Optimizing Binary Search

SERGEY SLOTIN

Cppcon
The C++ Conference

2022
September 12th-16th

ABOUT ME

- Author of *Algorithms for Modern Hardware* (en.algorithmica.org/hpc)
- Designed the world's fastest **binary search,** B-tree, integer parsing, factorization, prefix sum, array searching, argmin… (github.com/sslotin/amh-code)

Email: me@sereja.me

Twitter: @sergey_slotin

Telegram: @bydlokoder

OUTLINE

- Vanilla binary search (Babylonians, 200BC; Mauchly, 1946)
- Branchless binary search: 3x on small arrays (LLVM contributors, early 2010s–…)
- Eytzinger binary search: 2x on large arrays (Khuong & Morin, 2015)
- S-tree: 5-8x (Intel, 2010; Slotin, 2020)
- S+ tree: 7-15x (Slotin, 2022)

THE PROBLEM

```
// perform any O(n) precomputation with a sorted array
void prepare(int *a, int n);

// return the first element y ≥ x in O(log n)
int lower_bound(int x);
```

- 32-bit integers
- Random keys & queries
- Optimize for **throughput**

(Not a drop-in `std::lower_bound` replacement, but closer to C++23 `std::flat_set`)

Comparison-based search only

(no lookup tables, tries, interpolation searches, etc.)

```
int lower_bound(int x) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = (l + r) / 2;
        if (t[m] >= x)
            r = m;
        else
            l = m + 1;
    }
    return t[l];
}
```

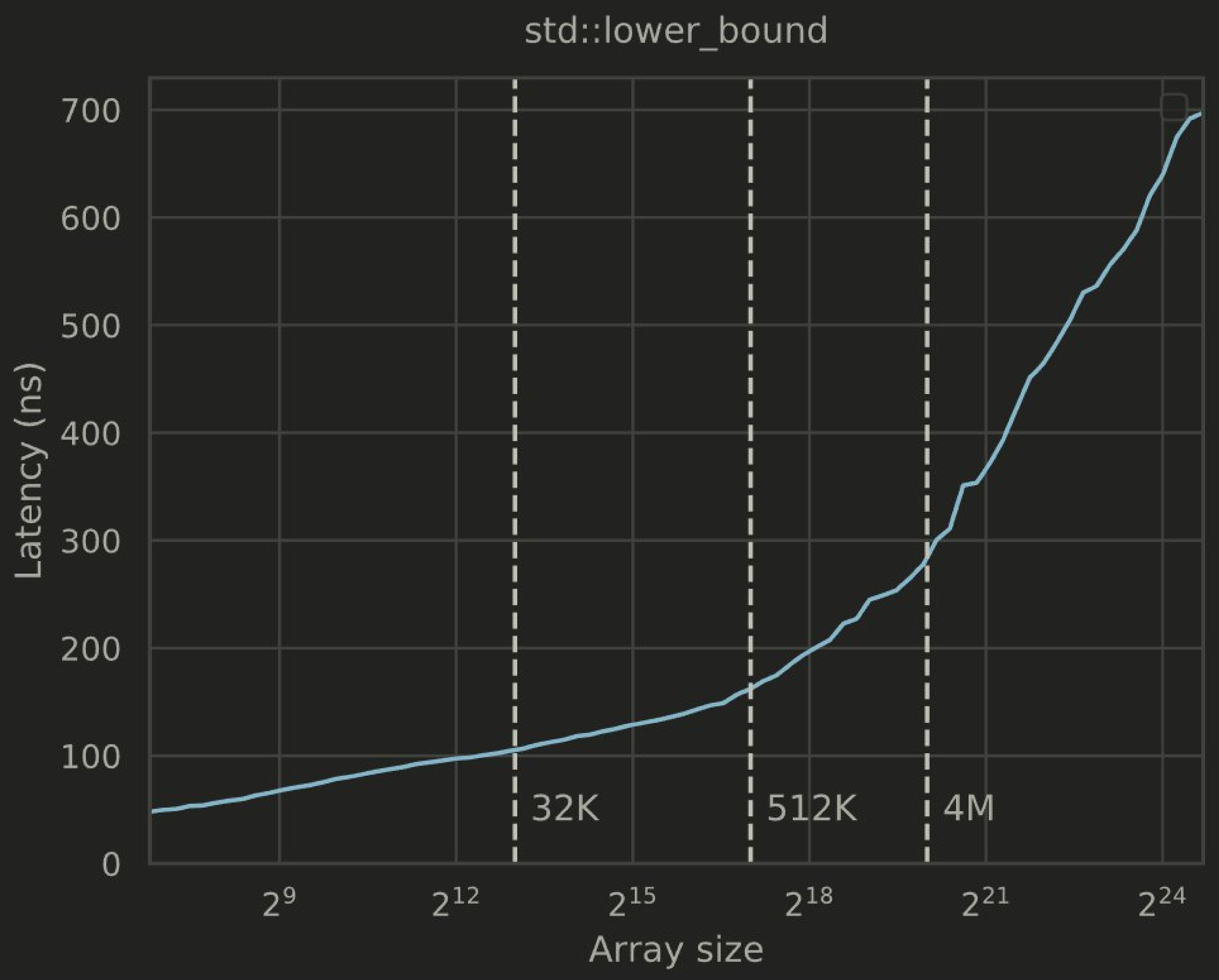
As found in the first 50 pages of any CS textbook

```
template <class _Compare, class _ForwardIterator, class _Tp> _LIBCPP_CONSTEXPR_AFTER_CXX17 _ForwardIterator
__lower_bound(_ForwardIterator __first, _ForwardIterator __last, const _Tp& __value_, _Compare __comp)
{
    typedef typename iterator_traits<_ForwardIterator>::difference_type difference_type;
    difference_type __len = _VSTD::distance(__first, __last);
    while (__len != 0)
    {
        difference_type __l2 = _VSTD::__half_positive(__len);
        _ForwardIterator __m = __first;
        _VSTD::advance(__m, __l2);
        if (__comp(*__m, __value_))
        {
            __first = ++__m;
            __len -= __l2 + 1;
        }
        else
            __len = __l2;
    }
    return __first;
}
```

