmp
73          "CMP {sum2_lo}, {p}",
74          "CSEL {tmp}, {p}, xzr, HS", // tmp = p if sum2_lo >= p, else 0
75          "SUB {sum2_lo}, {sum2_lo}, {tmp}", // sum2_lo -= tmp
76          "2:", // Label 2
77
78          "UMULH {sum1_hi}, {sum1_lo}, {a1}", // MSB sum1 *= a.0 (MSB = 0)
79          "MUL {sum1_lo}, {sum1_lo}, {a1}", // LSB sum1 *= a.0
80          "UMULH {sum2_hi}, {sum2_lo}, {a2}", // MSB sum2 *= a.1 (MSB = 0)
81          "MUL {sum2_lo}, {sum2_lo}, {a2}", // LSB sum2 *= a.1
82          "ADDS {sum1_lo}, {sum1_lo}, {b1}", // LSB sum1 += b.0
83          "ADC {sum1_hi}, {sum1_hi}, xzr", // MSB sum1 += b.0
84          "ADDS {sum2_lo}, {sum2_lo}, {b2}", // LSB sum2 += b.1
85          "ADC {sum2_hi}, {sum2_hi}, xzr", // MSB sum2 += b.1
86          "AND {sump}, {sum1_lo}, {p}", // LSB of sum1&p (MSB not necessary)
87          "EXTR {sum1_lo}, {sum1_hi}, {sum1_lo}, {q}", // LSB of sum1>>q
88          "LSR {sum1_hi}, {sum1_hi}, {q}", // MSB of sum1>>q
89          "ADDS {sum1_lo}, {sump}, {sum1_lo}", // LSB of sum1 = (sum1&p)+(sum1>>q
    )
90          "ADC {sum1_hi}, {sum1_hi}, xzr", // MSB of sum1 = (sum1&p)+(sum1>>q)
```

```
91          "AND {sump}, {sum2_lo}, {p}", // LSB of sum2&p (MSB not necessary)
92          "EXTR {sum2_lo}, {sum2_hi}, {sum2_lo}, {q}", // LSB of sum2>>q
93          "LSR {sum2_hi}, {sum2_hi}, {q}", // MSB of sum2>>q
94          "ADDS {sum2_lo}, {sump}, {sum2_lo}", // LSB of sum2 = (sum2&p)+(sum2>>q
    )
95          "ADC {sum2_hi}, {sum2_hi}, xzr", // MSB of sum2 = (sum2&p)+(sum2>>q)
96          "CMP {sum1_lo}, {p}",
97          "CSEL {tmp}, {p}, xzr, HS", // tmp = p if sum1_lo >= p, else 0
98          "SUB {sum1_lo}, {sum1_lo}, {tmp}", // sum1_lo -= tmp
99          "CMP {sum2_lo}, {p}",
100         "CSEL {tmp}, {p}, xzr, HS", // tmp = p if sum2_lo >= p, else 0
101         "SUB {sum2_lo}, {sum2_lo}, {tmp}", // sum2_lo -= tmp
102         "AND {sum1_lo}, {sum1_lo}, {m_mask1}", // sum1_lo &= m_mask_1
103         "AND {sum2_lo}, {sum2_lo}, {m_mask2}", // sum2_lo &= m_mask_2
104         "LSL {sum1_lo}, {sum1_lo}, {m_bits_2}", // sum1_lo <<= m_bits_2
105         "ORR {sum1_lo}, {sum1_lo}, {sum2_lo}", // sum1_lo |= sum2_lo
106
107         sum1_lo = inout(reg) 0u64 => result, sum1_hi = inout(reg) 0u64 => _,
108         sum2_lo = inout(reg) 0u64 => _, sum2_hi = inout(reg) 0u64 => _,
109         s_ptr = inout(reg) s.as_ptr() => _,
110         rem_ptr = inout(reg) buf.as_ptr() => _,
111         len = inout(reg) s.len() => _,
112         a1 = in(reg) a.0, a2 = in(reg) a.1,
113         b1 = in(reg) b.0, b2 = in(reg) b.1,
114         c1 = in(reg) c.0, c2 = in(reg) c.1,
115         m_mask1 = in(reg) m_mask_1, m_mask2 = in(reg) m_mask_2,
116         m_bits_2 = in(reg) m_bits_2 as u64,
117         tmp = out(reg) _, char = out(reg) _, sump = out(reg) _, i = out(reg) _,
118         q = const MAX_Q_U64 as u64, p = in(reg) P,
119         options(readonly, nostack)
120         )
121     }
122     result
123 }
124
125 #[inline]
126 pub fn variable_length_horner(sum: (u128, u128), s: &[u8], c: (u64, u64)) -> (
