GxHash: A High-Throughput, Non-Cryptographic Hashing Algorithm Leveraging Modern CPU Capabilities

Olivier Giniaux

Abstract

In the rapidly evolving landscape of data processing and cybersecurity, hashing algorithms play a pivotal role in ensuring data integrity and security. Traditional hashing methods, while effective, often fail to fully utilize the computational capabilities of modern processors. This paper introduces the GxHash hashing algorithm, a novel approach that harnesses the power of high instruction-level parallelism (ILP) and Single Instruction, Multiple Data (SIMD) capabilities of contemporary CPUs to achieve high-throughput non-cryptographic hashing. Through a comprehensive analysis, including benchmarks and comparisons with existing methods, we demonstrate that GxHash significantly outperforms conventional algorithms in terms of speed and computational efficiency without compromising on security. The paper also explores the implications, limitations, and avenues for future research in this burgeoning field.

Contents

1 Introduction **2**
- 1.1 Motivations . . . 2
- 1.2 Scope Limitation . . . 2
  - 1.2.1 Portability . . . 2

2 Related Work **3**
- 2.1 The Merkle–Damgård Construction . . . 3
- 2.2 The Wide-Pipe Construction Variant . . . 3
- 2.3 Other Variants . . . 4
  - 2.3.1 Sponge Construction . . . 4
  - 2.3.2 HAIFA Construction . . . 4
  - 2.3.3 Tree Hashing . . . 4

3 High ILP Construction **6**
- 3.1 ILP Awareness . . . 6
  - 3.1.1 Example . . . 6
  - 3.1.2 Benchmark . . . 7
- 3.2 The Laned Construction . . . 8
  - 3.2.1 Intermediate Hashes . . . 8
  - 3.2.2 Final Hash . . . 8

4 The GxHash Algorithm **9**
- 4.1 Pipe Width . . . 9
- 4.2 Compression . . . 9
- 4.3 Finalization . . . 10
  - 4.3.1 Mixing . . . 10
  - 4.3.2 Folding . . . 10
- 4.4 Implementation Details . . . 11
  - 4.4.1 CPU Alignement . . . 11
  - 4.4.2 Low-Overhead Looping . . . 11
  - 4.4.3 Padding . . . 11

5 Benchmarks **12**
- 5.1 Quality . . . 12
  - 5.1.1 Benchmark Quality Criteria . . . 12
  - 5.1.2 Quality Criteria Results . . . 13
- 5.2 Performance . . . 13

6 Discussion **13**
- 6.1 Implications . . . 13
- 6.2 Limitations . . . 13
- 6.3 Future Work . . . 13

7 Conclusion **13**

1 Introduction

1.1 Motivations

As a software engineer at Equativ, a company specializing in high-performance AdServing backends that handle billions of auctions daily, I face unique challenges in maximizing throughput while minimizing latency. In this high-stakes environment, every millisecond counts, and the performance of underlying data structures becomes critically important. We heavily rely on in-memory caches and other hash-based data structures, making the efficiency of hashing algorithms a non-trivial concern in our system's overall performance.

While diving into the theory of hashing out of both necessity and intellectual curiosity, I discovered that existing hashing algorithms, including those built on well-known constructions like Merkle–Damgård, are not optimized to exploit the full capabilities of modern general-purpose CPUs. These CPUs offer advanced features such as Single Instruction, Multiple Data (SIMD) and Instruction-Level Parallelism (ILP), which remain largely untapped by current hashing methods.

The challenge of creating a faster, more efficient hashing algorithm became not just a professional necessity but also a personal quest. It was both challenging and exhilarating to delve into hashing theory and experiment with new approaches. The result is what I believe to be the fastest non-cryptographic hashing algorithm developed to date.

The primary motivation behind this research is to bridge the existing performance gap by designing a hashing algorithm that fully leverages SIMD and ILP capabilities. The aim is to achieve an order-of-magnitude improvement in hashing speed, thereby revolutionizing the efficiency of various applications, from databases to real-time analytics and beyond.

In summary, this work is driven by both the practical needs of my professional environment and a personal passion for pushing the boundaries of what is technically possible in the realm of hashing algorithms.

1.2 Scope Limitation

This paper primarily investigates methods for enhancing the throughput of non-cryptographic hashing algorithms, which are commonly used in a variety of time-sensitive applications, including but not limited to hashmap lookups. In such contexts, hash computation is often expected to be extremely fast, typically requiring execution times ranging from nanoseconds to microseconds. Given these considerations, the paper intentionally excludes Thread-Level Parallelism (TLP) as a viable strategy for performance optimization.

Firstly, it's worth mentioning that TLP is orthogonal to the methods being explored in this paper. TLP focuses on optimizing coarser-grained operations by distributing tasks across multiple threads, which doesn't directly align with the fine-grained performance improvements we aim to achieve.