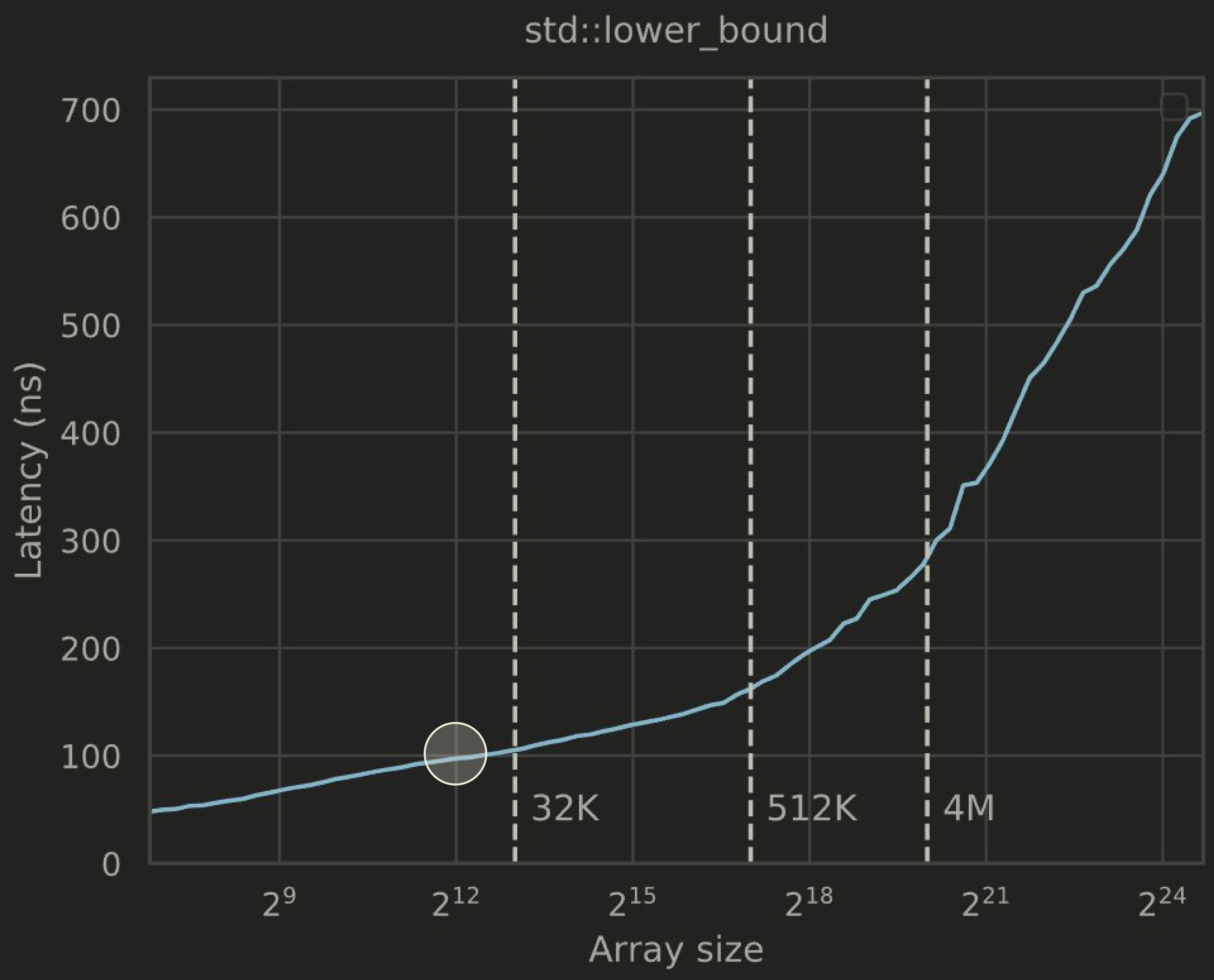
github.com/llvm-mirror/libcxx/blob/master/include/algorithm#L4169

```
      │ 35:   mov    %rax,%rdx
 0.52 │       sar    %rdx
 0.33 │       lea    (%rsi,%rdx,4),%rcx
 4.30 │       cmp    (%rcx),%edi      ←── a[(l + r) / 2] ≥ x?
65.39 │     ↓ jle    b0
 0.07 │       sub    %rdx,%rax
 9.32 │       lea    0x4(%rcx),%rsi
 0.06 │       dec    %rax
 1.37 │       test   %rax,%rax
 1.11 │     ↑ jg     35
```

perf record & perf report ($n = 10^6$)



Zen 2 @ 2GHz
DDR4 @ 2667MHz



Zen 2 @ 2GHz
DDR4 @ 2667MHz

$$\frac{100}{12} \times 2 \approx 17 \text{ cycles/iteration}$$

seems too high for L1

CPU PIPELINE

To execute any instruction, a lot of preparatory work is needed:

- **fetch** a chunk of machine code from memory
- **decode** it and split into instructions
- **execute** these instructions (possibly involving some **memory** operations)
- **write** the results back into registers

This whole sequence takes up to 15-20 CPU cycles

CPU PIPELINE

Pipelining: on each stage, the next instruction is processed the right away, without waiting for the previous one to fully complete

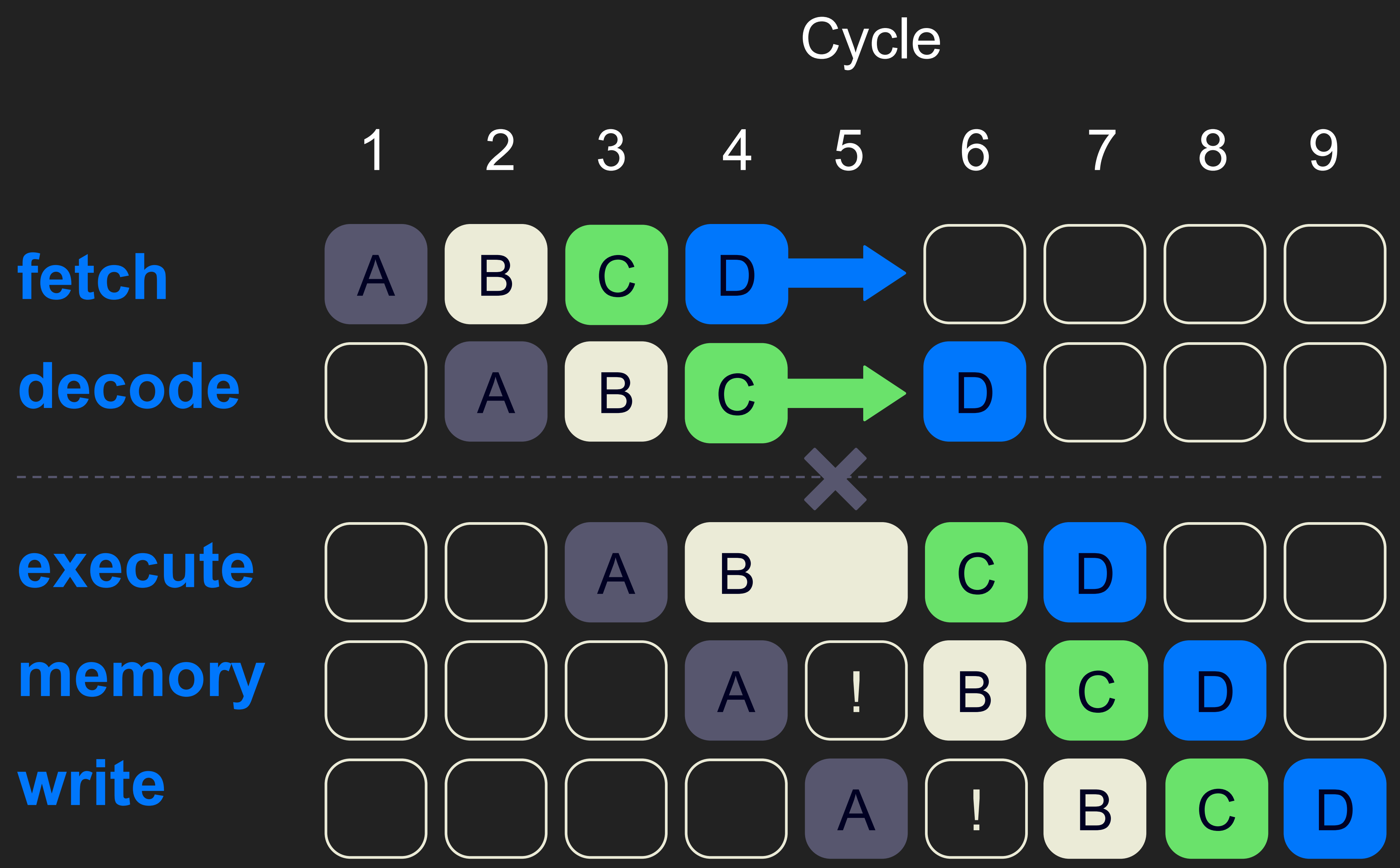
Cycle

	1	2	3	4	5	6	7
fetch	A	B	C				
decode		A	B	C			
execute			A	B	C		
memory				A	B	C	
write					A	B	C

Doesn’t reduce actual latency but makes it seem like it’s only execution & memory