        u128, u128) {
127     let mut buf: [u8; 4] = [0; 4]; // Remainder
```

```
128     buf[0..s.len() & 3].copy_from_slice(&s[s.len() - (s.len() & 3)..]);
129
130     let mut sum1_lo: u64 = sum.0 as u64;
131     let mut sum1_hi: u64 = (sum.0 >> 64) as u64;
132     let mut sum2_lo: u64 = sum.1 as u64;
133     let mut sum2_hi: u64 = (sum.1 >> 64) as u64;
134     unsafe {
135         asm!(
136         // Same assembly code between lines 18 and 76
137
138         sum1_lo = inout(reg) sum1_lo, sum1_hi = inout(reg) sum1_hi,
139         sum2_lo = inout(reg) sum2_lo, sum2_hi = inout(reg) sum2_hi,
140         s_ptr = inout(reg) s.as_ptr() => _,
141         rem_ptr = inout(reg) buf.as_ptr() => _,
142         len = inout(reg) s.len() => _,
143         c1 = in(reg) c.0, c2 = in(reg) c.1,
144         tmp = out(reg) _, char = out(reg) _, sump = out(reg) _, i = out(reg) _,
145         q = const MAX_Q_U64 as u64, p = in(reg) P,
146         options(readonly, nostack)
147         )
148     }
149     let sum1 = (sum1_hi as u128).wrapping_shl(64) | sum1_lo as u128;
150     let sum2 = (sum2_hi as u128).wrapping_shl(64) | sum2_lo as u128;
151     (sum1, sum2)
152 }
153
154 #[inline]
155 pub fn variable_length_final(sum: (u128, u128), a: (u64, u64), b: (u64, u64),
        m_bits: u32) -> u64 {
156     let m_bits_1 = m_bits >> 1;
157     let m_bits_2 = m_bits - m_bits_1;
158     let m_mask_1 = 1u64.wrapping_shl(m_bits_1).wrapping_sub(1);
159     let m_mask_2 = 1u64.wrapping_shl(m_bits_2).wrapping_sub(1);
160
161     let sum1_lo: u64 = sum.0 as u64;
162     let sum1_hi: u64 = (sum.0 >> 64) as u64;
163     let sum2_lo: u64 = sum.1 as u64;
164     let sum2_hi: u64 = (sum.1 >> 64) as u64;
165     let result: u64;
```

```
166     unsafe {
167         asm!(
168         // Same assembly code between lines 78 and 105
169 
170         sum1_lo = inout(reg) sum1_lo => result, sum1_hi = inout(reg) sum1_hi =>
    _,
171         sum2_lo = inout(reg) sum2_lo => _, sum2_hi = inout(reg) sum2_hi => _,
172         a1 = in(reg) a.0, a2 = in(reg) a.1,
173         b1 = in(reg) b.0, b2 = in(reg) b.1,
174         m_mask1 = in(reg) m_mask_1, m_mask2 = in(reg) m_mask_2,
175         m_bits_2 = in(reg) m_bits_2 as u64,
176         tmp = out(reg) _, sump = out(reg) _,
177         q = const MAX_Q_U64 as u64, p = in(reg) P,
178         options(readonly, nostack)
179         )
180     }
181     result
182 }
```

Code 2.22: Inline AArch64 assembly strong universal variable length string hash function implementation.

Figures 2.11 and 2.12 were generated using the following parameters: both $a = 0x1\,A34F\,98C1\,2B5E\,71D0\,E4A3$, both $b = 0xC51A\,2938\,B47D\,6E5F\,1029$, both $c = 0x7F5B\,3D1E\,8A92\,C640\,B5E1$, and $m_{bits} = 20$ with `opt-level = 3`.

As expected, the first Rust implementation has the worst performance since it uses operations with 256-bit operands. The performance gap between the second Rust implementation and the AArch64 implementation is due to the conversion of the vector from `u8` to `u32`.

