

Perfect Hashing in an Imperfect World

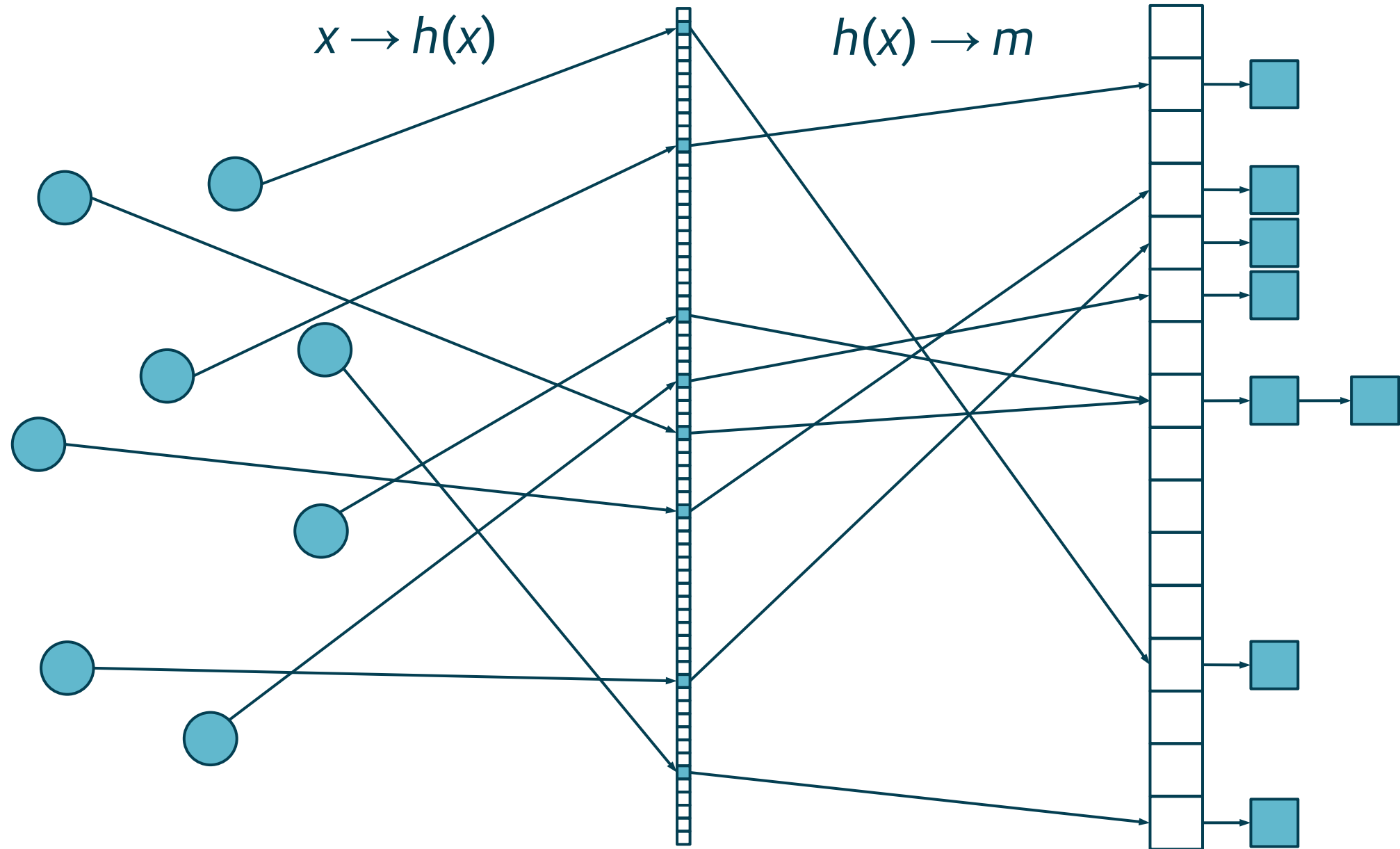


using `std::cpp` 2024

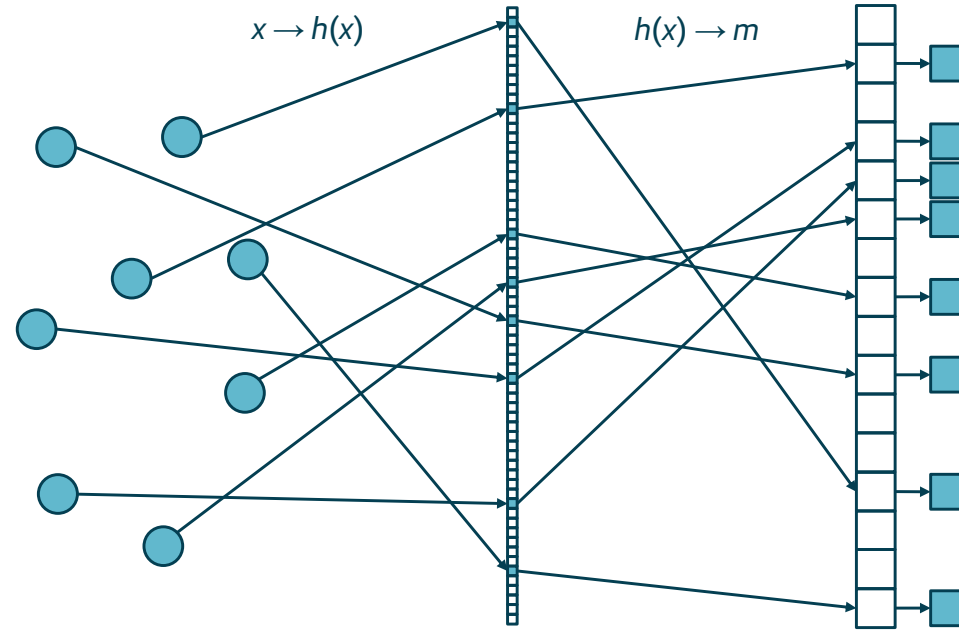
Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>