CPU PIPELINE

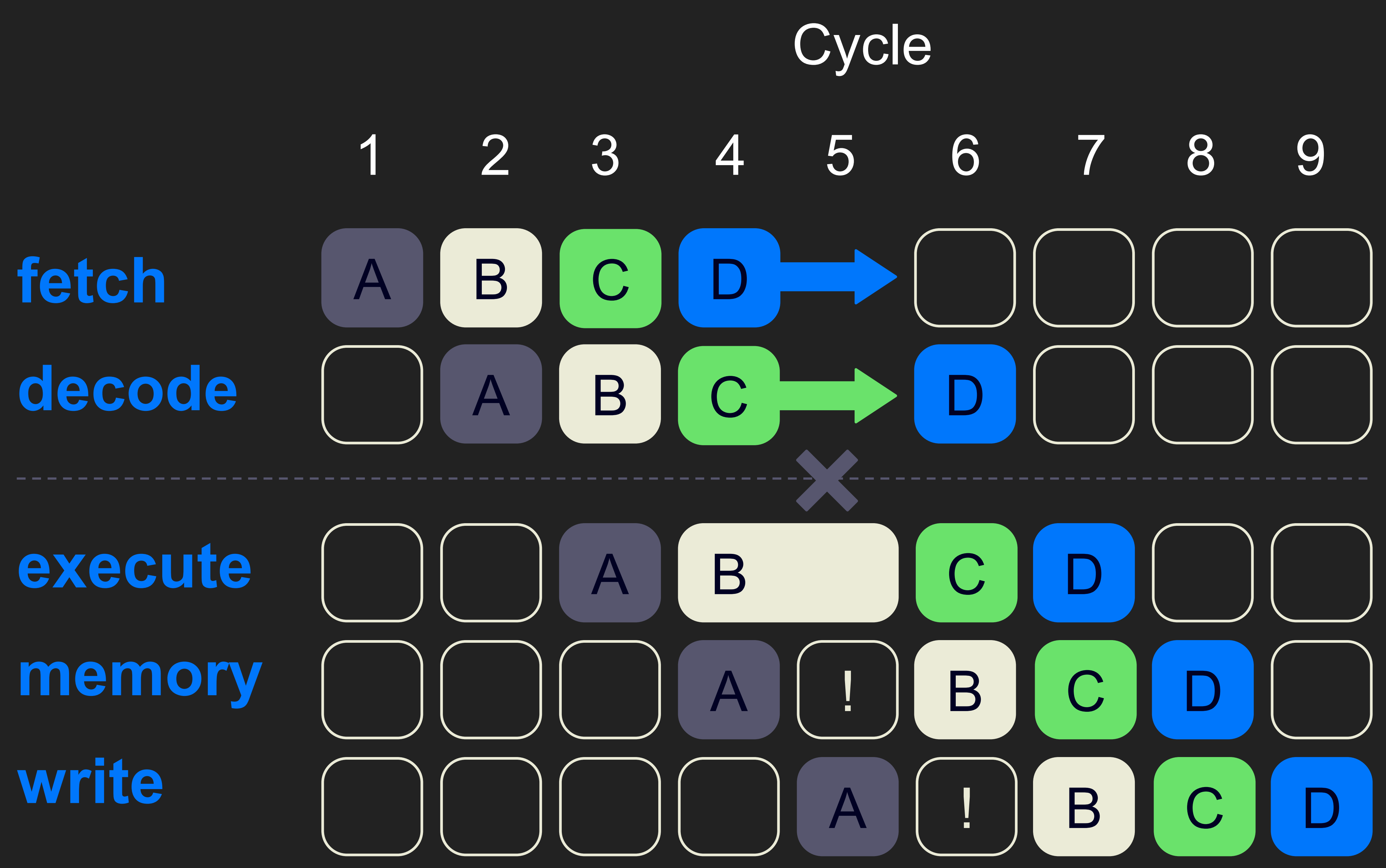
Pipeline hazard: the next instruction can’t execute on the following clock cycle



- Structural hazard: two or more instructions need the same part of the CPU
- Data hazard: need to wait for operands to be computed from a previous step
- Control hazard: the CPU can’t tell which instructions to execute next

CPU PIPELINE

Pipeline hazard: the next instruction can't execute on the following clock cycle



- Structural hazard: two or more instructions need the same part of the CPU
- Data hazard: need to wait for operands to be computed from a previous step
- Control hazard: the CPU can't tell which instructions to execute next

^ requires a pipeline flush on branch misprediction

REMOVING BRANCHES

```
int lower_bound(int x) {
    int l = 0, r = n - 1;
    while (l < r) { // <- this branch can be easily predicted
        int m = (l + r) / 2;
        if (t[m] >= x) // <- this branch is what we care about
            r = m;
        else
            l = m + 1;
    }
    return t[l];
}
```

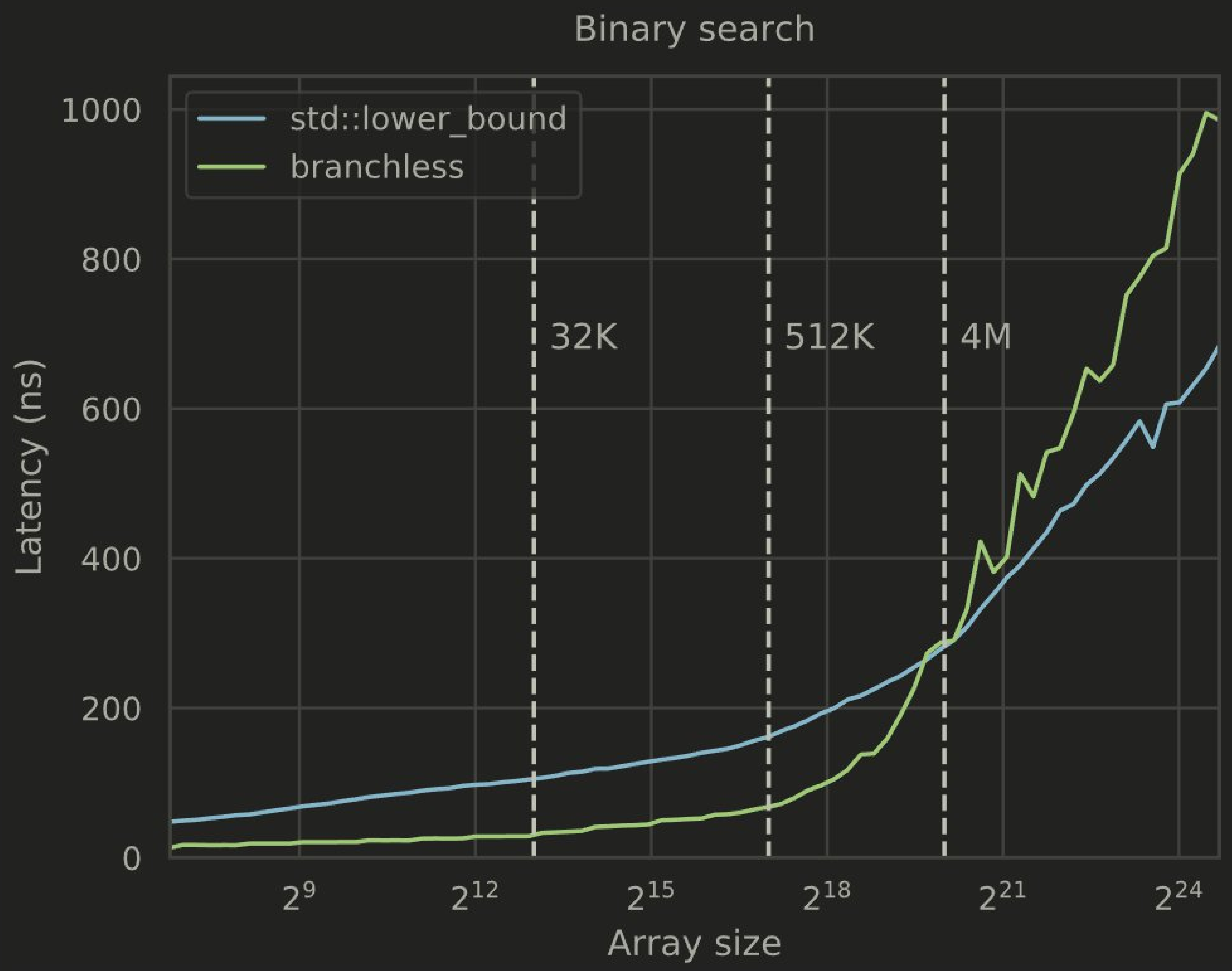
```c
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        if (base[half - 1] < x) {
            base += half;
            len = len - half; // = ceil(len / 2)
        } else {
            len = half; // = floor(len / 2)
        }
    }
    return *base;
}
```

```
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        if (base[half - 1] < x)
            base += half;
        len -= half; // = ceil(len / 2)
    }
    return *base;
}
```