Secondly, introducing TLP would necessitate the incorporation of task scheduling, synchronization, and context switching. These overheads, while perhaps manageable in applications that can afford greater latencies, become less justifiable when aiming for high-throughput, low-latency operations that are typical in non-cryptographic hashing scenarios.

In summary, while TLP might be a suitable performance optimization strategy in other computational contexts, we chose to leave it aside in this paper.

1.2.1 Portability

2 Related Work

The field of hashing algorithms has been a subject of extensive research and development, tracing its roots back to foundational architectures like the **Merkle–Damgård** Construction[5][2], introduced in 1989. Over the years, this area has seen a plethora of innovations, each attempting to address various aspects of hashing—be it collision resistance, distribution uniformity, or computational efficiency. While cryptographic hashing has often been the focal point of research, non-cryptographic hashing algorithms have also garnered attention for their utility in data structures like hashmaps and caches.

In this section, we will explore some of the seminal works and recent advancements in the realm of hashing algorithms to contextualize our research.

2.1 The Merkle–Damgård Construction

The Merkle–Damgård construction serves as the foundational architecture for many existing hash functions. It operates by breaking down an input into fixed-size blocks, which are then processed sequentially through a compression function. The output of each block feeds into the next, culminating in a final, fixed-size hash.

To formalize it, let us denote a hash function $h : \{0,1\}^{n_b \times s_b} \to \{0,1\}^{s_h}$, where:

- M represents an input message that is divided into n_b blocks, each of s_b bits. In formal notation, $M = M_1 \parallel M_2 \parallel \cdots \parallel M_{n_b}$.
- The output is a hash value with a fixed bit-length s_h.

Before proceeding to the formal definition, we need to establish the key functional components that constitute the Merkle–Damgård architecture:

- A compression function $f : \{0,1\}^{s_b} \times \{0,1\}^{s_b} \to \{0,1\}^{s_b}$,
- A finalization function $g : \{0,1\}^{s_b} \to \{0,1\}^{s_h}$,
- An initialization vector 0^{s_b}, comprised of s_b zero bits,

With these components, the hash function h may be articulated as follows:

$$h(M) = g\left(f(\cdots f(f(0^{s_b}, M_1), M_2) \cdots, M_{n_b})\right)$$

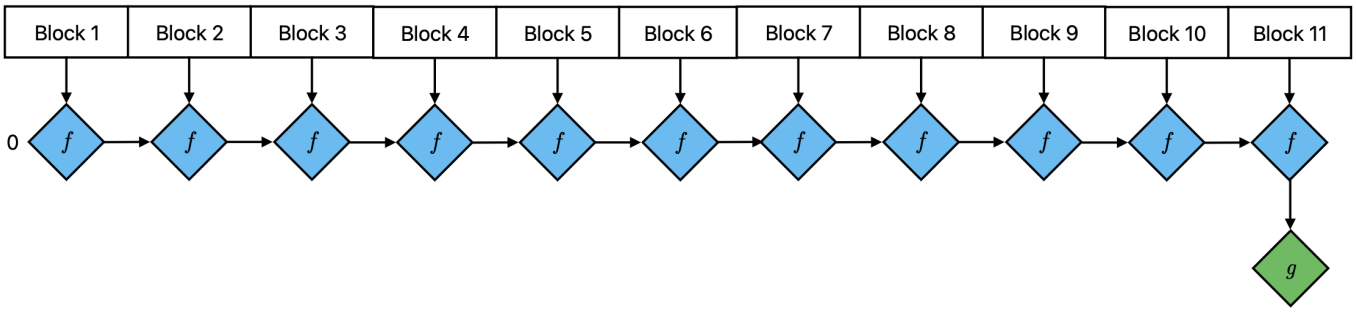

Figure 1: Merkle–Damgård Construction Overview

2.2 The Wide-Pipe Construction Variant

The **Wide-Pipe** construction serves as a variant of the traditional Merkle–Damgård architecture and aims to improve the hash function's cryptographic resilience.

The standard Merkle–Damgård construction has been found vulnerable to certain types of cryptographic attacks, including length extension and multicollision attacks. The Wide-Pipe variant mitigates these vulnerabilities by modifying the hash function's internal state. In this construction, the size of the internal state (s_b) is deliberately made much larger than the size of the hash output (s_h), as expressed by the inequality:

$$s_b \gg s_h$$

This design choice serves to complicate potential attack vectors by introducing a greater level of computational complexity. As a result, the hash function gains increased resistance against certain types of attacks that exploit the limitations of the Merkle–Damgård construction.

While performance remains our primary focus, the algorithm we examine in this paper incorporates the Wide-Pipe idea. By doing so, it offers a dual advantage: performance efficacy, which is central to this study, along with good cryptographic resilience, which is an undeniable advantage even in the realm of non-cryptographic hash functions.