Madrid, April 2024

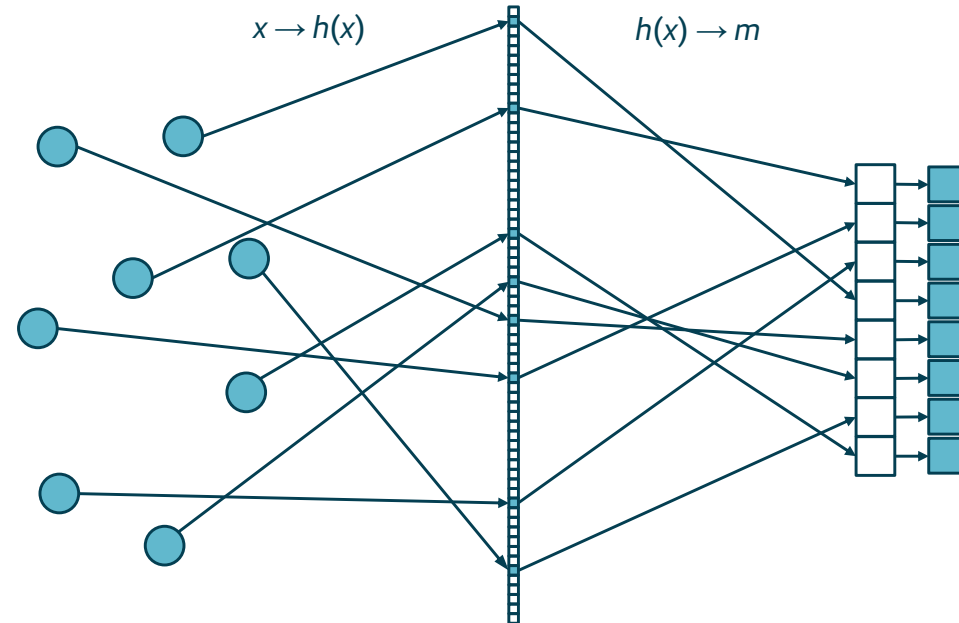




Perfect hashing



Minimal perfect hashing



- n elements, m buckets

- $m \rightarrow \infty$

- $m = n^2$

- $\alpha = n/m = 1$

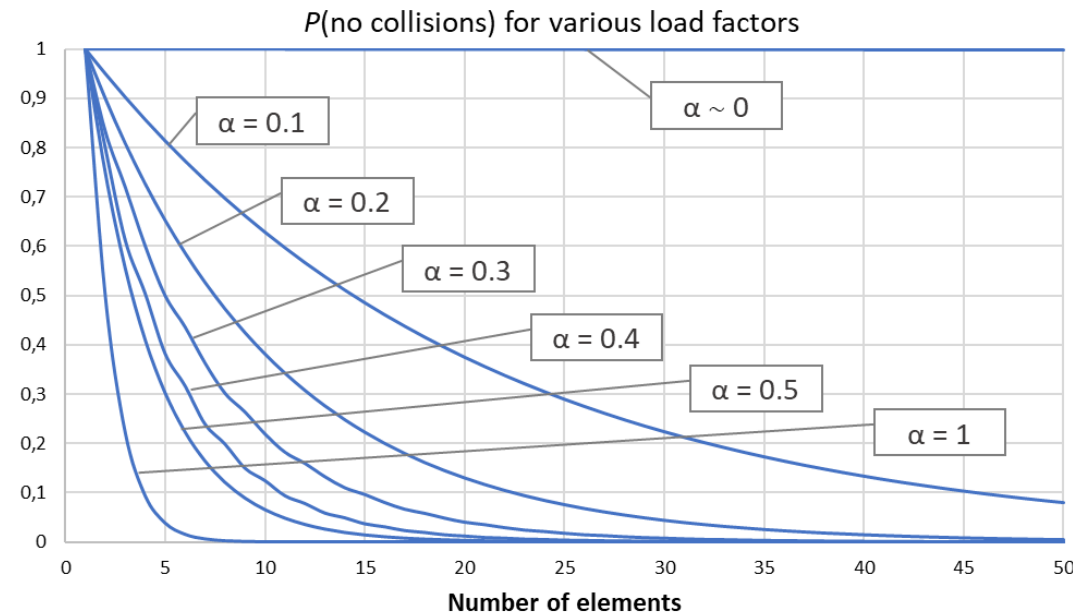
$$P(\text{no collisions}) = \frac{\binom{m}{n} n!}{m^n} = \frac{m!}{m^n (m-n)!}$$

$$P(\text{no collisions}) \rightarrow 1$$

$$P(\text{no collisions}) \geq 0.5$$

$$P(\text{no collisions}) = \frac{n!}{n^n} \sim \frac{\sqrt{2\pi n}}{e^n} = O(e^{c \ln n - n})$$

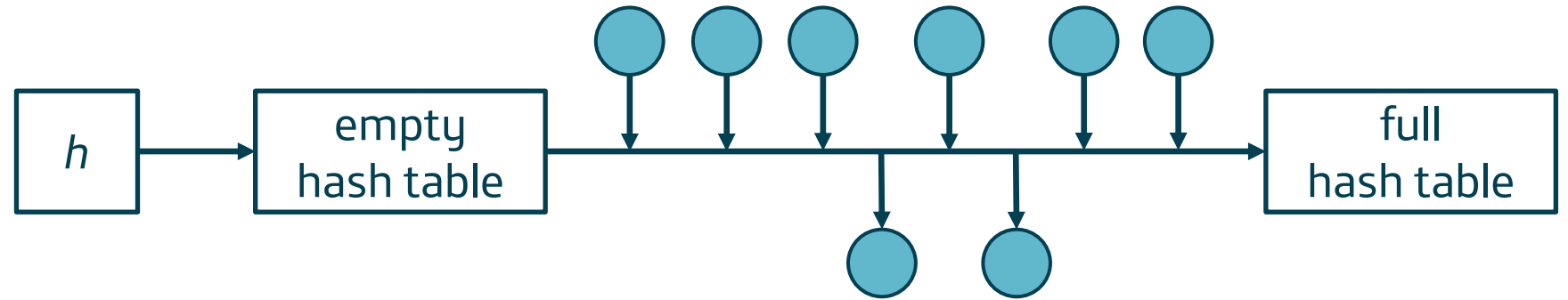
- So... very lucky



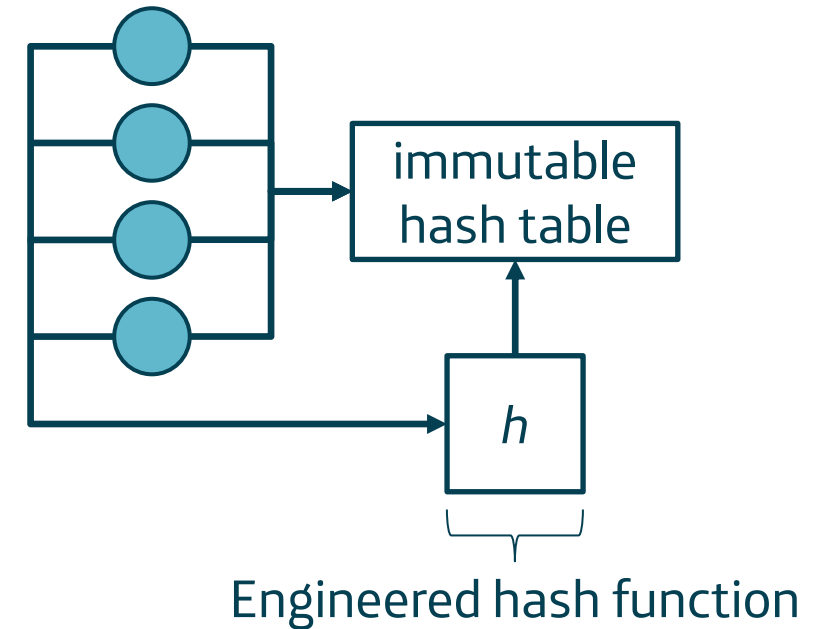
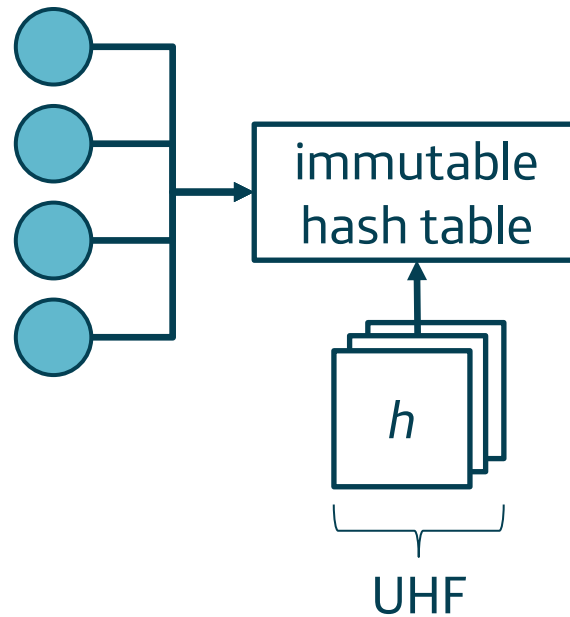
Enlightenment



Regular hashing



Perfect hashing





- Guaranteed $O(1)$ lookup
- And extremely fast at that
 - Very few branches
 - Particularly with minimal perfect hashing
 - Additional performance opportunities with `constexpr`



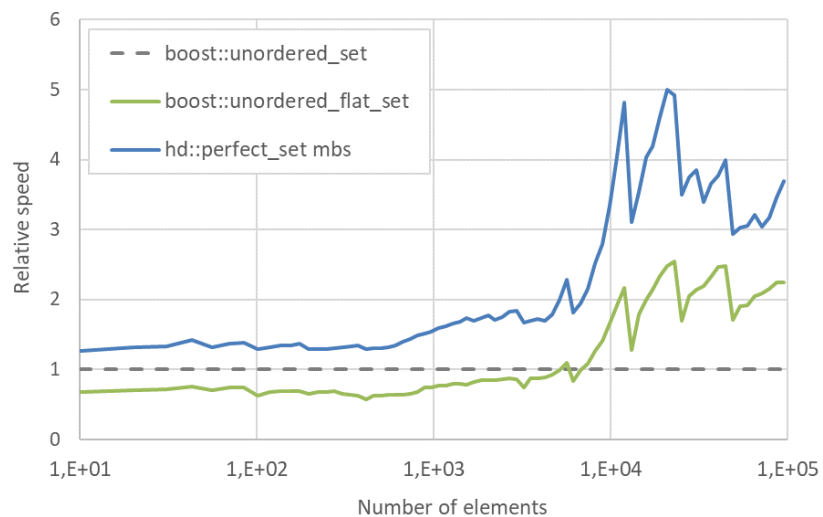
- Immutability
- Long construction times
- Can fail on initialization
 - Equivalent to pathological $O(n)$ behavior in regular hashing

