CityHash: Fast Hash Functions for Strings

Geoff Pike
(joint work with Jyrki Alakuijala)

Google

http://code.google.com/p/cityhash/
Introduction

- Who?
- What?
- When?
- Where?
- Why?
Outline

Introduction

A Biased Review of String Hashing

Murmur or Something New?

Interlude: Testing

CityHash

Conclusion
Recent activity

- SHA-3 winner was announced last month
- Spooky version 2 was released last month
- MurmurHash3 was finalized last year
- CityHash version 1.1 will be released this month
In my backup slides you can find ...

- My notation
- Discussion of cyclic redundancy checks
  - What is a CRC?
  - What does the \texttt{crc32q} instruction do?
Traditional String Hashing

- Hash function loops over the input
- While looping, the *internal state* is kept in registers
- In each iteration, consume a fixed amount of input
Traditional String Hashing

- Hash function loops over the input
- While looping, the *internal state* is kept in registers
- In each iteration, consume a fixed amount of input
- Sample loop for a traditional byte-at-a-time hash:

```c
for (int i = 0; i < N; i++) {
    state = Combine(state, Bj);
    state = Mix(state);
}
```
Two more concrete old examples (loop only)

\[
\text{for (int } i = 0; i < N; i++) \\
\text{ state } = \rho_{-5}(\text{state}) \oplus B_i
\]
Two more concrete old examples (loop only)

\begin{verbatim}
for (int i = 0; i < N; i++)
    state = \rho_{-5}(state) \oplus B_i

for (int i = 0; i < N; i++)
    state = 33 \cdot state + B_i
\end{verbatim}
// Bob Jenkins circa 1996
int state = 0
for (int i = 0; i < N; i++) {
    state = state + B_i
    state = state + σ_{-10}(state)
    state = state ⊕ σ_6(state)
}
state = state + σ_{-3}(state)
state = state ⊕ σ_{11}(state)
state = state + σ_{-15}(state)
return state
A complete byte-at-a-time example

// Bob Jenkins circa 1996
int state = 0
for (int i = 0; i < N; i++) {
    state = state + B_i
    state = state + σ_{-10}(state)
    state = state ⊕ σ_6(state)
}
state = state + σ_{-3}(state)
state = state ⊕ σ_{11}(state)
state = state + σ_{-15}(state)
return state

What’s better about this? What’s worse?
What Came Next—Hardware Trends

- CPUs generally got better
  - Unaligned loads work well: read words, not bytes
  - More registers
  - SIMD instructions
  - CRC instructions
- Parallelism became more important
  - Pipelines
  - Instruction-level parallelism (ILP)
  - Thread-level parallelism
What Came Next—Hash Function Trends

- People got pickier about hash functions
  - Collisions may be more costly
  - Hash functions in libraries should be “decent”
  - More acceptance of complexity
  - More emphasis on diffusion
Jenkins’ mix

Also around 1996, Bob Jenkins published a hash function with a 96-bit input and a 96-bit output. Pseudocode with 32-bit registers:

\[
\begin{align*}
    a &= a - b; & a &= a - c; & a &= a \oplus \sigma_{13}(c) \\
    b &= b - c; & b &= b - a; & b &= b \oplus \sigma_{-8}(a) \\
    c &= c - a; & c &= c - b; & c &= c \oplus \sigma_{13}(b) \\
    a &= a - b; & a &= a - c; & a &= a \oplus \sigma_{12}(c) \\
    b &= b - c; & b &= b - a; & b &= b \oplus \sigma_{-16}(a) \\
    c &= c - a; & c &= c - b; & c &= c \oplus \sigma_{5}(b) \\
    a &= a - b; & a &= a - c; & a &= a \oplus \sigma_{3}(c) \\
    b &= b - c; & b &= b - a; & b &= b \oplus \sigma_{-10}(a) \\
    c &= c - a; & c &= c - b; & c &= c \oplus \sigma_{15}(b)
\end{align*}
\]
Jenkins’ *mix*