2.3 Other Variants

While the classical Merkle–Damgård construction provides a reliable framework for cryptographic hash functions, there are a wide number of variants[6]. In this section, we discuss some of the most popular variants around opportunities for performance enhancement.

2.3.1 Sponge Construction

The Sponge construction, proceeds in two phases: absorption and squeezing. During the absorption phase, blocks of input data are XORed into a portion of the internal state, followed by a cryptographic permutation of the entire state. Importantly, each absorption step is inherently sequential, as it relies on the state resulting from the cryptographic permutation of the previous step. Consequently, the algorithm cannot benefit from instruction-level parallelism during the absorption phase. The squeezing phase, which follows the absorption of all input blocks, generates the output hash from the internal state. The opportunity for parallelization with the Sponge construction is thus constrained by its sequential nature during the absorbing phase.

2.3.2 HAIFA Construction

The HAsh Iterative FrAmework (HAIFA) not only mitigates vulnerabilities but also incorporates features like support for a configurable number of rounds and real-time incremental hashing. However, like the standard Merkle–Damgård and Sponge constructions, the HAsh Iterative FrAmework (HAIFA) is also inherently sequential in its processing of message blocks. Each iteration of the compression function in HAIFA depends on the outcome of the previous iteration. This dependency chain means that the construction does not naturally lend itself to instruction-level parallelism (ILP), although other types of optimizations may still be possible depending on the specific implementation and hardware architecture.

2.3.3 Tree Hashing

Tree hashing breaks the input message into smaller fragments and processes them independently in a tree-like structure. While tree-based hashing constructions offer a degree of parallelism, it is essential to recognize their inherent limitations. The extent of parallelization diminishes progressively as we move upward through the tree, owing to the dependencies between higher-level nodes. Here's a breakdown:

- **Leaf Level:** At this level, all blocks are independent, allowing the hashing operations to occur in parallel. If n is the number of leaves, then $\frac{n}{2}$ parallel operations can occur for an even n, or $\frac{n-1}{2} + 1$ for an odd n.
- **Second Level and Above:** The second level requires the hash results from the first level. Each node at the second level depends on two nodes from the level below, effectively halving the potential parallelism. This pattern of diminishing parallelization continues in subsequent levels.
- **Final Level:** At the top of the tree, we are left with a single hash that relies on the two hashes beneath it. This operation is inherently sequential.

Although the tree-based hashing approach has a theoretical advantage in parallelization, practical implementations often face several challenges that can impact efficiency. These issues can make tree hashing hardly as efficient as a sequential structure for certain applications. Here are the primary factors:

- **Synchronization Overhead:** The parallel nature of the algorithm necessitates synchronization between different processing threads or units, especially at higher tree levels where dependencies exist. This overhead counters the gains from parallelization.
- **Memory Consumption:** Tree constructions typically require more memory to store intermediate hash values, particularly when branching factors are high. Memory allocation and fragmentation usually impact performance.

- **Cache Efficiency:** Unlike sequential algorithms that can benefit from cache locality, the tree-based approach often has to handle multiple non-contiguous data blocks, potentially leading to cache misses and reduced efficiency.
- **Implementation Complexity:** The algorithmic and data-structure requirements for implementing tree-based hashing are more complex than those of a straightforward sequential hashing algorithm. The increased complexity can introduce more room for errors and maintenance challenges.

In light of these practical challenges, tree-based constructions might not be the best fit given our high performance goals and the architecture of today's general-purpose computers.

3 High ILP Construction

Most modern general-purpose CPUs employ a superscalar architecture which enables Instruction-Level Parallelism (ILP). Minimizing dependencies in an algorithm allows a superscalar processor to execute more instructions concurrently, thus maximizing its inherent parallelism and overall performance. The key limitation for ILP in the Merkle–Damgård construction is the inherent sequential dependency: each block's hash depends on the result of hashing the previous block.

3.1 ILP Awareness

While compilers and CPUs employ various techniques to optimize ILP, their capabilities are often constrained by the inherent data dependencies in the code. It's for this reason that algorithmic design can be pivotal. By structuring algorithms to minimize dependency chains from the outset, we create opportunities for higher ILP that even the most advanced compiler optimizations and CPU features cannot achieve alone. Therefore, algorithmic design that is mindful of ILP can be a game-changer for performance optimization.

3.1.1 Example

Let's take a FNV-like hashing function. The "naive" way to process an array of elements would look like the `baseline` method, as shown in Rust snippet (Figure 2). As we can see, every loop iteration requires the value of `h`, which has been computed the iteration before. Here we have a dependency chain, preventing the compiler from doing any optimization.

To make ILP possible, for the function `temp` we unroll the loop and hash a few inputs altogether, independently from `h`, and mix it once thereafter with `h`. We still have a dependency chain on `h`, but for fewer iterations. The temporary hashes are independent and thus eligible for ILP.