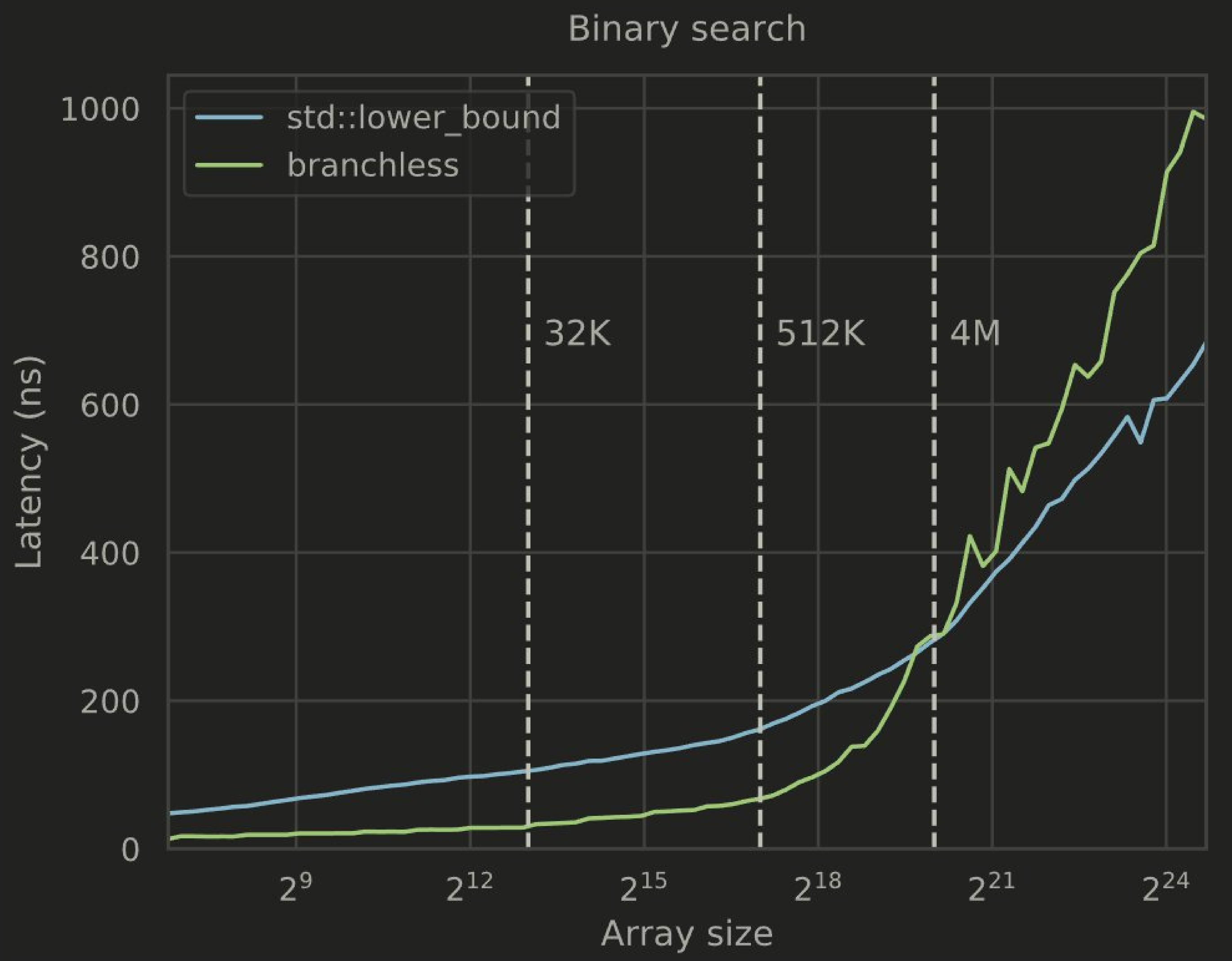
```
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        if (base[half - 1] < x)
            base += half;
        len -= half; // = ceil(len / 2)
    }
    return *base;
}
```

Exactly $\lceil \log_2 n \rceil$ iterations, so not strictly equivalent to binary search

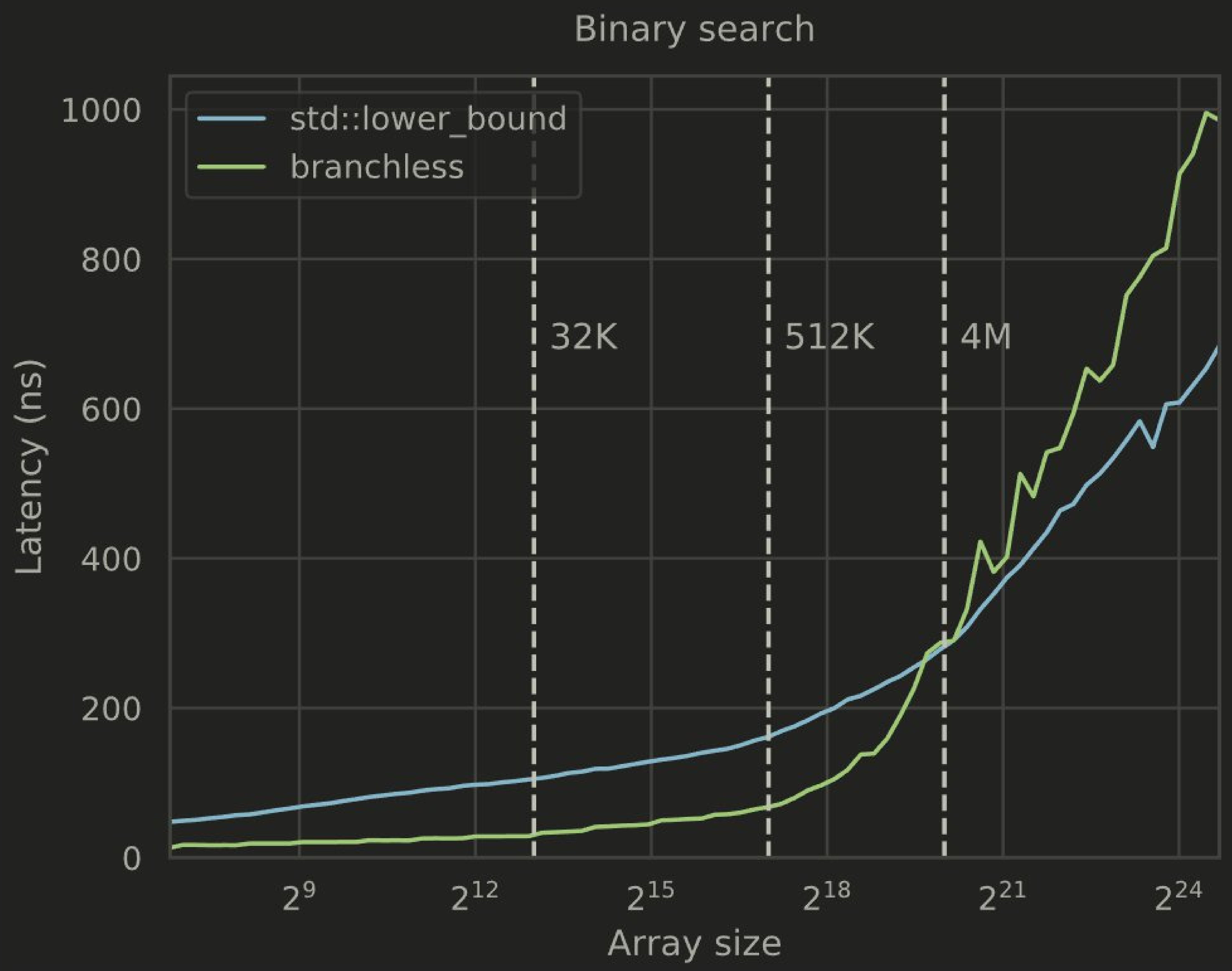
```
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        base += (base[half - 1] < x) * half; // will be replaced with a "cmov"
        len -= half;
    }
    return *base;
}
```



Binary search
std::lower_bound
branchless
32K
512K
4M
Latency (ns)
1000
800
600
400
200
0
2^9
2^{12}
2^{15}
2^{18}
2^{21}
2^{24}
Array size