- Perfect hash tables are no replacement for regular hashing

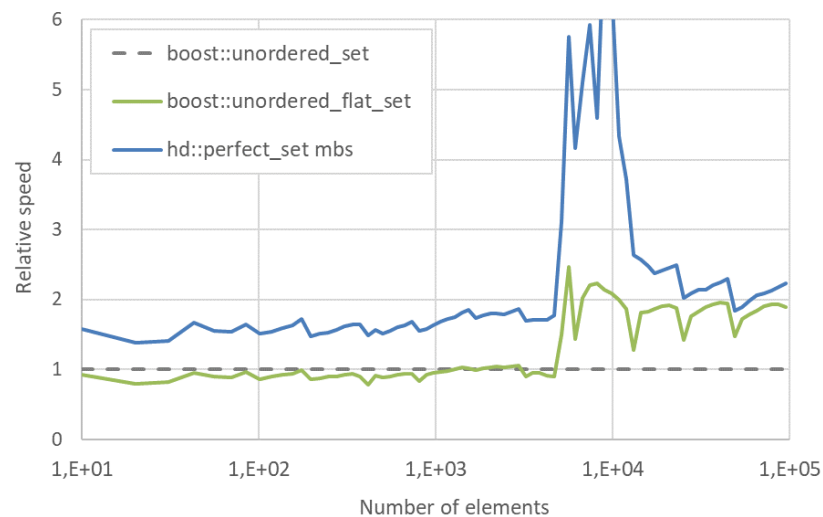
Relative performance (without tuning)