Also around 1996, Bob Jenkins published a hash function with a 96-bit input and a 96-bit output. Pseudocode with 32-bit registers:

\[
\begin{align*}
    a &= a - b; & a &= a - c; & a &= a \oplus \sigma_{13}(c) \\
    b &= b - c; & b &= b - a; & b &= b \oplus \sigma_{-8}(a) \\
    c &= c - a; & c &= c - b; & c &= c \oplus \sigma_{13}(b) \\
    a &= a - b; & a &= a - c; & a &= a \oplus \sigma_{12}(c) \\
    b &= b - c; & b &= b - a; & b &= b \oplus \sigma_{-16}(a) \\
    c &= c - a; & c &= c - b; & c &= c \oplus \sigma_{5}(b) \\
    a &= a - b; & a &= a - c; & a &= a \oplus \sigma_{3}(c) \\
    b &= b - c; & b &= b - a; & b &= b \oplus \sigma_{-10}(a) \\
    c &= c - a; & c &= c - b; & c &= c \oplus \sigma_{15}(b)
\end{align*}
\]

Thorough, but pretty fast!
Jenkins’ *mix*-based string hash

Given $mix(a, b, c)$ as defined on the previous slide, pseudocode for string hash:

```c
uint32 a = ...
uint32 b = ...
uint32 c = ...
int iters = \lfloor N/12 \rfloor
for (int i = 0; i < iters; i++) {
    a = a + \overline{W}_{3i}
    b = b + \overline{W}_{3i+1}
    c = c + \overline{W}_{3i+2}
    mix(a, b, c)
}
```

etc.
Modernizing Google’s string hashing practices

- Until recently, most string hashing at Google used Jenkins’ techniques
  - Some in the “32-bit” style
  - Some in the “64-bit” style, whose mix is $4/3$ times as long
- We saw Austin Appleby’s 64-bit Murmur2 was faster and considered switching
Modernizing Google’s string hashing practices

- Until recently, most string hashing at Google used Jenkins’ techniques
  - Some in the “32-bit” style
  - Some in the “64-bit” style, whose mix is 4/3 times as long
- We saw Austin Appleby’s 64-bit Murmur2 was faster and considered switching
- Launched education campaign around 2009
  - Explain the options; give recommendations
  - Encourage labelling: “may change” or “won’t”
Quality targets for string hashing

There are roughly four levels of quality one might seek:

- quick and dirty
- suitable for a library
- suitable for *fingerprinting*
- secure
Quality targets for string hashing

There are roughly four levels of quality one might seek:

- quick and dirty
- suitable for a library
- suitable for fingerprinting
- secure

Is Murmur2 good for a library? for fingerprinting? both?
First define two subroutines:

\[ \text{ShiftMix}(a) = a \oplus \sigma_{47}(a) \]
Murmur2 preliminaries

First define two subroutines:

\[ \text{ShiftMix}(a) = a \oplus \sigma_{47}(a) \]

and

\[ \text{TailBytes}(N) = \sum_{i=1}^{N \mod 8} 256^{(N \mod 8) - i} \cdot B_{N - i} \]
uint64 k = 14313749767032793493
int iters = \lceil N/8 \rceil
uint64 hash = seed \oplus Nk
for (int i = 0; i < iters; i++)
    hash = (hash \oplus (ShiftMix(W_i \cdot k) \cdot k)) \cdot k

if (N \mod 8 > 0)
    hash = (hash \oplus TailBytes(N)) \cdot k

return ShiftMix(ShiftMix(hash) \cdot k)
Murmur2 Strong Points

- Simple
- Fast (assuming multiplication is fairly cheap)
- Quality is quite good
Questions about Murmur2 (or any other choice)

- Could its speed be better?
- Could its quality be better?
Murmur2 Analysis

Inner loop is:

```java
for (int i = 0; i < iters; i++)
    hash = (hash ⊕ f(W_i)) · k
```

where $f$ is “Mul-ShiftMix-Mul”
Murmur2 Speed

- ILP comes mostly from parallel application of $f$
- Cost of $\text{TailBytes}(N)$ can be painful for $N < 60$ or so
Murmur2 Quality

- \( f \) is invertible
- During the loop, diffusion isn’t perfect
Common tests include:

- Hash a bunch of words or phrases
- Hash other real-world data sets
- Hash all strings with edit distance $\leq d$ from some string
- Hash other synthetic data sets
  - E.g., 100-word strings where each word is “cat” or “hat”
  - E.g., any of the above with extra space
- We use our own plus SMHasher
Common tests include:

▶ Hash a bunch of words or phrases
▶ Hash other real-world data sets
▶ Hash all strings with edit distance $\leq d$ from some string
▶ Hash other synthetic data sets
  ▶ E.g., 100-word strings where each word is “cat” or “hat”
  ▶ E.g., any of the above with extra space
▶ We use our own plus SMHasher
▶ avalanche
Avalanche (by example)

Suppose we have a function that inputs and outputs 32 bits. Find $M$ random input values. Hash each input value with and without its $j^{th}$ bit flipped. How often do the results differ in their $k^{th}$ output bit?
Suppose we have a function that inputs and outputs 32 bits. Find \( M \) random input values. Hash each input value with and without its \( j^{th} \) bit flipped. How often do the results differ in their \( k^{th} \) output bit?

Ideally we want “coin flip” behavior, so the relevant distribution has mean \( M/2 \) and variance \( 1/4M \).
64x64 avalanche diagram: $f(x) = x$
64x64 avalanche diagram: $f(x) = kx$
64x64 avalanche diagram: *ShiftMix*
64x64 avalanche diagram: \( \text{ShiftMix}(x) \cdot k \)
64x64 avalanche diagram: $\text{ShiftMix}(kx) \cdot k$
64x64 avalanche diagram: $f(x) = CRC(kx)$
The CityHash Project

Goals:

▶ Speed (on Google datacenter hardware or similar)
▶ Quality
  ▶ Excellent diffusion
  ▶ Excellent behavior on all contributed test data
  ▶ Excellent behavior on basic synthetic test data
  ▶ Good internal state diffusion—but not too good, cf. Rogaway’s Bucket Hashing
Portability

For speed without total loss of portability, assume:

- 64-bit registers
- pipelined and superscalar
- fairly cheap multiplication
- cheap $+,-,\oplus,\sigma,\rho,\beta$
- cheap register-to-register moves
For speed without total loss of portability, assume:

- 64-bit registers
- pipelined and superscalar
- fairly cheap multiplication
- cheap $+, −, \oplus, \sigma, \rho, \beta$
- cheap register-to-register moves
- $a + b$ may be cheaper than $a \oplus b$
- $a + cb + 1$ may be fairly cheap for $c \in \{0, 1, 2, 4, 8\}$
Branches are expensive

Is there a better way to handle the “tails” of short strings?
Branches are expensive

Is there a better way to handle the “tails” of short strings?

How many dynamic branches are reasonable for hashing a 12-byte input?
Branches are expensive

Is there a better way to handle the “tails” of short strings?

How many dynamic branches are reasonable for hashing a 12-byte input?

How many arithmetic operations?
CityHash64 initial design (2010)

- Focus on short strings
- Perhaps just use Murmur2 on long strings
- Use overlapping unaligned reads
- Write the minimum number of loops: 1
- Focus on speed first; fix quality later
The CityHash64 function: overall structure

```c
if (N <= 32)
    if (N <= 16)
        if (N <= 8)
            ...
        else ...
    else ...
else if (N <= 64) {
    // Handle 33 <= N <= 64
    ...
} else {
    // Handle N > 64
    int iters = ⌊N / 64⌋ ...
    ...
```
The CityHash64 function: overall structure

```c
if (N <= 32)
    if (N <= 16)
        if (N <= 8)
            ...
        else
            ...
    else
        ...
else
    ...
else if (N <= 64) {
    // Handle 33 <= N <= 64
    ...
} else {
    // Handle N > 64
    int iters = \lfloor N/64\rfloor
    ...
}
```
Define $\alpha(u, v, m)$:

\[
\begin{align*}
\text{let} & \quad a = u \oplus v \\
& \quad a' = \text{ShiftMix}(a \cdot m) \\
& \quad a'' = a' \oplus v \\
& \quad a''' = \text{ShiftMix}(a'' \cdot m)
\end{align*}
\]