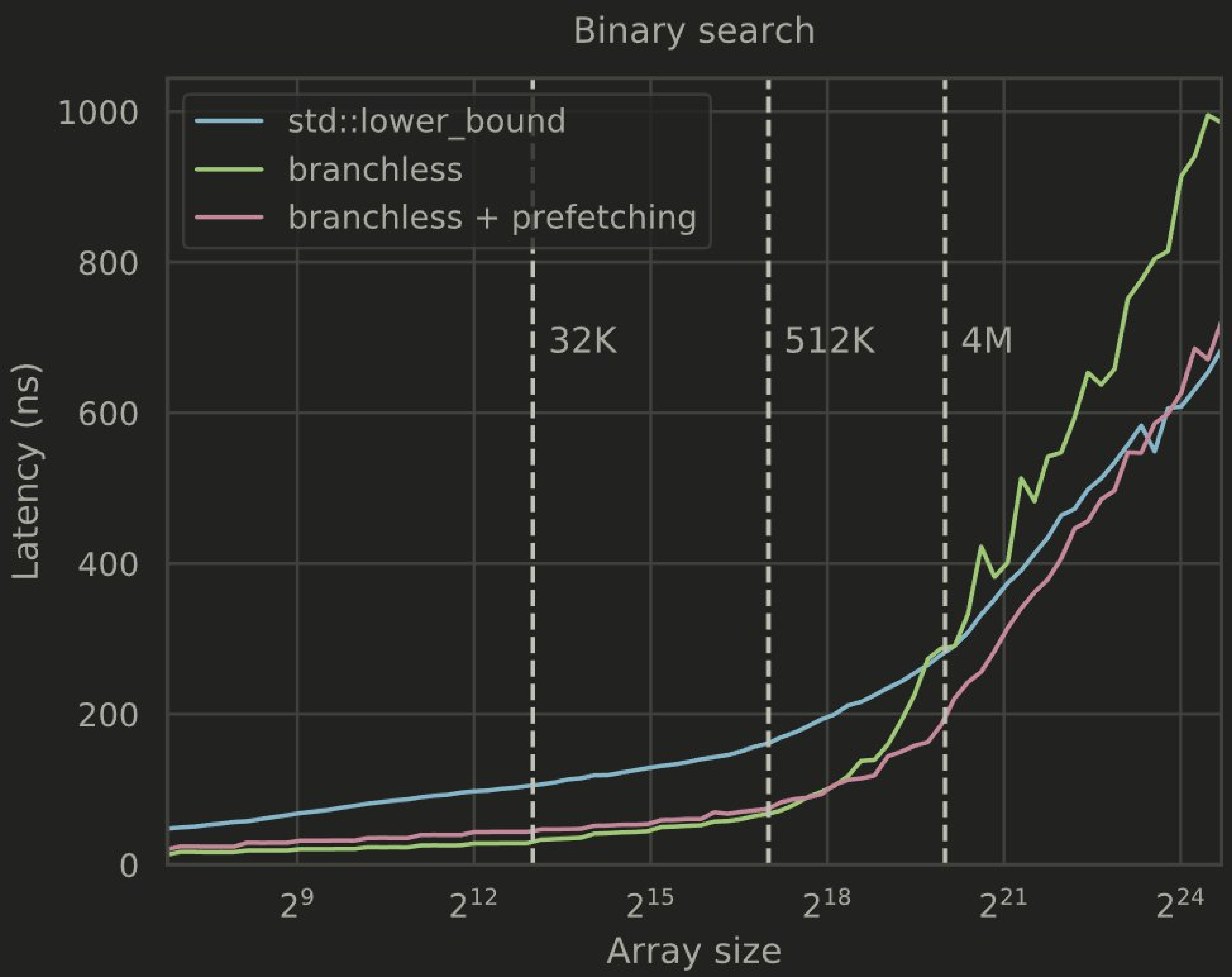
Why worse on larger arrays?



Why worse on larger arrays?

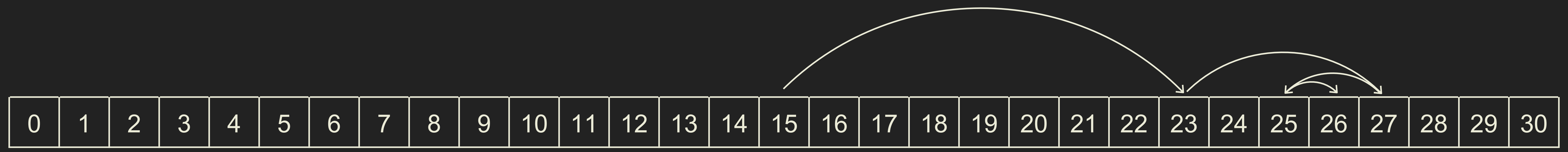
Prefetching (speculation)

```
int lower_bound(int x) {
    int *base = t, len = n;
    while (len > 1) {
        int half = len / 2;
        len -= half;
        __builtin_prefetch(&base[len / 2 - 1]);        // middle of the left half
        __builtin_prefetch(&base[half + len / 2 - 1]); // middle of the right half
        base += (base[half - 1] < x) * half;
    }
    return *base;
}
```

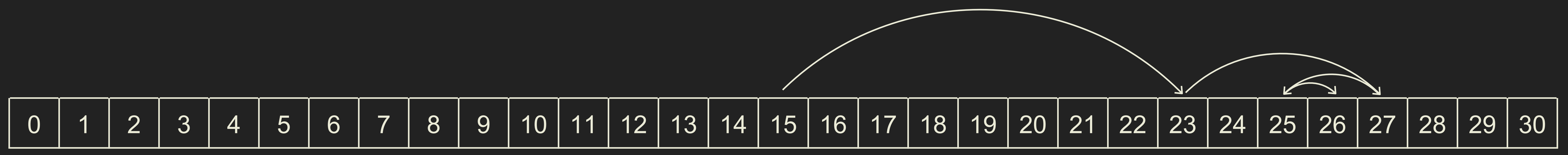


Still grows slightly faster as the branchy version prefetches further than one step ahead

CACHE LOCALITY

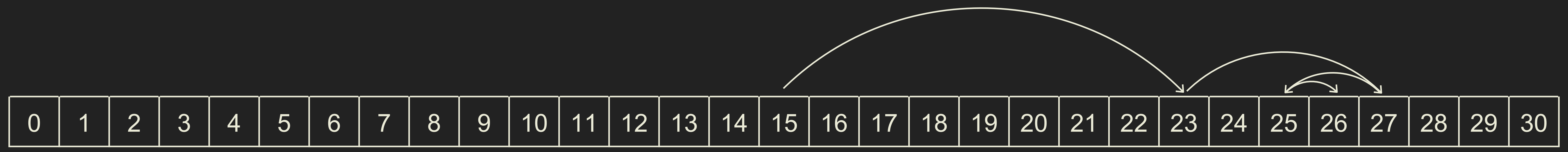


CACHE LOCALITY



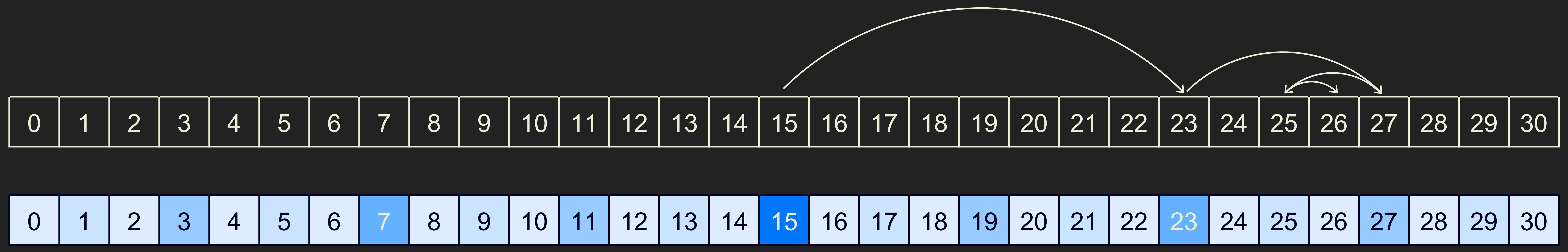
- *Temporal locality*: only okay for the first dozen or so requests

CACHE LOCALITY



- *Temporal locality*: only okay for the first dozen or so requests
- *Spatial locality*: only okay for the last 4-5 requests