7

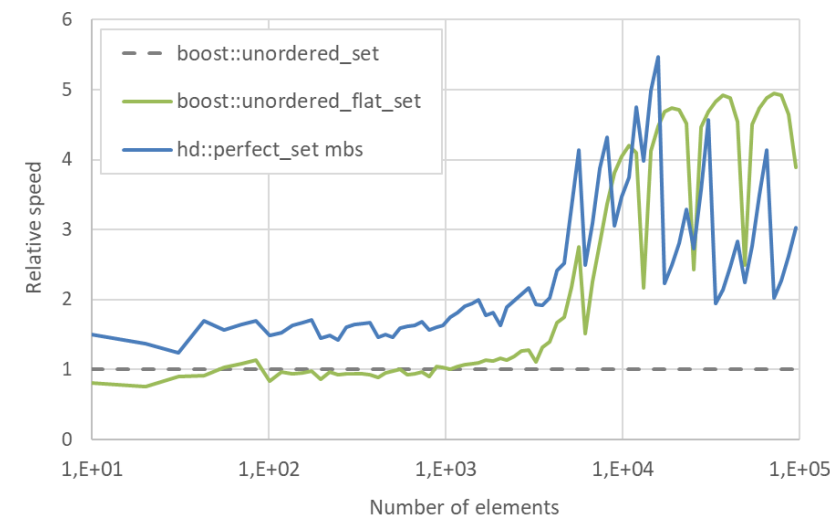
Successful find, integers



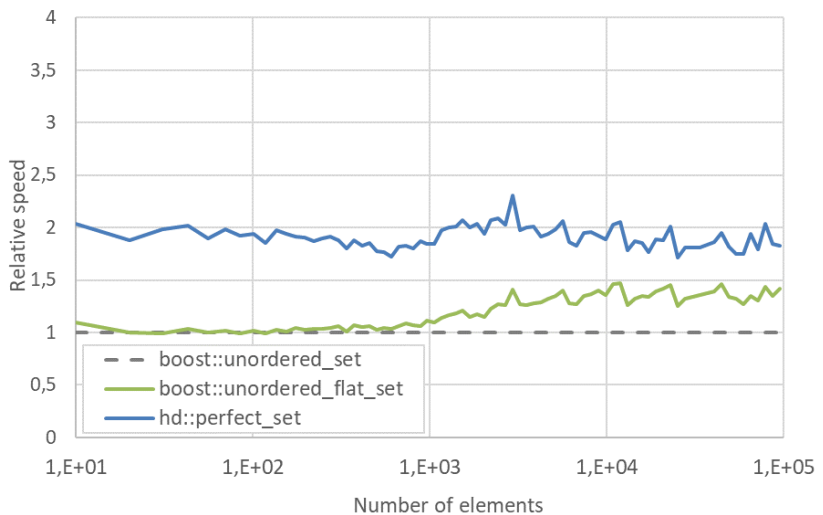
50/50 find, integers



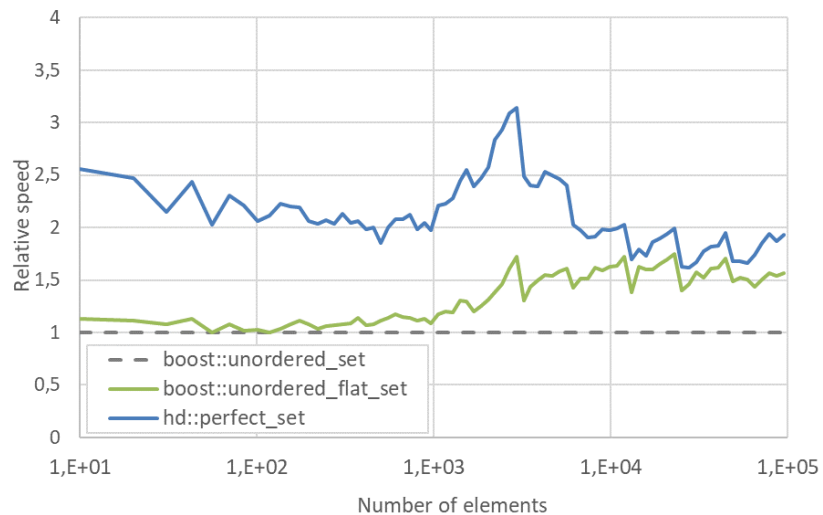
Unsuccessful find, integers



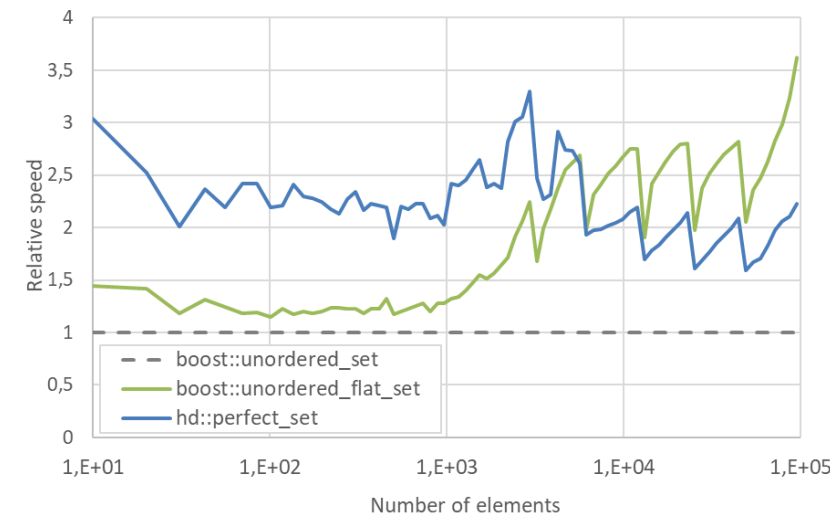
Successful find, strings



50/50 find, strings

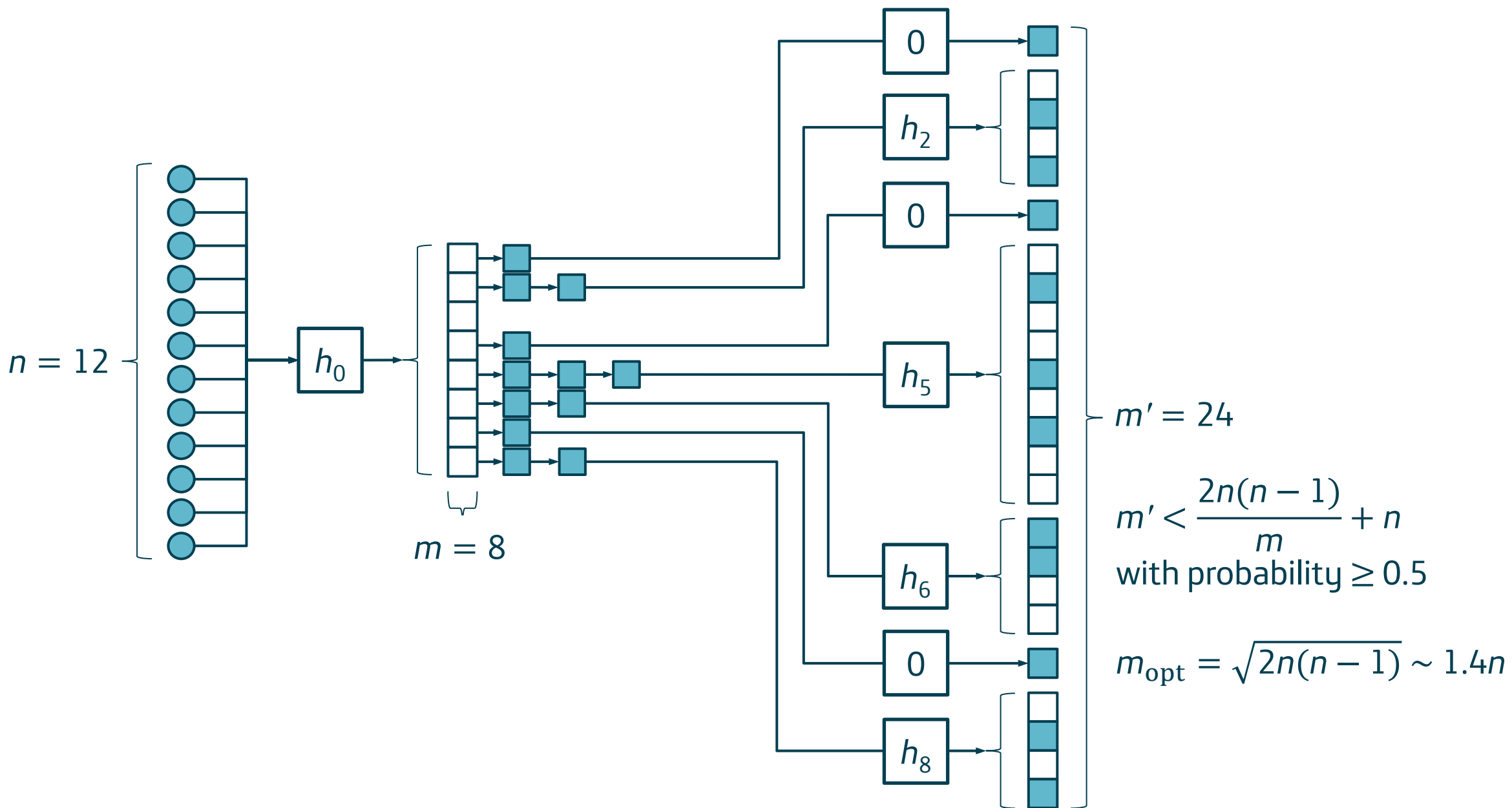


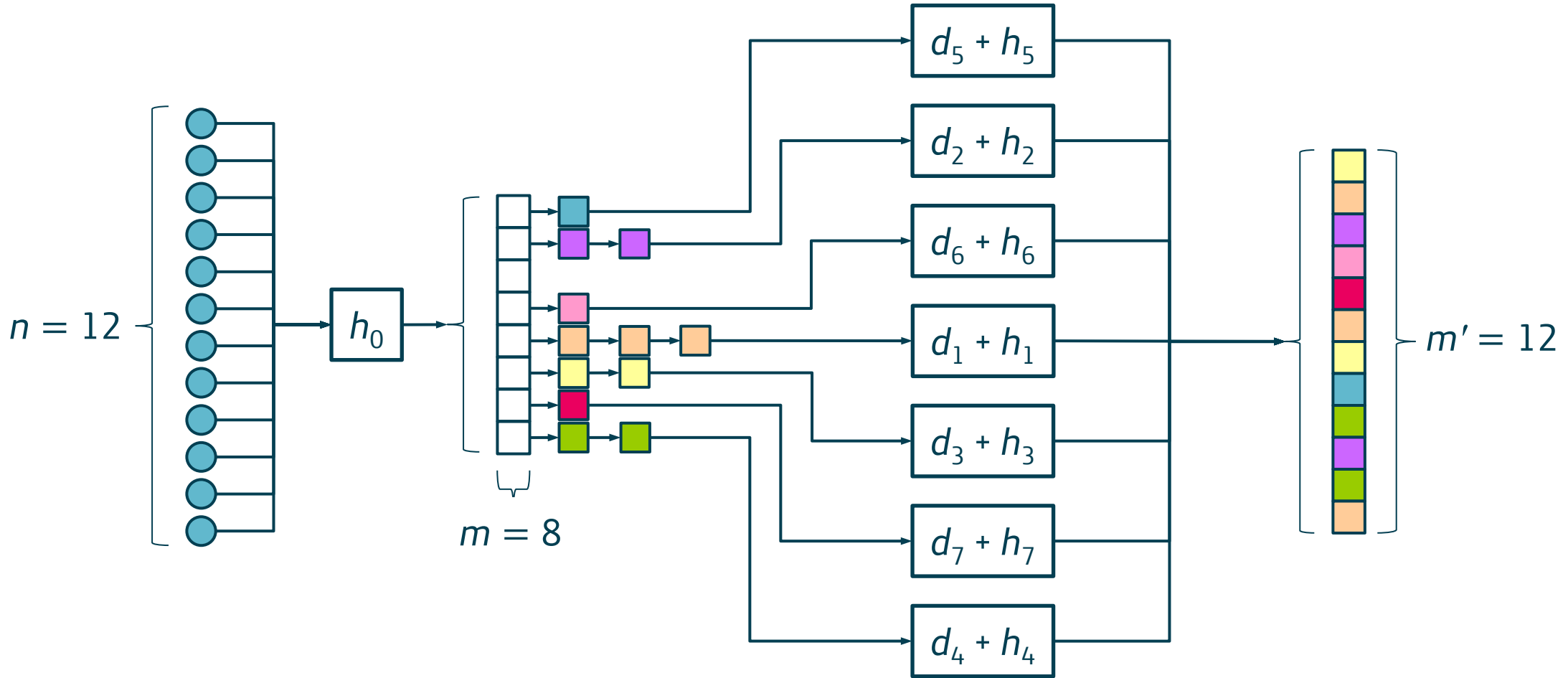
Unsuccessful find, strings

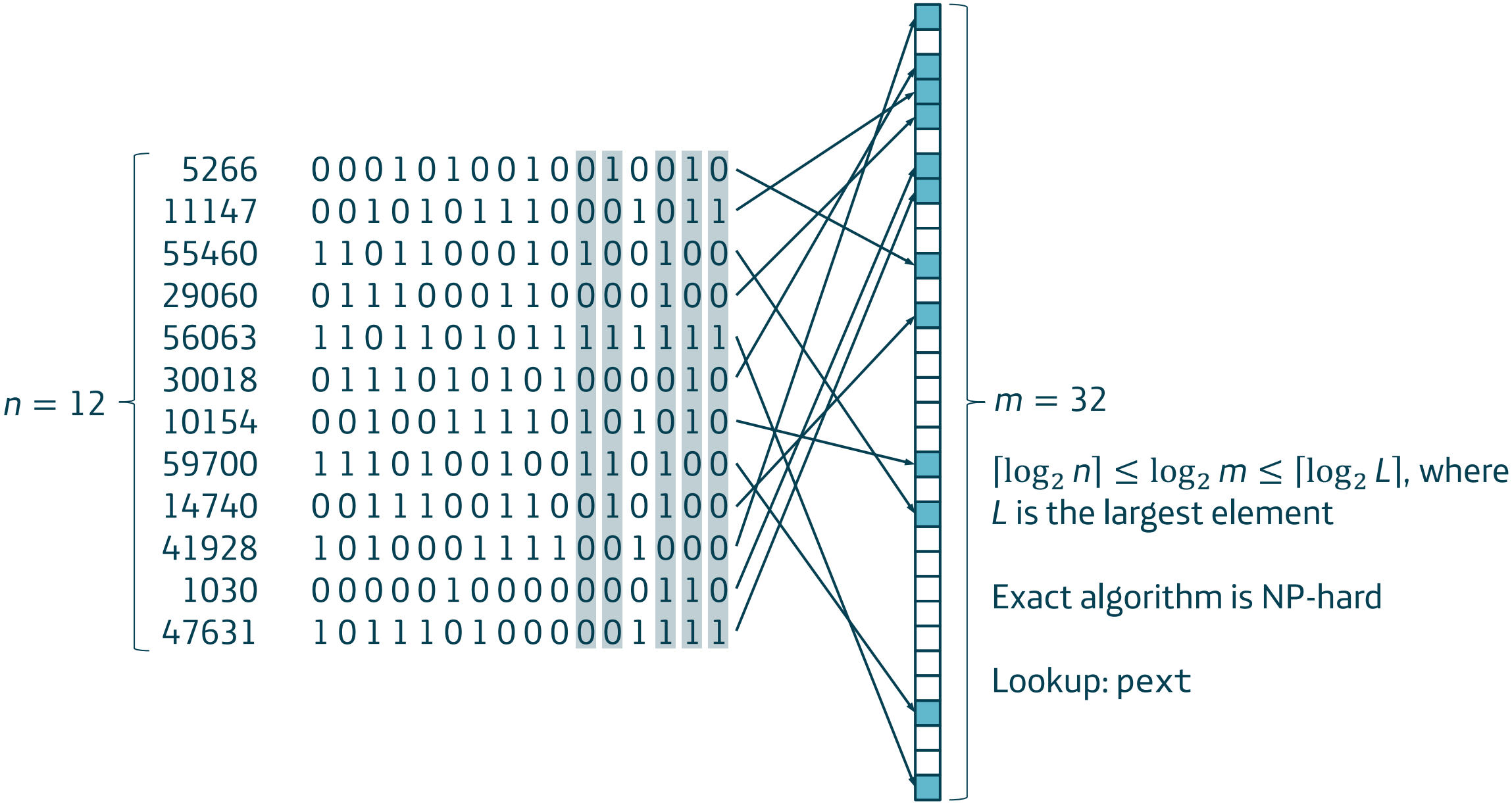


Techniques

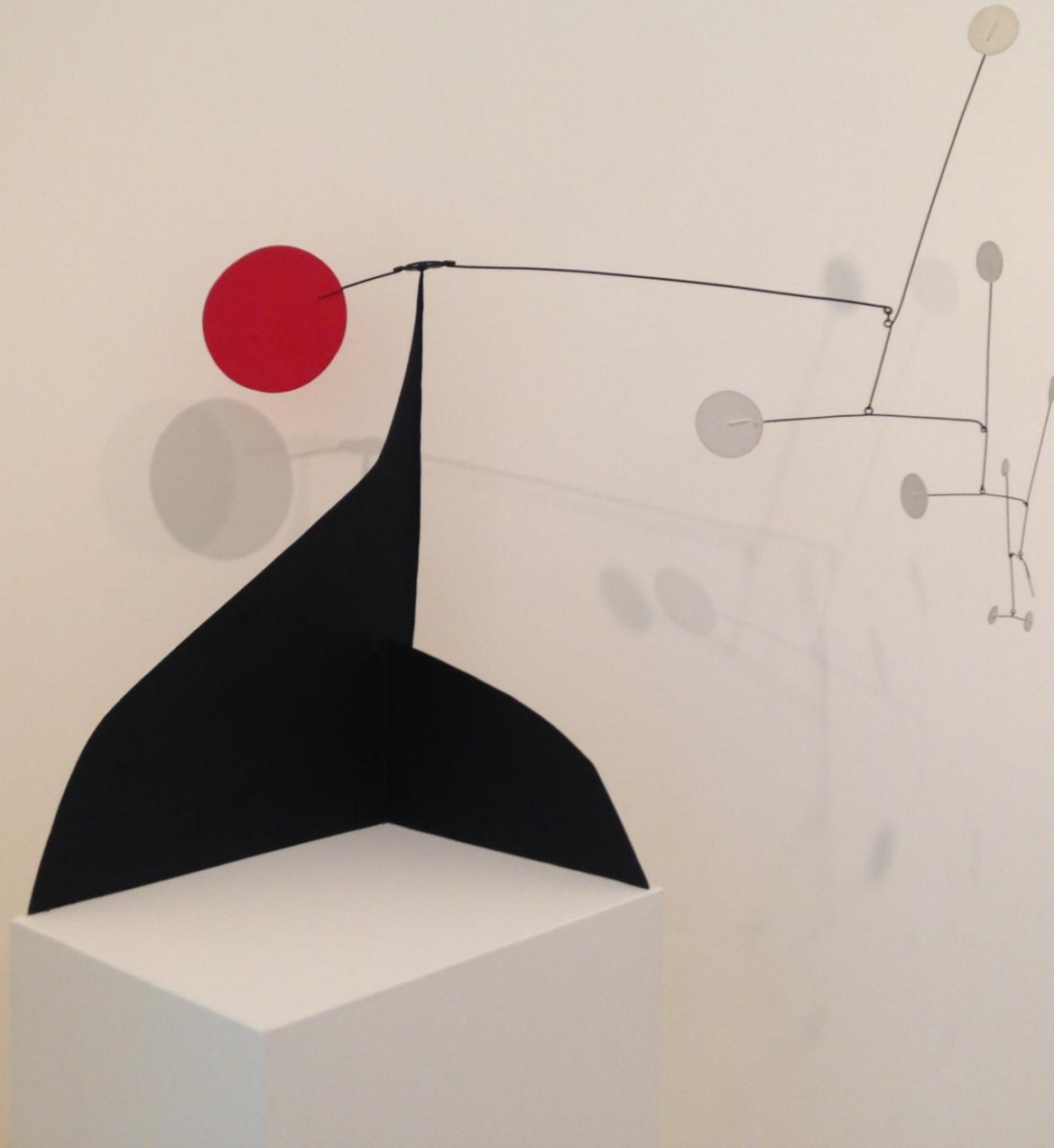


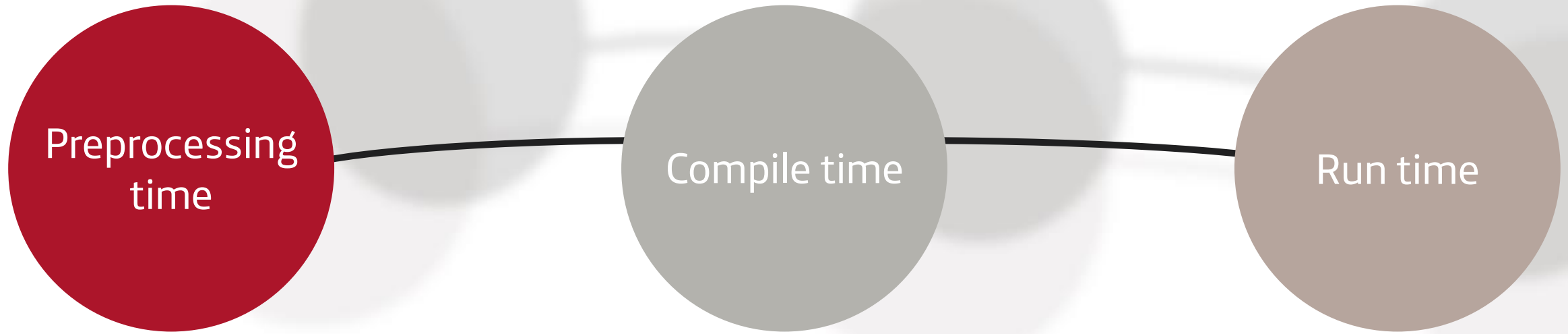






Construction time





- External tool → source code
- Works for any C++ standard
- Fast build times
- Complicates project build

- Using constexpr magic
- Requires later C++ standards
- Compilation times suffer (can be confined to its own TU)

- Sometimes the only option
- Doesn't require later C++ standards
- Compilation times OK
- Potentially slower?
- What if construction fails?

Libraries



- Douglas Schmidt 1989-2003
- Preprocessing time, `xx.gperf` \rightarrow `xx.h` (C or C++ code)
- Only strings, oriented towards keyword recognition
 - Used by GCC in various places
- Non-minimal, plus allows for “near-perfect” hash functions
- $h(s) = \sum_{i=1}^K (v[s[p_i]] + d_i) + v[s[\text{len}(s) - 1]] + \text{len}(s)$
- Tries to minimize K (number of string positions checked) and $\max h(s)$


```

#include <string.h>

#define TOTAL_KEYWORDS 100
#define MIN_WORD_LENGTH 2
#define MAX_WORD_LENGTH 20
#define MIN_HASH_VALUE 2
#define MAX_HASH_VALUE 166