Another track taken for the function `laned` is to unroll the loop and hash on separate lanes, and then mix the lanes together upon exiting the loop. Each lane has its own dependency chain but also on fewer iterations.

```rust
const PRIME: u64 = 0x00000100000001b3;
const OFFSET: u64 = 0xcbf29ce484222325;

#[inline]
fn hash(hash: u64, value: u64) -> u64 {
    (hash ^ value) * PRIME
}

fn baseline(input: &[u64]) -> u64 {
    let mut h = OFFSET;
    let mut i: usize = 0;
    while i < input.len() {
        h = hash(h, input[i]);

        i = i + 1;
    }
    h
}

fn unrolled(input: &[u64]) -> u64 {
    let mut h: u64 = OFFSET;
    let mut i: usize = 0;
    while i < input.len() {
        h = hash(h, input[i]);
        h = hash(h, input[i + 1]);
        h = hash(h, input[i + 2]);
        h = hash(h, input[i + 3]);
        h = hash(h, input[i + 4]);

        i = i + 5;
    }
    h
}
```

```rust
fn temp(input: &[u64]) -> u64 {
    let mut h: u64 = OFFSET;
    let mut i: usize = 0;
    while i < input.len() {
        let mut tmp: u64 = input[i];
        tmp = hash(tmp, input[i + 1]);
        tmp = hash(tmp, input[i + 2]);
        tmp = hash(tmp, input[i + 3]);
        tmp = hash(tmp, input[i + 4]);

        h = hash(h, tmp);

        i = i + 5;
    }
    h
}

fn laned(input: &[u64]) -> u64 {
    let mut h1: u64 = OFFSET;
    let mut h2: u64 = OFFSET;
    let mut h3: u64 = OFFSET;
    let mut h4: u64 = OFFSET;
    let mut h5: u64 = OFFSET;
    let mut i: usize = 0;
    while i < input.len() {
        h1 = hash(h1, input[i]);
        h2 = hash(h2, input[i + 1]);
        h3 = hash(h3, input[i + 2]);
        h4 = hash(h4, input[i + 3]);
        h5 = hash(h5, input[i + 4]);

        i = i + 5;
    }
    hash(hash(hash(hash(h1, h2), h3), h4), h5)
}
```

Figure 2: FNV-like hash functions in Rust

3.1.2 Benchmark

Here are the timing on both an x86 and an ARM CPU. It also includes timing for the function `unrolled`, to show that performance increase comes indeed from ILP and not the loop unrolling itself. We can see that `temp` and `laned` performed equally, leveraging ILP for a significant performance increase over the `baseline`.

CPU	`baseline`	`unrolled`	`temp`	`laned`
AMD Ryzen 5 5625U (x86 64-bit)	92.787 µs	93.047 µs	37.516 µs	37.434 µs
Apple M1 Pro (ARM 64-bit)	125.23 µs	124.42 µs	28.507 µs	30.716 µs

Figure 3: Benchmark timings for ILP example

While the `temp` and `laned` functions won't yield exactly the same hashes as the `baseline`, they serve the same purpose, while being much faster. Both approaches have their pros and cons. As the `laned` explicitly declares n variables for our lanes, this approach is simpler in regards to compiler analysis and is thus more likely to benefit from ILP, regardless of the compiler or programming language used. The following section will delve more in depth into the definition of a **Laned Construction**.

3.2 The Laned Construction

The **Laned Construction** introduces k_b lanes, and processes the message by groups of k_b blocks, so that for a given group each of the k_b block and be processed on its own lane. This way, we have k_b independent dependency chains. The lane hashes can then be compressed altogether thereafter upon exiting the loop.

3.2.1 Intermediate Hashes

Let's define $n_g = \lfloor n_b/k_b \rfloor$ as the number of whole groups of k_b message blocks.
For each lane we compute an intermediate hash, H_i, as follows:

$$
\begin{aligned}
H_1 &= f(\dots f(f(f(0^{s_b}, M_{0k_b+1}), M_{1k_b+1}), M_{2k_b+1}) \dots, M_{n_g+1}) \\
H_2 &= f(\dots f(f(f(0^{s_b}, M_{0k_b+2}), M_{1k_b+2}), M_{2k_b+2}) \dots, M_{n_g+2}) \\
&\vdots \\
H_{k_b} &= f(\dots f(f(f(0^{s_b}, M_{0k_b+k_b}), M_{1k_b+k_b}), M_{2k_b+k_b}) \dots, M_{n_g+k_b})
\end{aligned}
$$

3.2.2 Final Hash

The final hash H is calculated using f to compress the intermediate hashes and the remaining message blocks (if any), which is then passed through g:

$$
h(M) = g\left(f(\dots f(f(\dots f(f(0^{s_b}, H_1), H_2) \dots, H_{k_b}), M_{k_b n_g+1}) \dots, M_{n_b})\right).
$$