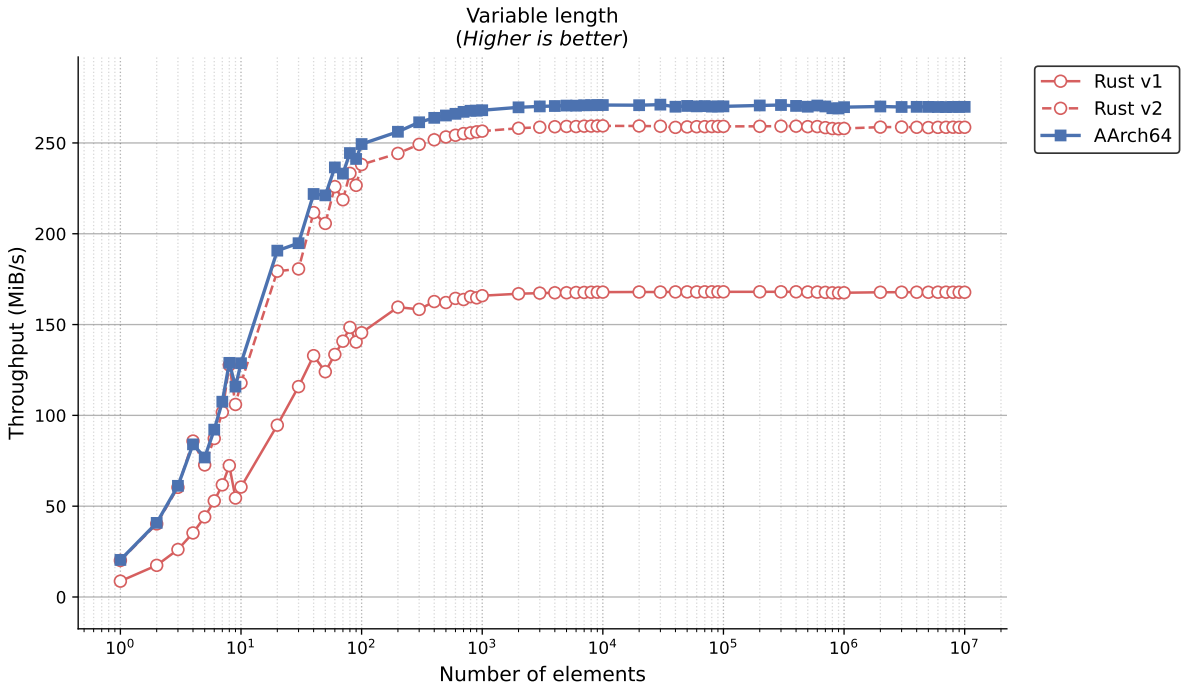

Figure 2.11: Strong universal variable length implementations' throughput.

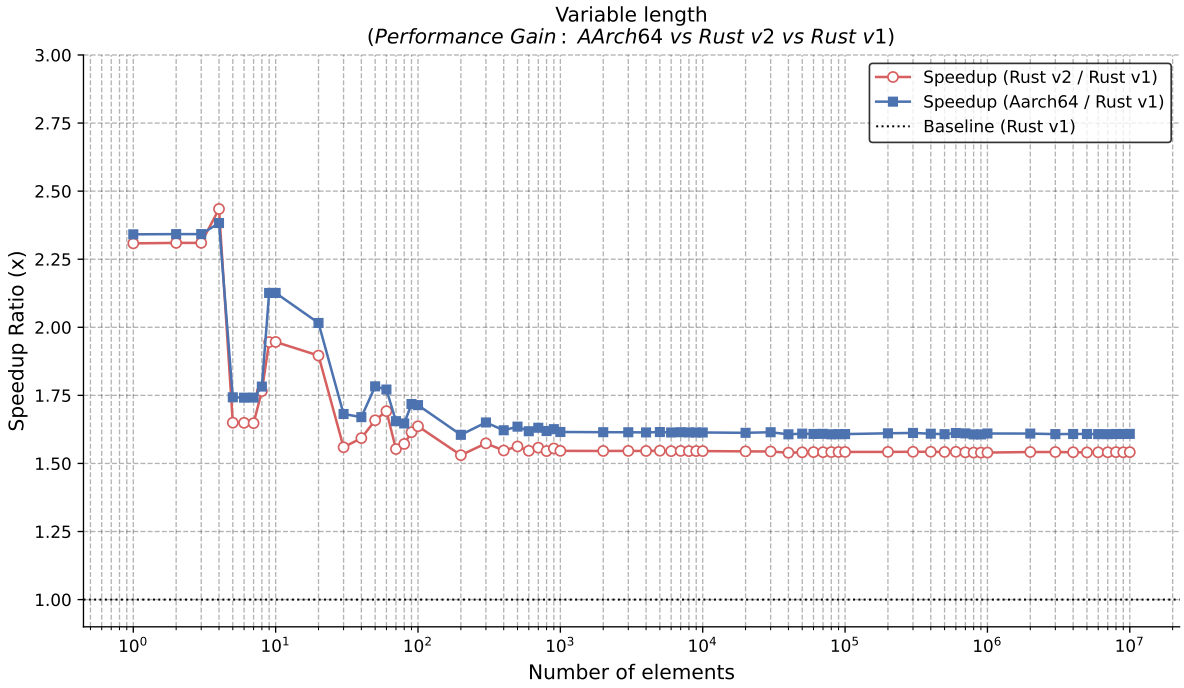

Figure 2.12: Strong universal variable length implementations' delta graph.