/* maximum key range = 165, duplicates = 0 */

class Perfect_Hash
{
private:
    static inline unsigned int
    hash (const char *str, size_t len);
public:
    static const char *
    in_word_set (const char *str, size_t len);
};

```

```

inline unsigned int
Perfect_Hash::hash (const char *str, size_t len)
{
    static unsigned char asso_values[] =
    {
        167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
        167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
        167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
        167, 167, 167, 167, 167, 167, 167, 167, 167, 167,
        167, 50, 45, 167, 167, 167, 167, 167, 167, 167,
        167, 167, 167, 167, 167, 80, 167, 10, 45, 167,
        167, 35, 167, 167, 167, 167, 60, 167, 35, 60,
        55, 20, 0, 5, 15, 35, 0, 167, 167, 0,
        5, 167, 167, 167, 167, 167, 167, 20, 10, 10,
        ... /* 256 entries */
    };
    unsigned int hval = len;

    switch (hval)
    {
        default:
            hval +=
                asso_values[static_cast<unsigned char>(str[4])];
            /*FALLTHROUGH*/
        case 4:
            hval +=
                asso_values[static_cast<unsigned char>(str[3])];
            /*FALLTHROUGH*/
        case 3:
        case 2:
            hval +=
                asso_values[static_cast<unsigned char>(str[1])];
            /*FALLTHROUGH*/
        case 1:
            hval +=
                asso_values[static_cast<unsigned char>(str[0])];
            break;
    }
    return hval;
}