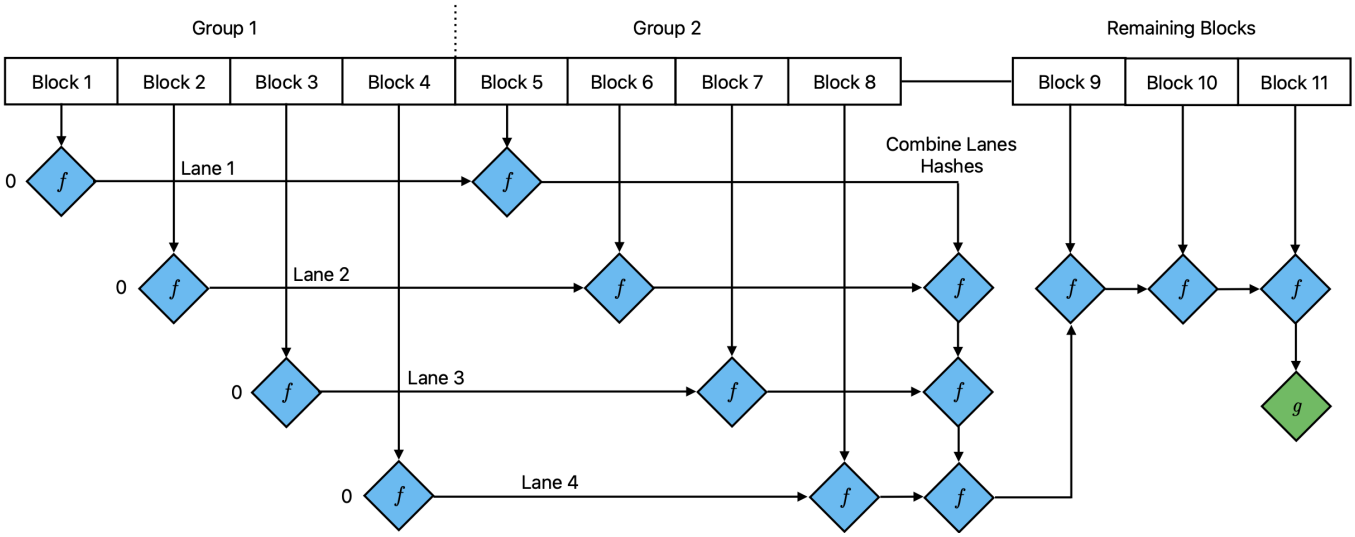

Figure 4: Laned Construction Overview

4 The GxHash Algorithm

The **GxHash** hashing algorithm is designed to maximize throughput by optimizing for Instruction-Level Parallelism (ILP) and making extensive use of Single Instruction, Multiple Data (SIMD) instructions. Notably, while these optimization avenues are orthogonal — focusing on parallelized and vectorized instructions respectively — they collectively harness the full potential of modern CPU architectures.

This design philosophy introduces specific constraints for the compression and finalization functions:

- **Utilization of Hardware Intrinsics:** To achieve the SIMD-oriented goal, arithmetic operations are tailored to be compatible with both x86 and ARM Neon intrinsics.
- **Efficiency through Simplicity:** Minimizing the number of operations is crucial, as fewer operations typically translate to faster execution.
- **Hash Quality Assurance:** Despite these performance optimizations, the algorithm must ensure a minimum level of hash quality to maintain low collision probabilities.

In the next sections, we'll delve into the specific operations and transformations chosen for the compression and finalization functions of the **GxHash-0** (version 0) algorithm.

4.1 Pipe Width

To optimize throughput, the pipe width s_b is set to match the native width of the CPU's SIMD registers. This alignment ensures that each SIMD instruction operates on the entire pipe width, maximizing data processed per instruction. Given that typical SIMD registers are at least 128 bits in width, the pipe width surpasses common 32-bit or 64-bit output hash sizes. Consequently, the GxHash design aligns with the characteristics of a wide-pipe construction, as discussed in section 2.2. This makes GxHash more resistant to multicollision attacks.

4.2 Compression

The role of the compression function is to transform a larger input (or message) into a smaller, fixed-size output. Due to this inherent reduction in size, the compression function cannot be bijective.

To delve deeper into this, consider the definition of a bijective function. A function is bijective if and only if it is both injective (one-to-one) and surjective (onto). In simpler terms, for every unique input, there is a unique output, and every possible output has a corresponding input.

Given the nature of the compression function $f : \{0,1\}^{s_b} \times \{0,1\}^{s_b} \to \{0,1\}^{s_b}$, where the domain is much larger than the codomain ($s_b \times s_b > s_b$), it becomes mathematically impossible for the function to be one-to-one. There will inevitably be multiple different inputs that map to the same output, known as collisions.