in

\[a'''. \cdot m\]
The CityHash64 function (2012): preliminaries

Define $\alpha(u, v, m)$:

\[
\begin{align*}
\text{let } a &= u \oplus v \\
a' &= \text{ShiftMix}(a \cdot m) \\
a'' &= a' \oplus v \\
a''' &= \text{ShiftMix}(a'' \cdot m) \\
\end{align*}
\]

in $a''' \cdot m$

Also, $k_0$, $k_1$, and $k_2$ are primes near $2^{64}$, and $K$ is $k_2 + 2N$. 
let $a = B_0$
$b = B_{\lfloor N/2 \rfloor}$
$c = B_{N-1}$
$y = a + 256b$
$z = N + 4c$

in

$ShiftMix((y \cdot k_2) \oplus (z \cdot k_0))$
CityHash64: $4 \leq N \leq 8$

$$\alpha(N + 4W_0^{32}, W_{-1}^{32}, K)$$
CityHash64: $9 \leq N \leq 16$

let $a = W_0 + k_2$
$b = W_{-1}$
$c = \rho_{37}(b) \cdot K + a$
$d = (\rho_{25}(a) + b) \cdot K$

in
\[ \alpha(c, d, K) \]
let \( a = W_0 \cdot k_1 \)
\( b = W_1 \)
\( c = W_{-1} \cdot K \)
\( d = W_{-2} \cdot k_2 \)
in
\[ \alpha(\rho_{43} (a + b) + \rho_{30} (c) + d, a + \rho_{18} (b + k_2) + c, K) \]
let  
\[ a = W_0 \cdot k_2 \]
\[ e = W_2 \cdot k_2 \]
\[ f = W_3 \cdot 9 \]
\[ h = W_{-2} \cdot K \]
\[ u = \rho_{43}(a + W_{-1}) + 9(\rho_{30}(W_1) + c) \]
\[ v = a + W_{-1} + f + 1 \]
\[ w = h + \beta((u + v) \cdot K) \]
\[ x = \rho_{42}(e + f) + W_{-3} + \beta(W_{-4}) \]
\[ y = (\beta((v + w) \cdot K) + W_{-1}) \cdot K \]
\[ z = e + f + W_{-3} \]
\[ r = \beta((x + z) \cdot K + y) + W_1 \]
\[ t = ShiftMix((r + z) \cdot K + W_{-4} + h) \]
\[ \text{in} \]
\[ tK + x \]
Evaluation for $N \leq 64$
Evaluation for $N \leq 64$

- CityHash64 is about 1.5x faster than Murmur2 for $N \leq 64$
- Quality meets targets (bug reports are welcome)
- Simplifying it would be nice
Evaluation for $N \leq 64$

- CityHash64 is about 1.5x faster than Murmur2 for $N \leq 64$
- Quality meets targets (bug reports are welcome)
- Simplifying it would be nice
- Key lesson: Don’t loop over bytes
- Key lesson: Understand the basics of machine architecture
- Key lesson: Know when to stop
Next steps

Arguably we should have written CityHash32 next. That’s still not done.

Instead, we worked on 64-bit hashes for $N > 64$, and 128-bit hashes.
CityHash64 for $N > 64$

The one loop in CityHash64:

- 56 bytes of state
- 64 bytes consumed per iteration
- 7 rotates, 4 multiplies, 1 xor, about 36 adds (??)
- influenced by mix and Murmur2
128-bit CityHash variants

- CityHash128
  - same loop body, manually unrolled
  - slightly faster for large $N$

- CityHashCrc128
  - totally different function
  - uses CRC instruction, but isn’t a CRC
  - faster still for large $N$
Evaluation for $N > 64$
Evaluation for $N > 64$

- CityHash64 is about 1.3 to 1.6x faster than Murmur2
- For long strings, the fastest CityHash variant is about 2x faster than the fastest Murmur variant
- Quality meets targets (bug reports are welcome)
- Jenkins’ Spooky is a strong competitor
My recommendations

For hash tables or fingerprints:

<table>
<thead>
<tr>
<th>Nehalem, Westmere, Sandy Bridge, etc.</th>
<th>similar CPUs</th>
<th>other CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>small $N$</td>
<td>CityHash</td>
<td>CityHash</td>
</tr>
<tr>
<td>large $N$</td>
<td>CityHash</td>
<td>Spooky or CityHash</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TBD</td>
</tr>
</tbody>
</table>
My recommendations

For hash tables or fingerprints:

<table>
<thead>
<tr>
<th>Nehalem, Westmere, Sandy Bridge, etc.</th>
<th>similar CPUs</th>
<th>other CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>small $N$</td>
<td>CityHash</td>
<td>CityHash</td>
</tr>
<tr>
<td>large $N$</td>
<td>CityHash</td>
<td>Spooky or CityHash</td>
</tr>
</tbody>
</table>

For quick-and-dirty hashing: Start with the above
Future work

- CityHash32
- Big Endian
- SIMD
The End
The End

Backup Slides
Notation

\[
\begin{align*}
N & \quad = \quad \text{the length of the input (bytes)} \\
\mathit{a} \oplus \mathit{b} & \quad = \quad \text{bitwise exclusive-or} \\
\mathit{a} + \mathit{b} & \quad = \quad \text{sum (usually mod } 2^{64}\text{)} \\
\mathit{a} \cdot \mathit{b} & \quad = \quad \text{product (usually mod } 2^{64}\text{)} \\
\mathit{\sigma}_n(\mathit{a}) & \quad = \quad \text{right shift } \mathit{a} \text{ by } n \text{ bits} \\
\mathit{\sigma}_{-n}(\mathit{a}) & \quad = \quad \text{left shift } \mathit{a} \text{ by } n \text{ bits} \\
\mathit{\rho}_n(\mathit{a}) & \quad = \quad \text{right rotate } \mathit{a} \text{ by } n \text{ bits} \\
\mathit{\rho}_{-n}(\mathit{a}) & \quad = \quad \text{left rotate } \mathit{a} \text{ by } n \text{ bits} \\
\beta(\mathit{a}) & \quad = \quad \text{byteswap } \mathit{a}
\end{align*}
\]
More Notation

\[ B_i = \text{the } i^{th} \text{ byte of the input (counts from 0)} \]
\[ W_i^b = \text{the } i^{th} b\text{-bit word of the input} \]
More Notation

\[ B_i = \text{the } i^{th} \text{ byte of the input (counts from 0)} \]
\[ W^b_i = \text{the } i^{th} b\text{-bit word of the input} \]
\[ W^b_{-1} = \text{the last } b\text{-bit word of the input} \]
\[ W^b_{-2} = \text{the second-to-last } b\text{-bit word of the input} \]
The commonest explanation of a CRC is in terms of polynomials whose coefficients are elements of GF(2).
The commonest explanation of a CRC is in terms of polynomials whose coefficients are elements of GF(2). In GF(2):
- 0 is the additive identity,
- 1 is the multiplicative identity, and
- $1 + 1 = 0 + 0 = 0$. 
Sample polynomial:

\[ p = x^{32} + x^{27} + 1 \]
We can use $p$ to define an equivalence relation: We’ll say $q$ and $r$ are equivalent iff they differ by a polynomial times $p$. 
Theorem: The equivalence relation has $2^{\text{Degree}(p)}$ elements.
Theorem: The equivalence relation has $2^{\text{Degree}(p)}$ elements.

Lemma: if $\text{Degree}(p) = \text{Degree}(q) > 0$
then $\text{Degree}(p + q) < \text{Degree}(p)$
and, if not, $\text{Degree}(p + q) = \max(\text{Degree}(p), \text{Degree}(q))$
Theorem: The equivalence relation has $2^{\text{Degree}(p)}$ elements.