```

```

const char *
Perfect_Hash::in_word_set (const char *str, size_t len)
{
    static const char * wordlist[] =
    {
        "", "",
        "pr",
        "", "", "", "", "",
        "sim",
        "pscr",
        "",
        "prurel",
        "ee",
        "Ycy",
        "npar",
        "ltrie",
        "nrtrie",
        "ne",
        "nlE",
        "ltcc",
        "rceil",
        "nltrie",
        "npolint",
        "lnapprox",
        ... /* 167 entries */
    };

    if (len <= MAX_WORD_LENGTH && len >= MIN_WORD_LENGTH)
    {
        unsigned int key = hash (str, len);

        if (key <= MAX_HASH_VALUE)
        {
            const char *s = wordlist[key];

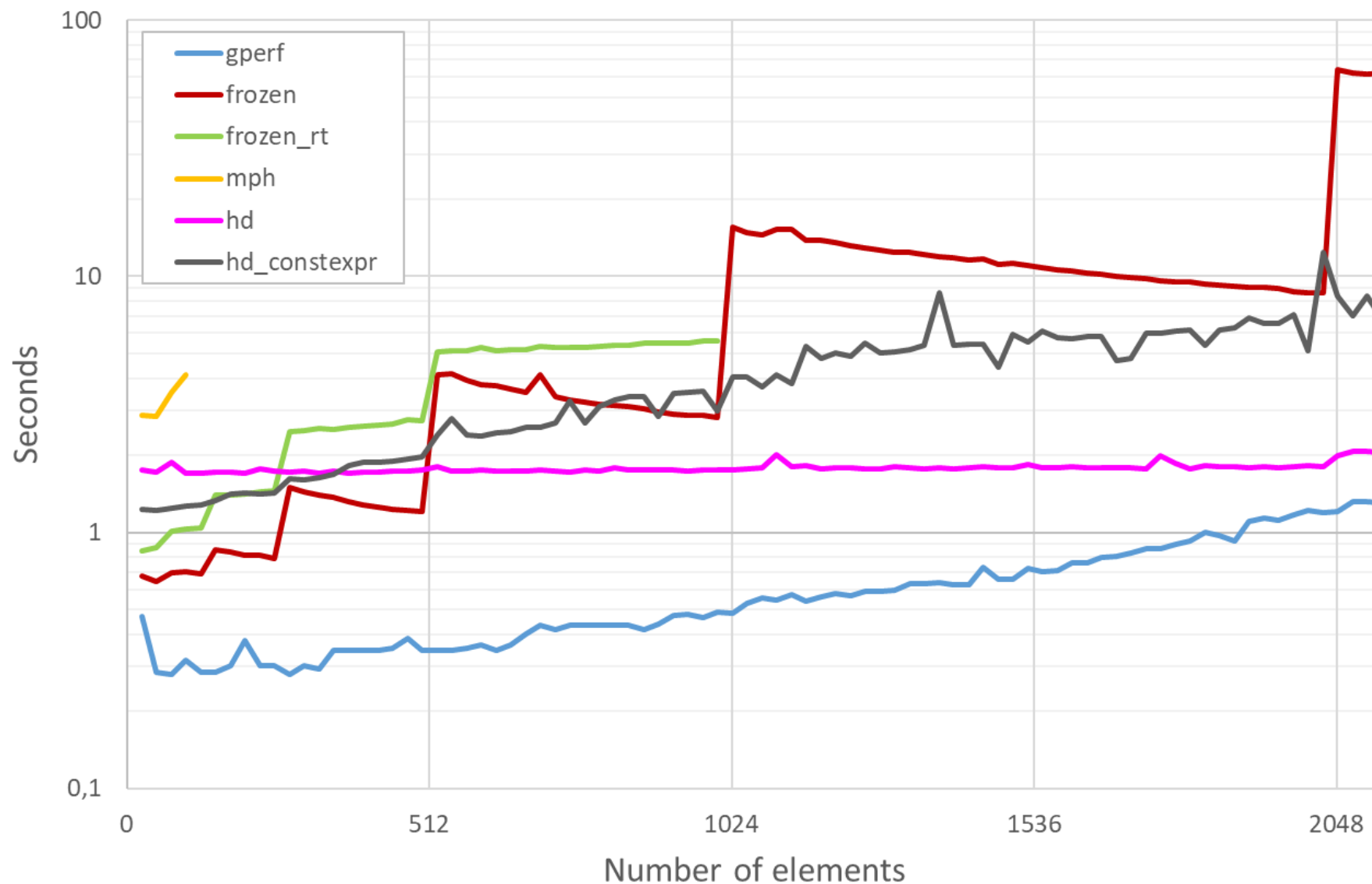
            if (*str == *s && !strcmp (str + 1, s + 1))
                return s;
        }
    }
    return 0;
}

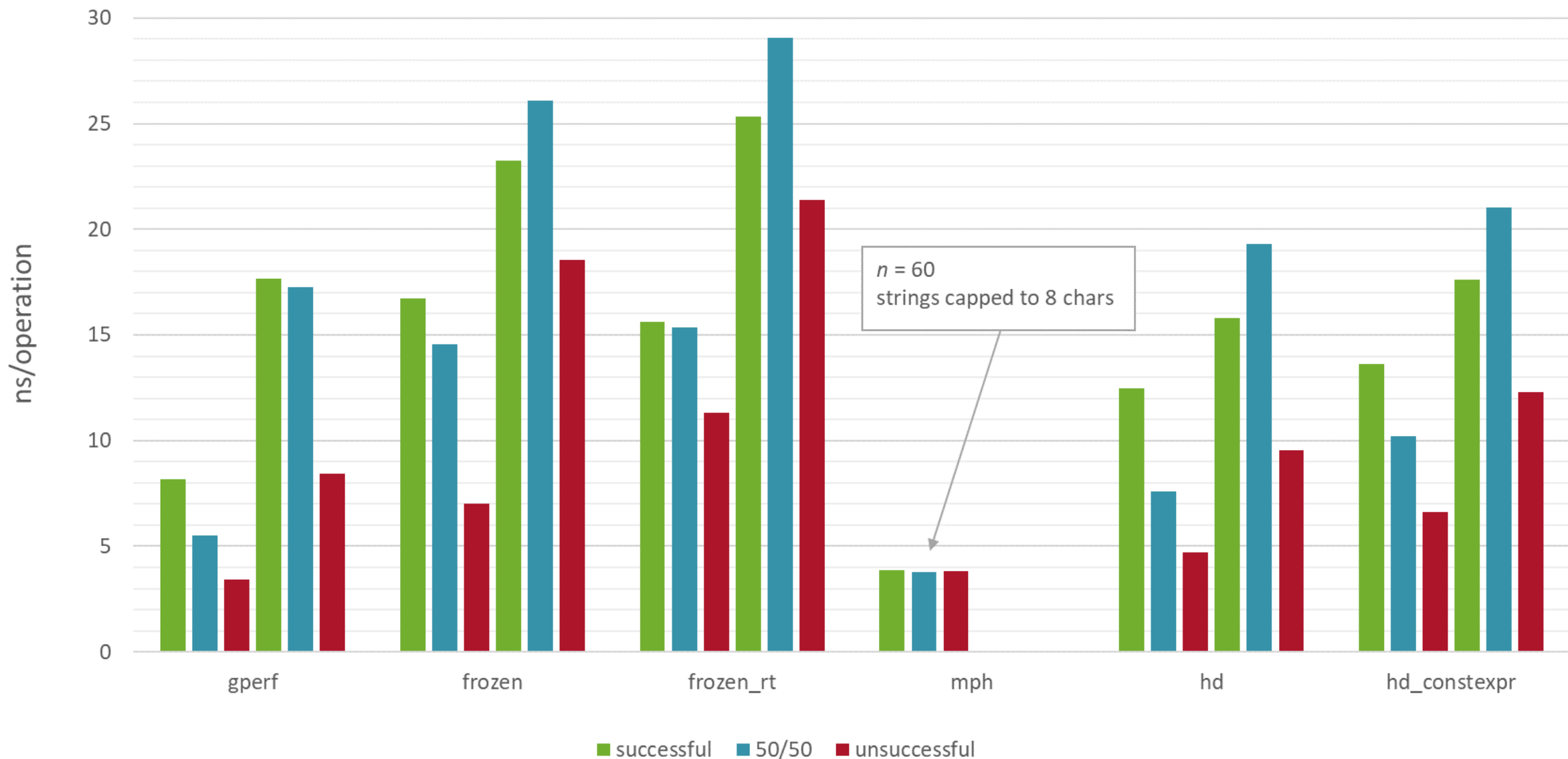
```

- Serge Guelton et al. 2017-today
- Compile time, C++14, sets/maps, map values mutable if non-constexpr
 - `-fconstexpr-ops-limit` required save for very small tables
- Works with any key type
 - Off-the-shelf support for integral types (Wang's 64 bit mixing) and `frozen::string` (FNV-1a)
- Non-minimal (m is a power of 2)
- Based on HD(C), UHF: $h(x, seed)$
 - Primary array \rightarrow secondary array \rightarrow element array