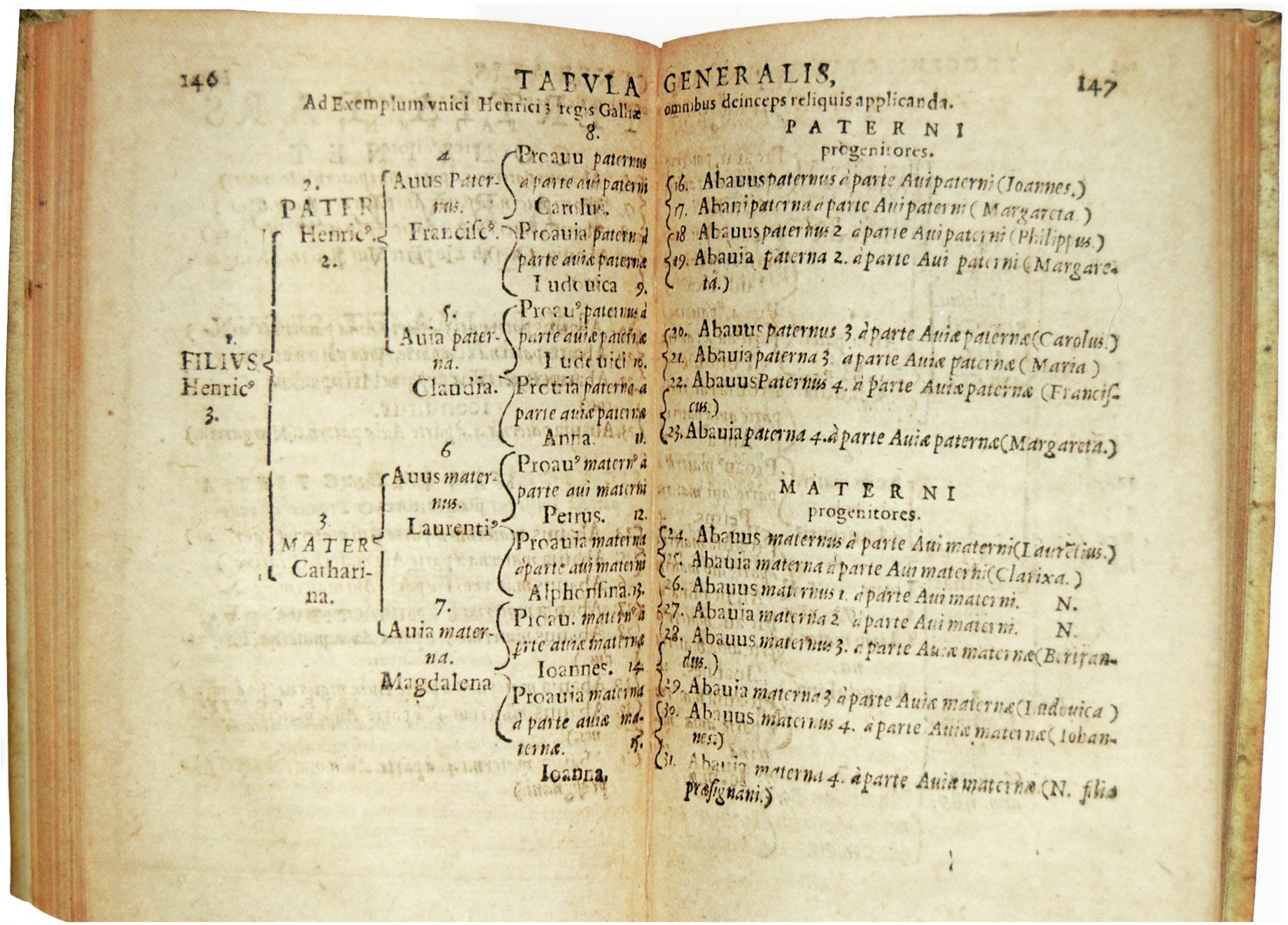
(Memory is fetched in 64B blocks called *cache lines*)



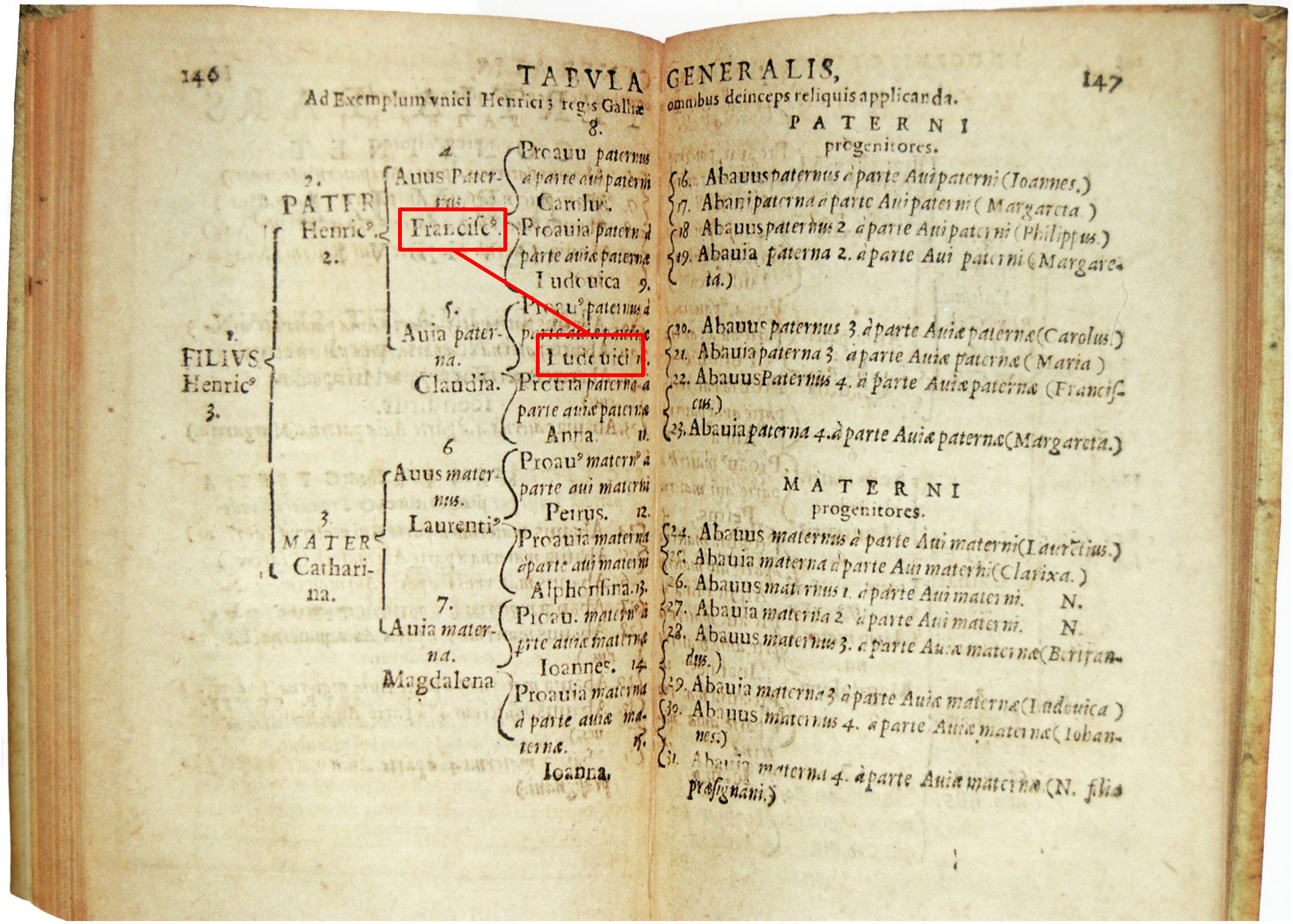
Hot and cold elements are grouped together
(the opposite of what we want)

Michaël Eytzinger is a 16th-century Austrian nobleman known for his work on geneology