Lemma: if $\text{Degree}(p) = \text{Degree}(q) > 0$
then $\text{Degree}(p + q) < \text{Degree}(p)$
and, if not, $\text{Degree}(p + q) = \max(\text{Degree}(p), \text{Degree}(q))$

Observation: There are $2^{\text{Degree}(p)}$ polynomials with degree less than $\text{Degree}(p)$, none equivalent.
Observation: Any polynomial with degree \( \geq \text{Degree}(p) \) is equivalent to a lower degree polynomial.
Observation: Any polynomial with degree $\geq \text{Degree}(p)$ is equivalent to a lower degree polynomial.

Example: What is a degree $\leq 31$ polynomial equivalent to $x^{50}$?
Observation: Any polynomial with degree $\geq \text{Degree}(p)$ is equivalent to a lower degree polynomial.

Example: What is a degree $\leq 31$ polynomial equivalent to $x^{50}$?

$\text{Degree}(x^{50}) - \text{Degree}(p) = 18$; therefore $x^{50} - x^{18} \cdot p$ has degree less than 50.
Observation: Any polynomial with degree $\geq \text{Degree}(p)$ is equivalent to a lower degree polynomial.

Example: What is a degree $\leq 31$ polynomial equivalent to $x^{50}$?

$\text{Degree}(x^{50}) - \text{Degree}(p) = 18$; therefore $x^{50} - x^{18} \cdot p$ has degree less than 50.

\[
x^{50} - x^{18} \cdot p = x^{50} - x^{18} \cdot (x^{32} + x^{27} + 1) = x^{50} - (x^{50} + x^{45} + x^{18}) = x^{45} + x^{18}
\]
Applying the same idea repeatedly will lead us to the lowest degree polynomial that is equivalent to $x^{50}$. 
Applying the same idea repeatedly will lead us to the lowest degree polynomial that is equivalent to $x^{50}$.

The result:

$$x^{50} \equiv x^{30} + x^{18} + x^{13} + x^{8} + x^{3}$$
CRC, part 7

More samples:

\[ x^{50} \equiv x^{30} + x^{18} + x^{13} + x^8 + x^3 \]
\[ x^{50} + 1 \equiv x^{30} + x^{18} + x^{13} + x^8 + x^3 + 1 \]
\[ x^{51} \equiv x^{31} + x^{19} + x^{14} + x^9 + x^4 \]
\[ x^{51} + x^{50} \equiv x^{31} + x^{30} + x^{19} + x^{18} + x^{14} + x^{13} + x^9 + x^8 + x^4 + x^3 \]
\[ x^{51} + x^{31} \equiv x^{19} + x^{14} + x^9 + x^4 \]
There are thousands of CRC implementations

We’ll focus on those that use `_mm_crc32_u64()` or `crc32q`

The inputs are a 32-bit number and a 64-bit number

The output is a 32-bit number
What is crc32q?

crc32q for inputs $u$ and $v$ returns
$C(u \text{xor} v) = F(E(D(u \text{xor} v)))$.

$D(0) = 0, D(1) = x^{95}, D(2) = x^{94}, D(3) = x^{95} + x^{94}, D(4) = x^{93}, \ldots$

$E$ maps a polynomial to the equivalent with lowest-degree.

$F(0) = 0, F(x^{31}) = 1, F(x^{30}) = 2, F(x^{31} + x^{30}) = 3, F(x^{29}) = 4, \ldots$
How is \texttt{crc32q} used?

\textit{C} operates on 64 bits of input, so:

For a 64-bit input, use \( C(\text{seed}, u_0) \).
How is \texttt{crc32q} used?

\( C \) operates on 64 bits of input, so:

For a 64-bit input, use \( C(\text{seed}, u_0) \).

For a 128-bit input, use \( C(C(\text{seed}, u_0), u_1) \).
How is \texttt{crc32q} used?

\( C \) operates on 64 bits of input, so:

For a 64-bit input, use \( C(\text{seed}, u_0) \).

For a 128-bit input, use \( C(C(\text{seed}, u_0), u_1) \).

For a 192-bit input, use \( C(C(C(\text{seed}, u_0), u_1), u_2) \).
A $32 \times 64$ matrix times a $64 \times 1$ vector yields a $32 \times 1$ result.
A 32 × 64 matrix times a 64 × 1 vector yields a 32 × 1 result. The matrix and vectors contain elements of GF(2):