Chapter 3

Rust HashMap with custom hashing functions

Universal and strong universal hash functions are used in various applications: the former are commonly used in message authentication [12] [13], while the latter are widely used in the field of Big Data, such as coordinated sampling [3].

In this chapter, the most efficient hash function implemented in the previous section will be used to construct a custom hasher that can be used within Rust's `HashMap` and `HashSet` standard collections. After that, the `HashMap` with our custom hashers will be benchmarked against other available hashmaps:

- Standard Rust `HashMap` [14]: This hashmap uses the *SipHash 1-3* hashing algorithm, which is extremely resistant to HashDoS attacks. This protection comes at a higher computational complexity. In fact, it is recommended to use other hashmaps when such protection is not required.
- Rust's compiler hashmap [15]: HashDoS protection is not necessary during compile time. Therefore, the Rust compiler uses the *Fx* non-cryptographic hash algorithm.
- `fnv` [16]: This hashmap implements the *Fowler–Noll–Vo hash function*, which is tested to be more efficient for smaller hash keys [17].
- `ahash` [18]: This hashmap implements the fastest, DoS resistant hash currently available in Rust.
- `foldhash` [19]: This hashmap uses *foldhash*, a fast, non-cryptographic, minimally DoS-resistant hashing algorithm.
- `xxhash` [20]: This hashmap uses the *xxhash* extremely fast non-cryptographic algorithm.

- `nohash` [21]: The simplest hashmap for integers only. It uses the integer itself as hash value.

3.1 Implementing the hasher

To implement a custom hasher in Rust, a struct must be defined to encapsulate its internal state. The struct must then implement two traits:

- `BuildHasher`: Defines how the hasher instance is constructed, acting as a factory for the collections.
- `Hasher`: Defines the internal state mutation when data is fed into the hasher, and how the final hash value is computed.

The `Hasher` trait requires two primary methods: `write()`, to ingest the byte stream, and `finish()`, to compute and retrieve the final hash. While there are many `write()` methods, one for each primitive type, only one is mandatory to implement (`write(&mut self, bytes: &[u8])`), as all the other methods fall back to it by default.

For integer hash functions, the generic `write(&mut self, bytes: &[u8])` method will be disabled by panicking and the specific functions for each type will be used instead. Since the implementation of `write()` is repetitive for each integer hash function, the following Rust macro is used:

```
1     #[macro_export]
2     macro_rules! impl_hasher_for_int {
3         () => {
4             #[inline]
5             fn write(&mut self, _: &[u8]) {
6                 panic!("Invalid use of Hasher. Only write_u* and write_i*
    methods are supported.")
7             }
8
9             #[inline]
10            fn write_u8(&mut self, n: u8) {
11                self.num = u64::from(n)
12            }
13            #[inline]
14            fn write_u16(&mut self, n: u16) {
15                self.num = u64::from(n)
16            }
```

```
17              #[inline]
18              fn write_u32(&mut self, n: u32) {
19                  self.num = u64::from(n)
20              }
21              #[inline]
22              fn write_u64(&mut self, n: u64) {
23                  self.num = n
24              }
25              #[inline]
26              fn write_usize(&mut self, n: usize) {
27                  self.num = n as u64
28              }
29
30              #[inline]
31              fn write_i8(&mut self, n: i8) {
32                  self.num = n as u64
33              }
34              #[inline]
35              fn write_i16(&mut self, n: i16) {
36                  self.num = n as u64
37              }
38              #[inline]
39              fn write_i32(&mut self, n: i32) {
40                  self.num = n as u64
41              }
42              #[inline]
43              fn write_i64(&mut self, n: i64) {
44                  self.num = n as u64
45              }
46              #[inline]
47              fn write_isize(&mut self, n: isize) {
48                  self.num = n as u64
49              }
50          };
51      }
```

Code 3.1: Macro for implementing `write()` for integers only.