With the inevitable non-bijectivity, the performance requirements and the limited set of available SIMD intrinsics, the selection for the compression has to be empirical, thus implying specifying a version to account for these current choices that may be improved in future versions.

In practice, the **GxHash-0** uses a combination of SIMD 8-bit wrapping add and state-wide circular shift of one. The addition is a simple way to mix the state with the next message block bits, while providing an arithmetic carry which helps in regard to distribution (as opposed to a XOR for instance). Adding on a 8-bit basis also helps in regards to distribution, as opposed to adding on larger bit widths. This operation alone however comes with several weaknesses:

- Such a simple operation is inherently weak to different kind of attacks. This can however be partially addressed with a more robust bit mixing for the finalization function. In the context of non-cryptographic hashing, which is the scope of usage of our algorithm, we think it is an acceptable compromise.
- A major issue lies in the addition being associative $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. An associative compression function would make the hashing algorithm insensitive to the ordering of the input message blocks, which is something we want to avoid. To address this, we circularly shift the state bits by one.

While mathematically collisions are inevitable due to the inherent non-bijectivity, odds of collisions using the GxHash-0 compression remain statically low given the uniformity of distribution, although those collisions are predictable due to the inherent simplicity of the function.

```rust
// For ARM 64-bit

use core::arch::aarch64::*;

pub fn compress(a: int8x16_t, b: int8x16_t)
-> int8x16_t {
    let sum: int8x16_t = vaddq_s8(a, b);
    return vextq_s8(sum, sum, 1);
}
```

```rust
// For x86 64-bit

use core::arch::x86_64::*;

pub fn compress(a: __m256i, b: __m256i)
-> __m256i {
    let sum: state = _mm256_add_epi8(a, b);
    return _mm256_alignr_epi8(sum, sum, 1);
}
```

Figure 5: GxHash-0 Compression in Rust

4.3 Finalization

The finalization process in the GxHash-0 algorithm is crucial to ensure the transformation of its internal state into a fixed-size, uniformly distributed hash output. This process is delineated into two primary steps: mixing the bits and folding (reducing) to the desired hash size.

4.3.1 Mixing

This step is responsible for ensuring the even distribution of bits in the state, thereby reducing patterns or biases that might arise from the input data or the compression process. Given the inherent simplicity of the GxHash-0 compression, it is worth for the finalization to incorporate slightly more intricate bit mixing operations, especially given it runs only once per message hashed, as opposed to the compression that occurs once for each block.
Leveraging SIMD capabilities can help in regard to performance and efficiency, which remains for us a primary consideration. Fortunately, both x86 and ARM architectures provide AES (Advanced Encryption Standard) intrinsics that serve as efficient tools for bit mixing. The use of an AES block cipher intrinsics ensures a robust diffusion of bits across the state at a cheap computational cost.
On top of that, salt is added while forming the AES block cipher keys. This not only improves the distribution but also provides a way to use random salt values per-process, protecting from eventual precomputed or replay attack attempts.

```rust
use core::arch::x86_64::*;

pub fn mix(hash: state) -> state {
    // Salt (Knuth primes recommended)
    let salt = _mm256_set_epi64x(
        -4860325414534694371,
        8120763769363581797,
        -4860325414534694371,
        8120763769363581797);

    let keys = _mm256_mul_epu32(salt, hash);
    return _mm256_aesenc_epi128(hash, keys);
}
```

Figure 6: GxHash-0 Mixing in Rust

4.3.2 Folding

Once the state's bits have been thoroughly mixed, the next step is to condense or "fold" this state into a smaller, fixed-size hash output, typically 32 or 64 bits. A straightforward and effective approach taken for GxHash-0 to achieve this reduction is by summing the constituent X-bit integer parts of the mixed state. This summation serves as a reduction function, ensuring that the output hash remains within the desired size bounds while still retaining the essence of the mixed state.

While it would have been interesting to leverage SIMD once again, it turned out that, in practice, for both 128-bit and 256-bit state and for both x86 and ARM, summing the integer parts one by one is as performant if not more performant than using the various SIMD workarounds I tested.

4.4 Implementation Details

4.4.1 CPU Alignement

Data alignment in memory, commonly referred to as CPU alignment, directly impacts the efficiency of data access and processing. The CPU is optimized to access data from addresses that align with its natural word size. When data is properly aligned, the CPU can retrieve and process it in fewer cycles, resulting in increased computational efficiency.

In practice, a program usually allocates memory with some degree of alignment, and so data is generally aligned. However, a given input message to our hash function is still not guaranteed to be aligned. To handle this case, we can either read our data with an offset to account for the misalignment (at the cost of a much-increased complexity) or use specific SIMD intrinsics designed to handle potentially unaligned data.

Benchmarks conducted show a less than 20% performance degradation on both our x86 and ARM hardware when using the second solution. This is the chosen solution for GxHash-0.