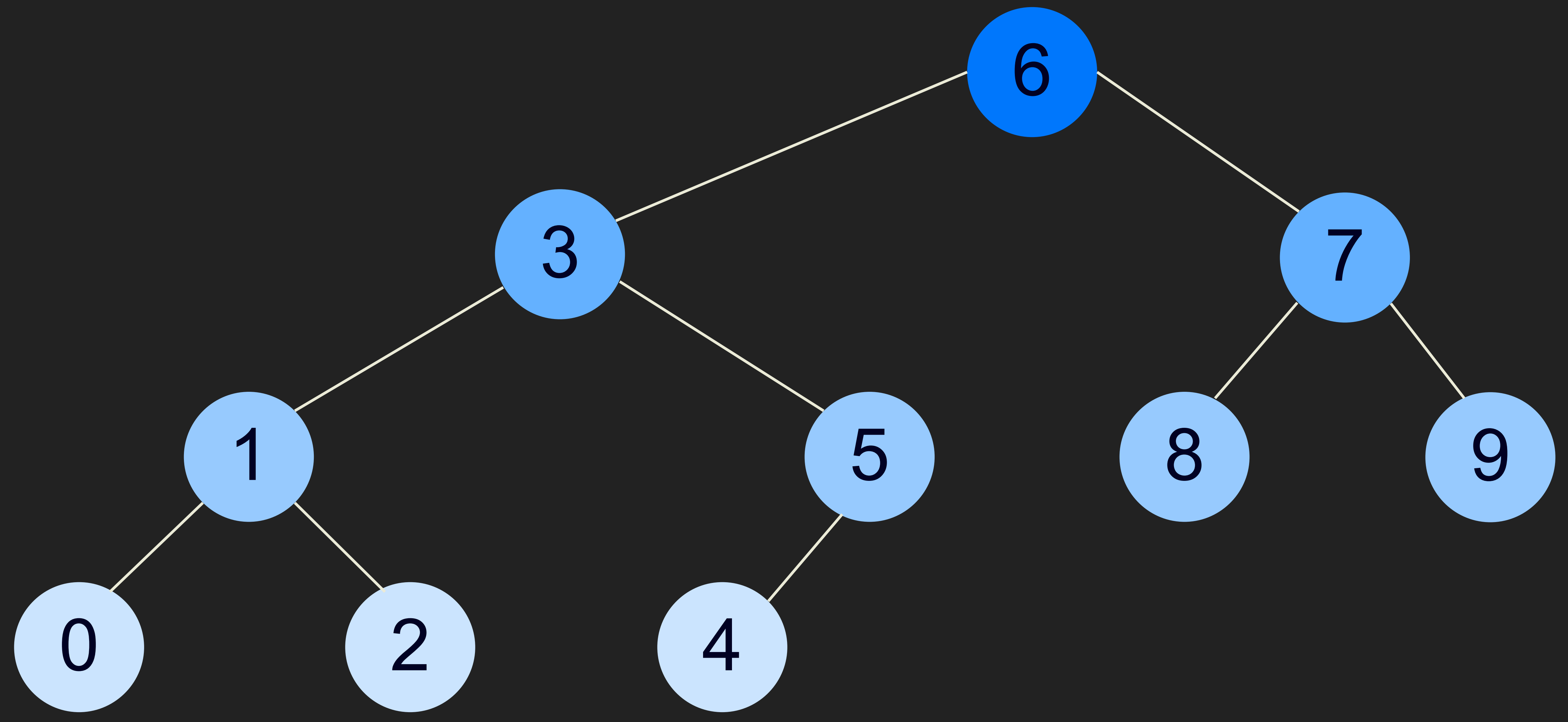
In particular, for a compact ancestor numbering system known as *ahnentafel* (German for “ancestor table”)



- The subject of the ahnentafel is listed as number 1
- The father of person k is listed as $2k$, and their mother is listed as $(2k + 1)$

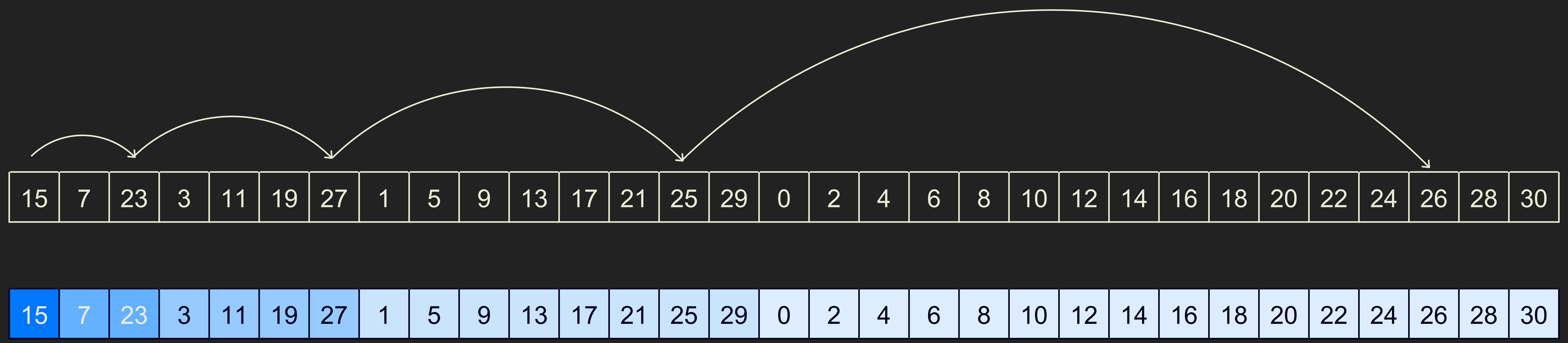


- The subject of the ahnentafel is listed as number 1
- The father of person k is listed as $2k$, and their mother is listed as $(2k + 1)$



6	3	7	1	5	8	9	0	2	4

Used in heaps, segment trees, and other binary tree structures



Temporal locality is much better

BUILDING

```
int a[n], t[n + 1]; // the original sorted array and the eytzinger array we build
//              ^ we need one element more because of one-based indexing
void eytzinger(int k = 1) {
    static int i = 0;
    if (k <= n) {
        eytzinger(2 * k);
        t[k] = a[i++];
        eytzinger(2 * k + 1);
    }
}
```

The sorted array can be permuted in O(n) time

SEARCHING

```
int k = 1;
while (k <= n)
    k = 2 * k + (t[k] < x);
```

SEARCHING

```
int k = 1;
while (k <= n)
    k = 2 * k + (t[k] < x);
```

But how do we get the lower bound itself?

```
    array:  0 1 2 3 4 5 6 7 8 9
eytzinger:  6 3 7 1 5 8 9 0 2 4
1st range:  ------------?------  k := 2*k     = 2   (6 ≥ 3)
2nd range:  ------?------        k := 2*k     = 4   (3 ≥ 3)
3rd range:  --?----              k := 2*k + 1 = 9   (1 < 3)
4th range:      ?--              k := 2*k + 1 = 19  (2 < 3)
5th range:        !
```