When implementing the trait with the variable length string hash function, illustrated in Chapter 2.3.2, the `write(&mut self, bytes: &[u8])` method will

invoke `variable_length_horner()`, while the `finish(&self)` method will call `variable_length_finish()`.

3.2 Comparison with other hash functions

The benchmark used to measure the performance of the hashmap with different hashers consists of counting the frequency of elements within a vector of randomly generated integers. The structure of the code that is going to be measured is:

```
// inputs, the integers vector, is defined before the measurements
let mut hashmap = <hashmap creation>;
for &input in &inputs {
    *hashmap.entry(input).or_insert(0) += 1;
}
black_box(hashmap) // Avoid dead code elimination
```

Code 3.2: Example of code used for the benchmark.

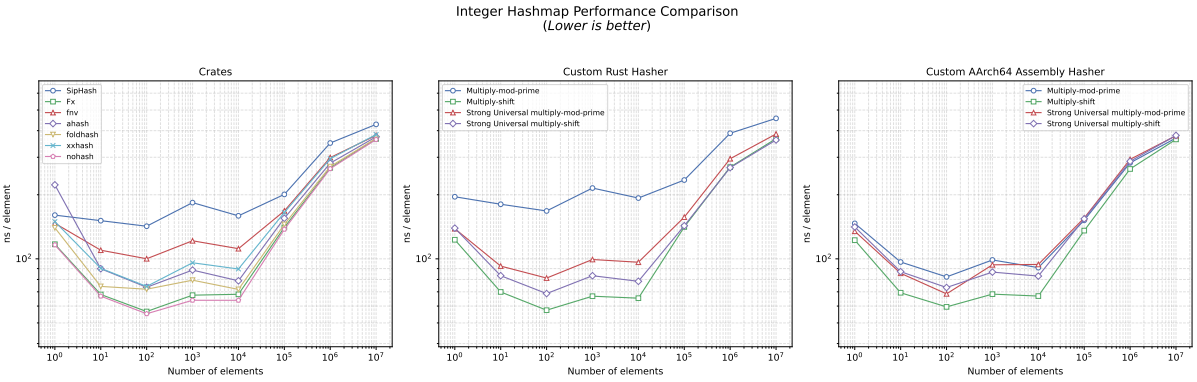

Figure 3.1: Performance of hashmaps with various hashers.

Figure 3.1 illustrates the result of the benchmark. Notably, the `SipHash 1-3` and the Rust implementation of a multiply-mod-prime hash function stand out among the others due to their complexity, caused by HashDoS resistance and the use of `ethnum` and a 128-bit modulo operation respectively. Table 3.1 details the exact execution times. For each column, the best hashmap is highlighted in bold.

The `nohash` hashmap outperforms the other hashers because the vector's elements, randomly generated, are uniformly distributed. If the input data were highly clustered, the `nohash` hashmap would suffer from frequent collisions, causing performance to degrade to $\mathcal{O}(n)$. Despite this, beyond 10^5 elements, the universal and strong universal multiply-shift hashers achieve lower

execution times, demonstrating their efficiency despite their higher baseline computational complexity.