4.4.2 Low-Overhead Looping

While the looping semantics will vary from one language to another, any overhead from looping over the input message blocks is likely to directly affect the overall throughput of the algorithm, given how optimized the rest of the algorithm is meant to be. In some cases, compilers might do a great job at generating loop instructions with minimal overhead, but it isn't the case in the language where GxHash-0 was ported, where looping using pointer arithmetics was needed.

Unrolling the loop is a complementary optimization that diminishes greatly any loop overhead. In the case of GxHash, the laned construction implies some kind of unrolling, which is sufficient for achieving the high throughput numbers we see in our benchmarks.

4.4.3 Padding

Merkle–Damgård and derivatives can handle message of an arbitrary size s_m by padding the message upfront with the padding function $p : \{0,1\}^{s_m} \to \{0,1\}^{n_b \times s_b}$ where $n_b = \lceil s_m / s_b \rceil$. In the case where the last block is not whole, the padding fills it with zero-bytes until the size s_b is reached. The last block can then be processed like any other block by the compression function.

In practice, a naive implementation for p for GxHash in computer code implies copying the remaining bytes into a zero-initialized buffer of size s_b, which can then be loaded onto an SIMD registry and then handed to the compression. In our performance-critical context, these allocations and copies have a substantial overhead.

Read beyond and mask. To avoid this overhead, one possible trick consists of reading s_b bytes starting from the last block address, even if it implies reading beyond the memory storing the input message. The read bytes can then be masked with the help of a sliding mask, transforming the trailing bytes that don't belong to our message into zeros, in a single SIMD operation. Compared to the naive method, this solution is up to ten times faster on our test machine (Ryzen 5, x86 64-bit, AVX2).

```rust
use core::arch::x86_64::*;

unsafe fn read_padded(p: *const __m256i, len: isize) -> __m256i {
    // The mask array is twice as long as the SIMD registry size (32*2 here), so that
    // each 32-byte slice accounts for a different number of trailing bytes to mask out
    const MASK: [u8; 64] = [
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ];
    // Loading the mask slice
    let mask = _mm256_loadu_epi8((MASK.as_ptr() as *const i8).offset(32 - len));
    // Read beyond and mask
    return _mm256_and_si256(_mm256_loadu_si256(p), mask);
}
```

Figure 7: GxHash-0 Padding in Rust

Safety considerations. Reading beyond a given pointer can lead to accessing memory that is either not mapped to the program's address space or is protected. When the program tries to read such memory, the operating system detects the violation and typically terminates the program, resulting in a crash. This mechanism protects processes from interfering with each other and from accessing system-critical memory regions. While this is very unlikely to occur in most scenarios, the fact that it can theoretically occur is a problem.

In modern computers, memory is divided into fixed-size chunks called pages. Once a page is given to a program, it can freely access any part of that page without causing system-level errors like segmentation faults. We can take this to our advantage by checking if our unsafe operation is entirely contained in a single block. If so, it means we can use the optimized method safely. Otherwise, we can fallback to the naive method.

In practice, this is quite trivial to implement since memory is divided into pages in such a way that addresses within a single page will share the same higher bits and only vary in the lower bits that represent offsets within that page. The snippet in figure 8 does this considering a minimal page size of 4096. Statistically speaking, the odds of having 32 bytes on the same page are above 99%. This safety check being very cheap to compute, we keep most of the performance advantages of our method while addressing the safety issue.

```rust
unsafe fn is_same_page(ptr: *const __m256i) -> bool {
    // Usual minimal page size on modern computers
    const PAGE_SIZE = 4096;
    // Get the actual pointer address integer value
    let address = ptr as usize;
    // Mask to keep only the last 12 bits (2^12 = 4096)
    let offset_within_page = address & 0xFFF;
    // Check if the 32nd byte from the current offset exceeds the page boundary
    offset_within_page <= (PAGE_SIZE - 31)
}
```

Figure 8: Read-Beyond Safety Check in Rust

5 Benchmarks

5.1 Quality

5.1.1 Benchmark Quality Criteria

The primary quality criteria for non-cryptographic hash functions include:

- **Uniform Distribution:** A high-quality hash function distributes its output values as uniformly as possible across the output space. This ensures that, when used in applications like hash tables, the data is spread evenly, reducing clustering and the frequency of collisions.
- **Minimal Collisions:** While no hash function can be entirely collision-free due to the pigeonhole principle, a good non-cryptographic hash should minimize collisions for typical input sets, ensuring that different inputs usually produce distinct outputs.
- **Avalanche Effect:** A subtle change in the input should result in a considerably different output, ensuring sensitivity to input variations. This also contributes to lessen the risk of clustered hashes in applications like hash tables.
- **Performance:** The performance of a non cryptographic hash function is usually reflected by the performance of the application using it. For instance, a fast non-cryptographic hash function generally implies a fast hash table. This specific criteria will be tackled in the next section which is dedicated to it.