- Kris Jusiak 2024
- Compile time, C++20, function from key to value
 - `-fconstexpr-ops-limit` required save for very small tables
- Supported keys: integral types and strings (max size 8)
- Supported values: integral types
- Algorithms
 - $n < 4$, integrals: generated switch
 - $n < 4$, same-sized strings (4 or 8): generated switch on SWAR values
 - else: minimal cover + pext

- WIP
- `hd::perfect_set`
 - Run time, C++11
 - Any key value (`boost::hash` by default)
 - Minimal HD(C), $h_0 = \text{low}(h)$, UHF: $\text{high}(h \cdot m)$
 - Primary array \rightarrow secondary array
- `hd::constexpr_perfect_set`
 - Compile-time version (watch out, `boost::hash` not `constexpr`)
 - `-fconstexpr-ops-limit` required for medium sized tables ($n > 1,000$)

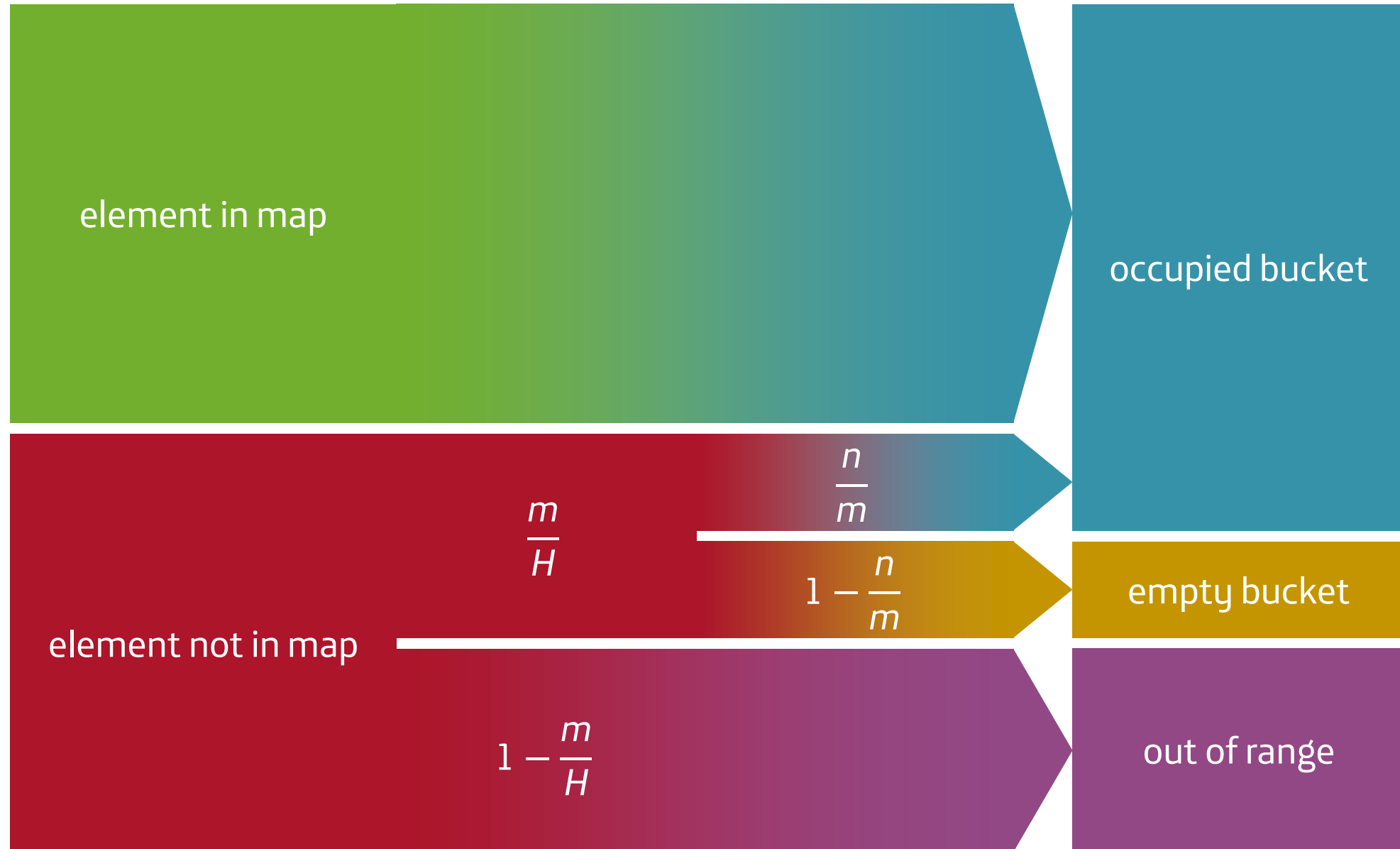




❶ `constexpr table<T, N>::table(Input)`

❷ `constexpr table<Input>::table()`

- ❶ is enough for UHF approaches
- ❷ may potentially be faster (particularly with engineered hash functions)
 - But requires structural strings and generates unwieldy symbols
- `constexpr` only allows transient dynamic memory (as of C++23)
 - `footprint(table) ≠ sizeof(table)` (❷ can go static)
 - `footprint(table) = f(std::size(Input))` → OK
 - `footprint(table) = f(Input)` → double computation



Use cases



Translation tables
Keyword detection

Update seldom, read many
(e.g. JSON objects, YOMM2
dispatch table)

Faster switches

Database indexing

<https://godbolt.org/z/qfKEhE4bM>

```
unsigned y=0;
```

```
void foo(unsigned x)
```

```
{  
    switch(x){  
        case 0: y*=2; break;  
        case 1: y+=7; break;  
        case 2: y*=2; break;  
        case 3: y-=5; break;  
        case 4: y/=4; break;  
        default: break;  
    }  
}
```

```
foo(unsigned int):  
    cmp     edi, 4  
    ja     .L1  
    mov     edi, edi  
    jmp     [QWORD PTR .L4[0+rdi*8]]  
.L4:  
    .quad   .L6  
    .quad   .L7  
    .quad   .L6  
    .quad   .L5  
    .quad   .L3  
.L3:  
    shr     DWORD PTR y[rip], 2  
.L1:  
    ret  
.L6:  
    sal     DWORD PTR y[rip]  
    ret  
.L7:  
    add     DWORD PTR y[rip], 7  
    ret  
.L5:  
    sub     DWORD PTR y[rip], 5  
    ret  
y:  
    .zero   4
```

<https://godbolt.org/z/Y79csGve3>

```
unsigned y=0;
```

```
void foo(unsigned x)
```

```
{  
    switch(x){  
        case 0: y*=2; break;  
        case 2: y+=7; break;  
        case 3: y*=2; break;  
        case 10: y-=5; break;  
        case 11: y/=4; break;  
        default: break;  
    }  
}
```

```
foo(unsigned int):
```

```
    cmp     edi, 11
```

```
    ja     .L1
```

```
    mov     edi, edi
```

```
    jmp     [QWORD PTR .L4[0+rdi*8]]
```

```
.L4:
```

```
    .quad   .L6
```

```
    .quad   .L1
```

```
    .quad   .L7
```

```
    .quad   .L6
```

```
    .quad   .L1
```

```
    .quad   .L1
```

```
    .quad   .L1
```

```
    .quad   .L1
```

```
    .quad   .L1
```

```
    .quad   .L1
```

```
    .quad   .L5
```

```
    .quad   .L3
```

```
.L3:
```

```
    shr     DWORD PTR y[rip], 2
```

```
.L1:
```

```
    ret
```

```
.L6:
```

```
    sal     DWORD PTR y[rip]
```

```
    ret
```

```
.L7:
```

```
    add     DWORD PTR y[rip], 7
```

```
    ret
```

```
.L5:
```

```
    sub     DWORD PTR y[rip], 5
```

```
    ret
```

```
y:
```

```
    .zero   4
```

<https://godbolt.org/z/4K6WqWdv9>

```
unsigned y=0;
```

```
void foo(unsigned x)
```

```
{
    switch(x){
        case 10: y*=2; break;
        case 21: y+=7; break;
        case 32: y*=2; break;
        case 49: y-=5; break;
        case 52: y/=4; break;
        default: break;
    }
}
```

```
foo(unsigned int):
    cmp     edi, 32
    je      .L2
    jbe     .L15
    cmp     edi, 49
    je      .L6
    cmp     edi, 52
    jne     .L16
    shr     DWORD PTR y[rip], 2
    ret

.L15:
    cmp     edi, 10
    je      .L2
    cmp     edi, 21
    jne     .L17
    add     DWORD PTR y[rip], 7
    ret

.L2:
    sal     DWORD PTR y[rip]
    ret

.L17:
    ret

.L16:
    ret

.L6:
    sub     DWORD PTR y[rip], 5
    ret
```

```
y:
    .zero   4
```