	Vector length							
Hasher	1	10^1	10^2	10^3	10^4	10^5	10^6	10^7
SipHash 1-3	$160.47 \cdot 10^{-9}$	$1.5110 \cdot 10^{-6}$	$14.230 \cdot 10^{-6}$	$183.70 \cdot 10^{-6}$	$1.5924 \cdot 10^{-3}$	$20.077 \cdot 10^{-3}$	$350.00 \cdot 10^{-3}$	4.2825
Fx	$117.11 \cdot 10^{-9}$	$680.09 \cdot 10^{-9}$	$5.6566 \cdot 10^{-6}$	$67.511 \cdot 10^{-6}$	$681.46 \cdot 10^{-6}$	$14.050 \cdot 10^{-3}$	$267.65 \cdot 10^{-3}$	3.6600
fnv	$146.87 \cdot 10^{-9}$	$1.0979 \cdot 10^{-6}$	$10.008 \cdot 10^{-6}$	$121.56 \cdot 10^{-6}$	$1.1161 \cdot 10^{-3}$	$16.791 \cdot 10^{-3}$	$299.06 \cdot 10^{-3}$	3.8016
ahash	$222.61 \cdot 10^{-9}$	$898.36 \cdot 10^{-9}$	$7.3544 \cdot 10^{-6}$	$88.552 \cdot 10^{-6}$	$788.53 \cdot 10^{-6}$	$15.514 \cdot 10^{-3}$	$281.49 \cdot 10^{-3}$	3.7431
foldhash	$139.90 \cdot 10^{-9}$	$742.22 \cdot 10^{-9}$	$7.1994 \cdot 10^{-6}$	$79.389 \cdot 10^{-6}$	$718.52 \cdot 10^{-6}$	$14.503 \cdot 10^{-3}$	$271.32 \cdot 10^{-3}$	3.6799
xxhash	$149.55 \cdot 10^{-9}$	$902.63 \cdot 10^{-9}$	$7.4214 \cdot 10^{-6}$	$95.829 \cdot 10^{-6}$	$896.02 \cdot 10^{-6}$	$16.502 \cdot 10^{-3}$	$295.38 \cdot 10^{-3}$	3.8357
nohash	$\mathbf{116.31 \cdot 10^{-9}}$	$\mathbf{667.94 \cdot 10^{-9}}$	$\mathbf{5.5211 \cdot 10^{-6}}$	$\mathbf{63.881 \cdot 10^{-6}}$	$\mathbf{638.62 \cdot 10^{-6}}$	$13.765 \cdot 10^{-3}$	$265.10 \cdot 10^{-3}$	3.6358
Multiply-mod-prime (Rust)	$195.70 \cdot 10^{-9}$	$1.8048 \cdot 10^{-6}$	$16.785 \cdot 10^{-6}$	$215.06 \cdot 10^{-6}$	$1.9300 \cdot 10^{-3}$	$23.453 \cdot 10^{-3}$	$388.98 \cdot 10^{-3}$	4.5691
Multiply-mod-prime (AArch64)	$146.76 \cdot 10^{-9}$	$966.36 \cdot 10^{-9}$	$8.2270 \cdot 10^{-6}$	$98.788 \cdot 10^{-6}$	$909.29 \cdot 10^{-6}$	$15.177 \cdot 10^{-3}$	$282.43 \cdot 10^{-3}$	3.6936
Multiply-shift (Rust)	$122.92 \cdot 10^{-9}$	$698.95 \cdot 10^{-9}$	$5.7387 \cdot 10^{-6}$	$66.752 \cdot 10^{-6}$	$653.47 \cdot 10^{-6}$	$14.117 \cdot 10^{-3}$	$269.73 \cdot 10^{-3}$	3.6676
Multiply-shift (AArch64)	$122.42 \cdot 10^{-9}$	$692.85 \cdot 10^{-9}$	$5.9470 \cdot 10^{-6}$	$68.265 \cdot 10^{-6}$	$668.59 \cdot 10^{-6}$	$\mathbf{13.569 \cdot 10^{-3}}$	$\mathbf{264.33 \cdot 10^{-3}}$	3.6405
Strong universal multiply-mod-prime (Rust)	$138.52 \cdot 10^{-9}$	$924.57 \cdot 10^{-9}$	$8.1225 \cdot 10^{-6}$	$99.283 \cdot 10^{-6}$	$963.93 \cdot 10^{-6}$	$15.699 \cdot 10^{-3}$	$295.23 \cdot 10^{-3}$	3.8587
Strong universal multiply-mod-prime (AArch64)	$134.75 \cdot 10^{-9}$	$855.29 \cdot 10^{-9}$	$6.8466 \cdot 10^{-6}$	$93.731 \cdot 10^{-6}$	$941.22 \cdot 10^{-6}$	$15.550 \cdot 10^{-3}$	$293.35 \cdot 10^{-3}$	3.7882
Strong universal multiply-shift (Rust)	$139.38 \cdot 10^{-9}$	$833.37 \cdot 10^{-9}$	$6.8676 \cdot 10^{-6}$	$83.311 \cdot 10^{-6}$	$784.29 \cdot 10^{-6}$	$14.298 \cdot 10^{-3}$	$268.20 \cdot 10^{-3}$	**3.6166**
Strong universal multiply-shift (AArch64)	$141.17 \cdot 10^{-9}$	$870.51 \cdot 10^{-9}$	$7.3334 \cdot 10^{-6}$	$86.646 \cdot 10^{-6}$	$829.84 \cdot 10^{-6}$	$15.408 \cdot 10^{-3}$	$287.07 \cdot 10^{-3}$	3.7987

Table 3.1: Exact execution time for each hashmap in seconds.