While we can compute quality metrics, the result will greatly vary depending on the actual inputs used for our hash function. Pragmatically, we choose a few patterns for our input data:

- Randomly generated inputs to observe how the hash function behaves with truly unpredictable data
- Sequential inputs to observe how the function handles closely related values. Typically, close values would highlight weaknesses in distribution.
- English words inputs to observe how the function behaves in a "real world scenario"

5.1.2 Quality Criteria Results

For reference, we include a few well-known algorithms on top of GxHash-0.

5.2 Performance

t1ha[4] xxhash[3] HighwayHash[1]

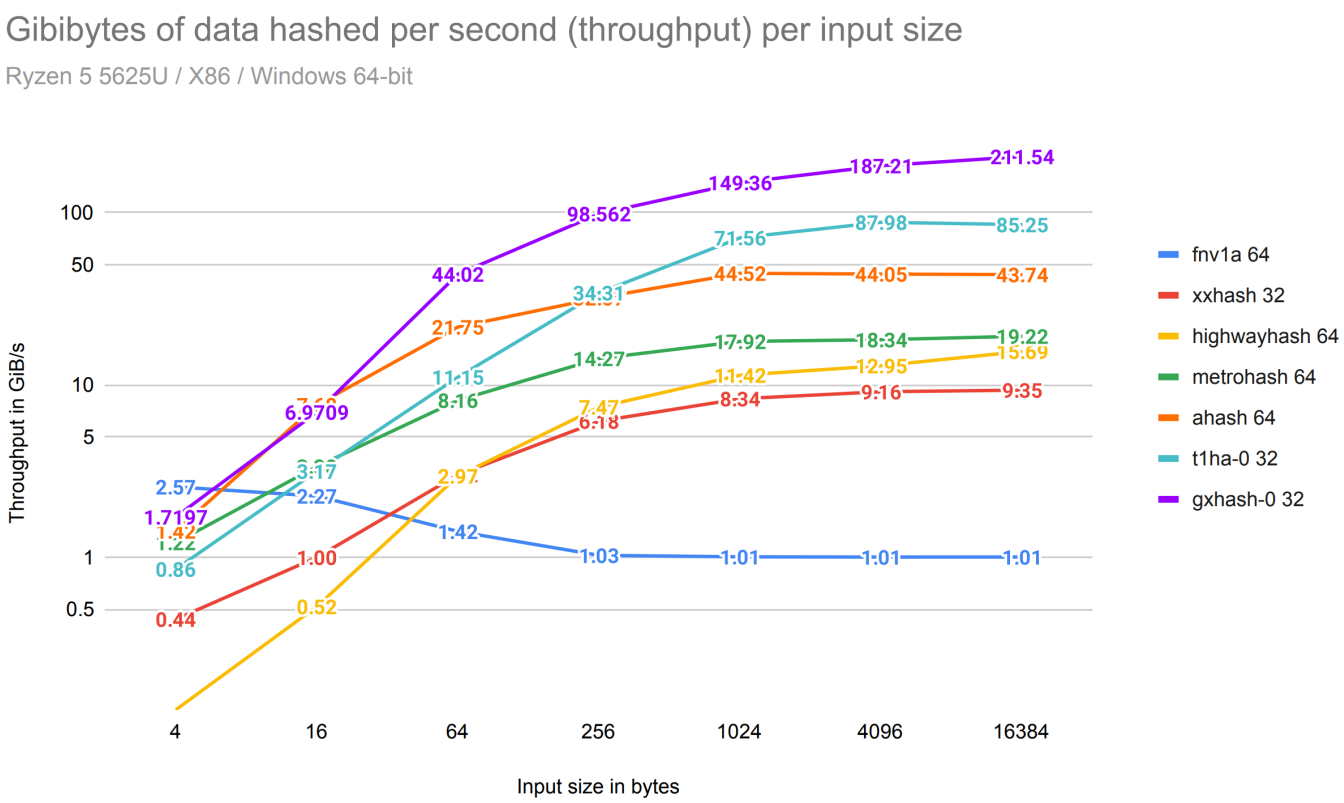

Figure 9: Throughput Benchmark Results

6 Discussion

6.1 Implications

6.2 Limitations

6.3 Future Work

7 Conclusion

References

[1] Nick Babcock. github.com/nickbabcock/highway-rs. v1.1.0.

[2] I. Damgård. A design principle for hash functions. *Crypto'89*, 435:416–427, 1989.

[3] Jake Goulding. github.com/shepmaster/twox-hash. v1.6.3.

[4] Flier Lu. github.com/flier/rust-t1ha. v0.1.0.

[5] R. C. Merkle. One way hash functions and des. *Crypto'89*, 435:428–446, 1989.

[6] Harshvardhan Tiwari. Merkle-damgård construction method and alternatives: A review. *Journal of Information and Organizational Sciences*, 41:283–304, 2017.