<https://godbolt.org/z/9sKdbhhrc>

```
constexpr std::pair<int,int> bs[] = {
    {32,2}, {49,3}, {10,0}, {-1,5},
    {52,4}, {21,1}, {-1,5}, {-1,5}
};

inline int ph(unsigned x)
{
    auto b = bs[x&7];
    return x==b.first? b.second: 5;
}

unsigned y=0;

void foo(unsigned x)
{
    switch(ph(x)){
        case 0: y*=2; break;
        case 1: y+=7; break;
        case 2: y*=2; break;
        case 3: y-=5; break;
        case 4: y/=4; break;
        case 5:      break;
        default:      std::unreachable();
    }
}
```

```
foo(unsigned int):
    mov     eax, edi
    and     eax, 7
    mov     edx, DWORD PTR bs[4+rax*8]
    cmp     edi, DWORD PTR bs[0+rax*8]
    je      .L10
.L1:
    ret
.L10:
    jmp     [QWORD PTR .L4[0+rdx*8]]
.L4:
    .quad   .L7
    .quad   .L8
    .quad   .L7
    .quad   .L6
    .quad   .L5
    .quad   .L1
.L7:
    sal     DWORD PTR y[rip]
    ret
.L5:
    shr     DWORD PTR y[rip], 2
    ret
.L6:
    sub     DWORD PTR y[rip], 5
    ret
```

```
.L8:
    add     DWORD PTR y[rip], 7
    ret
y:
    .zero   4
bs:
    .long   32
    .long   2
    .long   49
    .long   3
    .long   10
    .long   0
    .long   -1
    .long   5
    .long   52
    .long   4
    .long   21
    .long   1
    .long   -1
    .long   5
    .long   -1
    .long   5
```

<https://godbolt.org/z/zcrGhWh6j>

```
unsigned y=0;
```

```
void foo(std::string_view x)
{
    if(x=="0") y*=2;
    else if(x=="1") y+=7;
    else if(x=="2") y*=2;
    else if(x=="3") y-=5;
    else if(x=="4") y/=4;
}
```

```
foo(std::string_view):
    cmp     rdi, 1
    je      .L11
.L1:
    ret
.L11:
    movzx   eax, BYTE PTR [rsi]
    cmp     al, 48
    jne     .L3
.L9:
    sal     DWORD PTR y[rip]
    ret
.L3:
    cmp     al, 49
    jne     .L5
    add     DWORD PTR y[rip], 7
    ret
.L5:
    cmp     al, 50
    je      .L9
    cmp     al, 51
    jne     .L7
    sub     DWORD PTR y[rip], 5
    ret
```

```
.L7:
    cmp     al, 52
    jne     .L1
    shr     DWORD PTR y[rip], 2
    ret
y:
    .zero   4
```

<https://godbolt.org/z/4njjj4cbh>

```
inline int ph(std::string_view x)
{
    return x.size()==1?
        x[0]-'0': 5;
}
```

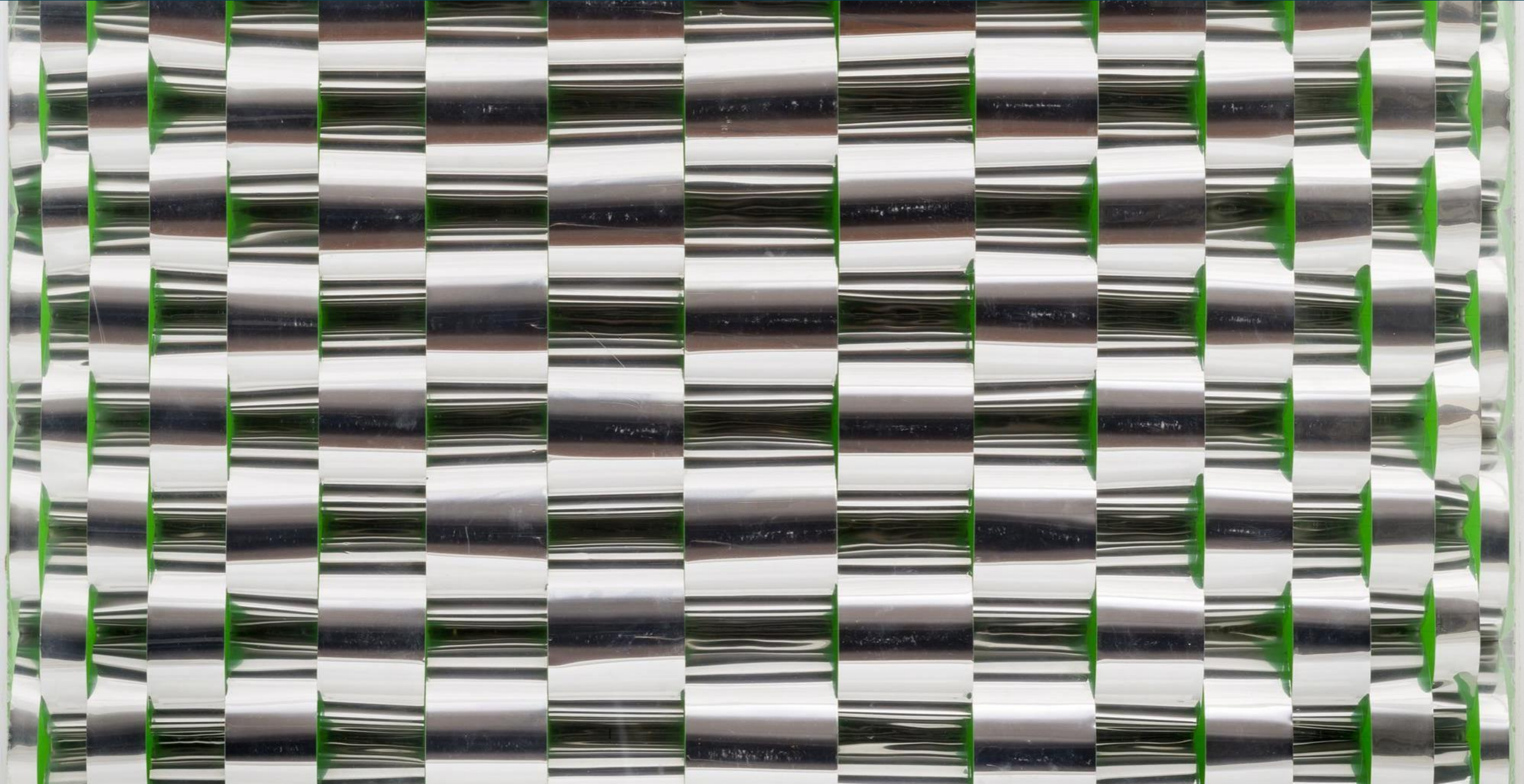
```
unsigned y=0;
```

```
void foo(std::string_view x)
{
    switch(ph(x)){
        case 0: y*=2; break;
        case 1: y+=7; break;
        case 2: y*=2; break;
        case 3: y-=5; break;
        case 4: y/=4; break;
        default: break;
    }
}
```

```
foo(std::string_view):
    cmp     rdi, 1
    je      .L9
.L1:
    ret
.L9:
    movsx   eax, BYTE PTR [rsi]
    sub     eax, 48
    cmp     eax, 4
    ja      .L1
    jmp     [QWORD PTR .L4[0+rax*8]]
.L4:
    .quad   .L6
    .quad   .L7
    .quad   .L6
    .quad   .L5
    .quad   .L3
.L6:
    sal     DWORD PTR y[rip]
    ret
.L3:
    shr     DWORD PTR y[rip], 2
    ret
.L5:
    sub     DWORD PTR y[rip], 5
    ret
```

```
.L7:
    add     DWORD PTR y[rip], 7
    ret
y:
    .zero   4
```

Conclusions



Conclusions

- Key insight: perfection achieved through immutability
 - Perfect hashing covers a narrower problem space than regular hash tables
- Two algorithmic approaches: UHF, engineered hash function
- Three construction strategies: preprocessing time, compile time, run time
 - Run time has received very little attention so far
- gperf is a really good tool
- Compile-time constructs are still subpar
 - Transient constexpr dynamic memory allocation (P2670R1)
 - Complex code generation still difficult
- Stay tuned to [#boost-unordered](#) in [cpplang.slack.com](#)

Perfect Hashing in an Imperfect World

Thank you

github.com/joaquintides/usingstdcpp2024

“Venezia era tutta d'oro”, © 1961 Lucio Fontana; “Descending Light”, © 2007 Ai Weiwei; “Peine del viento XV”, © 1976 Eduardo Chillida;
“Red Disc, White Dots on Black”, © 1960 Alexander Calder; “Power Tools”, © 2007 Thomas Hirschhorn;
“Los bañistas, Cannes”, © 1956 Pablo Ruiz Picasso; “Relief No. 505”, © ca. 1968 Marina Apollonio

`using std::cpp 2024`

Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>

Madrid, April 2024