To evaluate the performance of hashmaps on string keys, the *Sentiment140* [22] dataset was used. The dataset, comprising 1.6 million tweets and their polarity, was used to compute the mean polarity of every word. The core logic of the code used in the benchmark is:

```
// words is a vector of tuples (usize, String), where usize represents the
    polarity of the tweet contained in the String.
let mut hashmap = <hashmap creation>;
for (sentiment, word) in &words {
    let entry = hashmap.entry(word).or_insert((0, 0));
    entry.0 += sentiment;
    entry.1 += 1;
}
let mut averages = <hashmap creation>;
for (word, (sum, count)) in hashmap {
    let average = sum as f64 / count as f64;
    averages.insert(word, average);
}
black_box(averages) // Avoid dead code elimination
```

Code 3.3: Example of code used for the benchmark.

As shown in Figure 3.2, our implementation of a hash function for strings is suboptimal for this specific workload. The algorithm is designed for longer data streams, whereas the average word length in a tweet is exceptionally short. Furthermore, other hashers use the byte stream directly, without chunking the `u8` integers, while using CPU registers or hardcoded parameters.

Performance would likely not improve significantly even if a bounded-length hash function implementation were used. The implementation would be complicated because of the separation

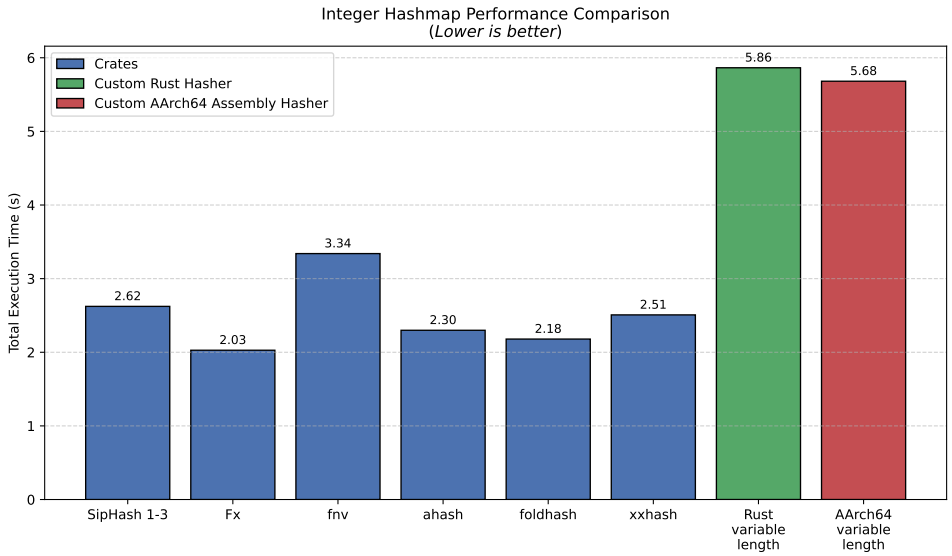

Figure 3.2: Execution time for determining the mean polarity of words.

between data ingestion (`write()`) and hash computation (`finish()`): after creating the vector of parameters, an index would be needed to point to the first unused integer within the parameter vector. Furthermore, each chunk of integers would require two fetches, one for each parameter vector, significantly increasing the memory access overhead.

Chapter 4

Conclusion

This thesis has shown that AArch64 assembly implementations generally offer superior performance compared to their Rust counterparts. Specifically, the use of inline assembly is beneficial in scenarios where:

- There is overhead in Rust's safety guarantees: Sometimes, Rust's safety system introduces an invisible overhead that makes the program run slower than equivalents written in lower-level programming languages. For this exact reason, the `unsafe {}` block exists as a way to bypass these safety checks. Performance can be further improved with the `asm!()` macro, at the cost of platform dependency.
- There are type conversions: High-level programming languages can lose performance during type casting or conversion. For instance, in the implementation of a bounded length and variable length string hash function in Chapters 2.3.1 and 2.3.2, the AArch64 assembly implementation is faster primarily because it avoids the type conversion from `&[u8]` to `&[u64]`.

The use of inline assembly is not recommended and can be counterproductive when:

- The function is computationally simple: A pure inline assembly implementation can be slower than the Rust equivalent when there are constant parameters. In this case, the Rust compiler is able to perform constant folding and, in some cases, strength reduction, thus generating assembly code that is more optimized than manual implementations. With a hybrid approach, as seen in both multiply-shift implementations in Chapters 2.1.2 and 2.2.2, it is possible to achieve the benefits of fast assembly code with Rust's compiler strength reduction when possible.
- The system is memory-bound: For instance, in the vector multiply-shift and pair-multiply-shift implementations in Chapters 2.2.3 and 2.2.4, the computational benefits of the assembly implementations are quickly overshadowed by the memory latency.

Bibliography

[1] NASA, *New nasa Supercomputer Shows How Dust Fills Invisible Spider Webs Around Stars*, https://web.archive.org/web/20160403061211/http://www.nasa.gov/centers/goddard/news/releases/2010/10-051.html, Accessed via the Wayback Machine; original link: http://www.nasa.gov/centers/goddard/news/releases/2010/10-051.html, 2010.

[2] A. L. Morris, "The design and performance of the ATLAS Inner Detector Trigger, and a search for Extra Dimensions in the diphoton final state", Ph.D. dissertation, Southampton U., Southampton, 2013. [Online]. Available: https://repository.cern/records/syzx7-tzm59

[3] M. Thorup, *High speed hashing for integers and strings*, 2020. arXiv: 1504.06804 [cs.DS]. [Online]. Available: https://arxiv.org/abs/1504.06804

[4] The Rust Project Developers, *Rust programming language*, 2024. [Online]. Available: https://www.rust-lang.org

[5] *Criterion*, https://crates.io/crates/criterion.

[6] *Proptest*, https://crates.io/crates/proptest, 2024.

[7] T. Jamil, "Risc versus cisc", *Ieee Potentials*, vol. 14, no. 3, pp. 13–16, 1995.

[8] L. Ryzhyk, "The arm architecture", *Chicago University, Illinois, EUA*, vol. 1, 2006.

[9] *Ethnum*, https://crates.io/crates/ethnum.

[10] *Num-bigint*, https://crates.io/crates/num-bigint.

[11] G Sridevi, M. Ramakrishna, and D Ashoka, "Comprehensive performance study of hashing functions", *Computer Science Journal of Moldova*, vol. 92, no. 2, pp. 183–199, 2023.

[12] M. Etzel, S. Patel, and Z. Ramzan, "Square hash: Fast message authentication via optimized universal hash functions", in *Annual International Cryptology Conference*, Springer, 1999, pp. 234–251.

[13] K. Yuksel, J.-P. Kaps, and B. Sunar, “Universal hash functions for emerging ultra-low-power networks”, in *Proceedings of the communications networks and distributed systems modeling and simulation conference*, 2004.

[14] The Rust Project Developers, *Struct std::collections::hashmap*, `https://doc.rust-lang.org/std/collections/struct.HashMap.html`, 2026.

[15] The Rust Project Developers, *Rustc-hash: A speedy, non-cryptographic hash used within rustc*, `https://crates.io/crates/rustc-hash`, 2026.

[16] A. Crichton, *An implementation of the fowler–noll–vo hash function*, `https://crates.io/crates/fnv`, 2026.

[17] L. C. Noll, *The Fowler-Noll-Vo (FNV) hash function*, `http://www.isthe.com/chongo/tech/comp/fnv/index.html`, 2024.

[18] T. Kaitchuck, *A non-cryptographic hash function using aes-ni for high performance*, `https://crates.io/crates/ahash`, 2026.

[19] O. Peters, *A fast, non-cryptographic, minimally dos-resistant hashing algorithm*, `https://crates.io/crates/foldhash`, 2026.

[20] Shepmaster, *A rust implementation of the xxhash and xxh3 algorithms*, `https://crates.io/crates/twox-hash`, 2026.

[21] S. Glavina, *An implementation of ‘std::hash::hasher‘ which does not hash at all*, `https://crates.io/crates/nohash`, 2026.

[22] kazanova, *Sentiment140 dataset with 1.6 million tweets*, `https://www.kaggle.com/datasets/kazanova/sentiment140`, 2017.