k does not necessarily point to the lower bound

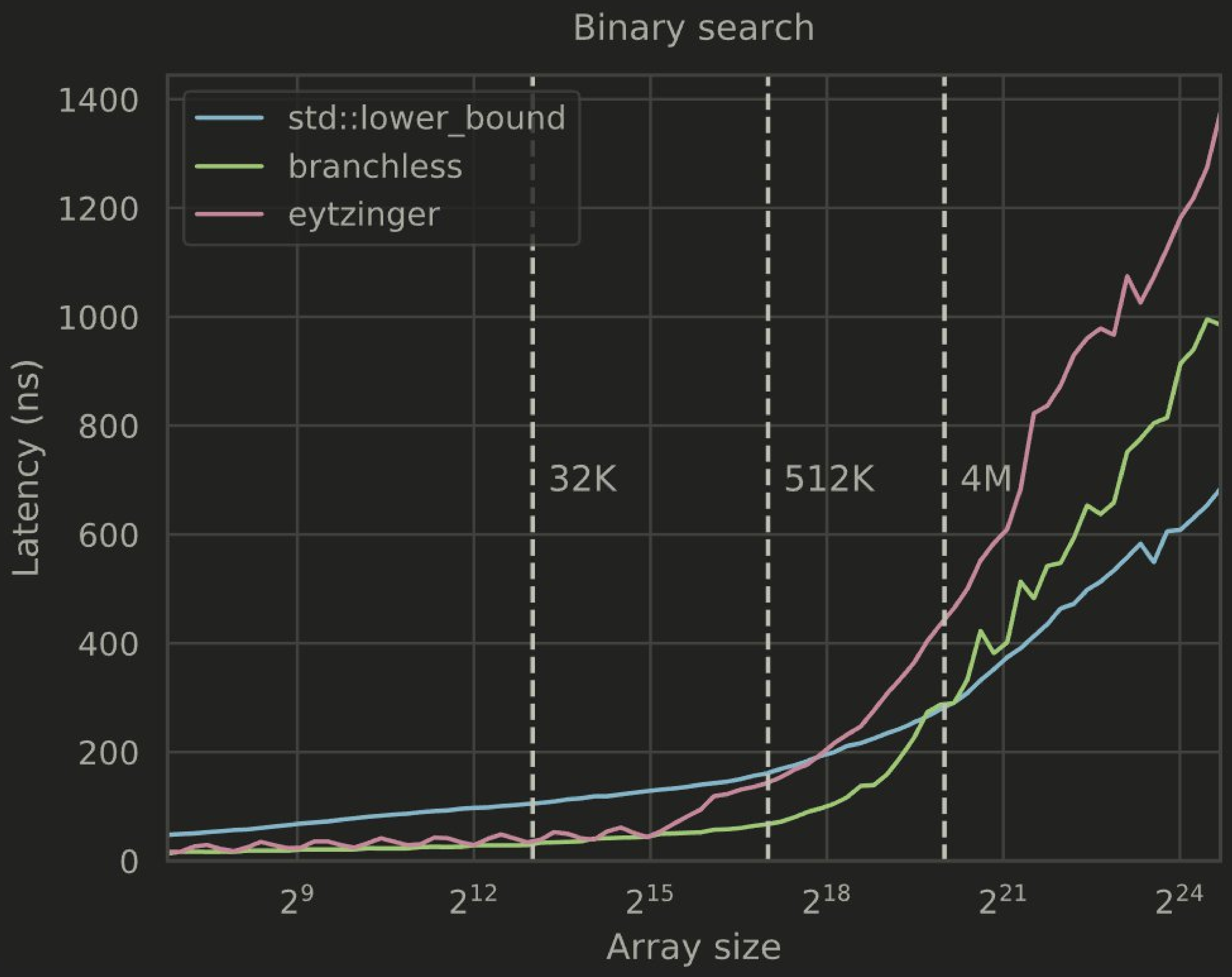
```
    array:  0 1 2 3 4 5 6 7 8 9
eytzinger:  6 3 7 1 5 8 9 0 2 4
1st range:  ------------?------  k := 2*k     = 2   (6 ≥ 3)
2nd range:  ------?------        k := 2*k     = 4   (3 ≥ 3)
3rd range:  --?----              k := 2*k + 1 = 9   (1 < 3)
4th range:      ?--              k := 2*k + 1 = 19  (2 < 3)
5th range:        !
```

k does not necessarily point to the lower bound

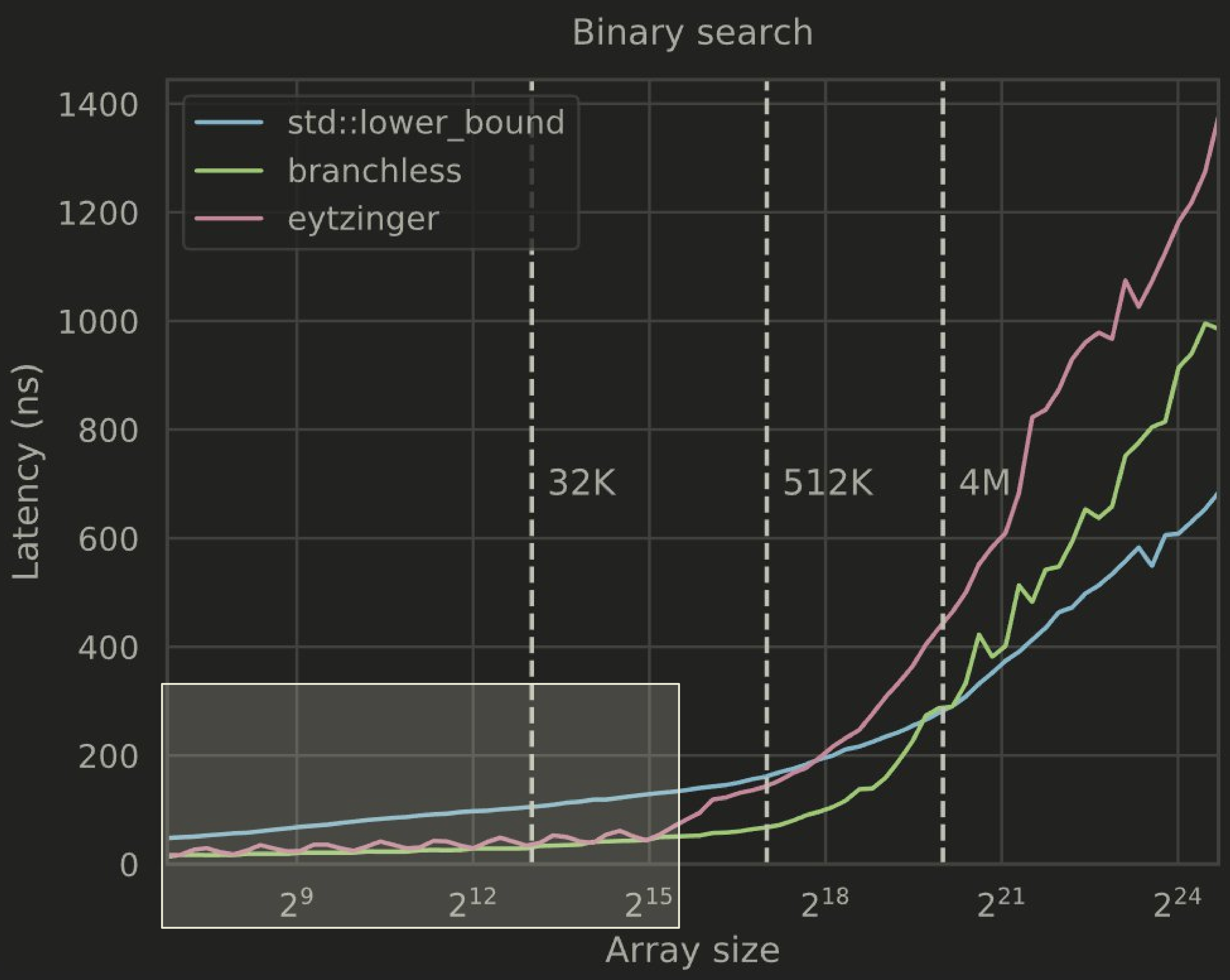
…but we compare against it at some point and keep going “right” ever after

```
int lower_bound(int x) {
    int k = 1;
    while (k <= n)
        k = 2 * k + (t[k] < x);
    k >>= __builtin_ffs(~k);
    return t[k];
}
```

- Right-turns are recorded in the binary notation of k as 1-bits
- To “cancel” them, we can right-shift k by the number of trailing ones



- Worse than branchless on large arrays (the last 4-5 iterations are not cached)



- Worse than branchless on large arrays (the last 4-5 iterations are not cached)
- Also, there is one unpredictable branch (non-constant # of iterations)

PREFETCHING

```
int lower_bound(int x) {
    int k = 1;
    while (k <= n) {
        __builtin_prefetch(&t[k * 2]);
        __builtin_prefetch(&t[k * 2 + 1]); // do we really need both?
        k = 2 * k + (t[k] < x);
    }
    k >>= __builtin_ffs(~k);
    return t[k];
}
```

PREFETCHING

```
int lower_bound(int x) {
    int k = 1;
    while (k <= n) {
        __builtin_prefetch(&t[k * 2]);
        // __builtin_prefetch(&t[k * 2 + 1]);
        k = 2 * k + (t[k] < x);
    }
    k >>= __builtin_ffs(~k);
    return t[k];
}
```

PREFETCHING

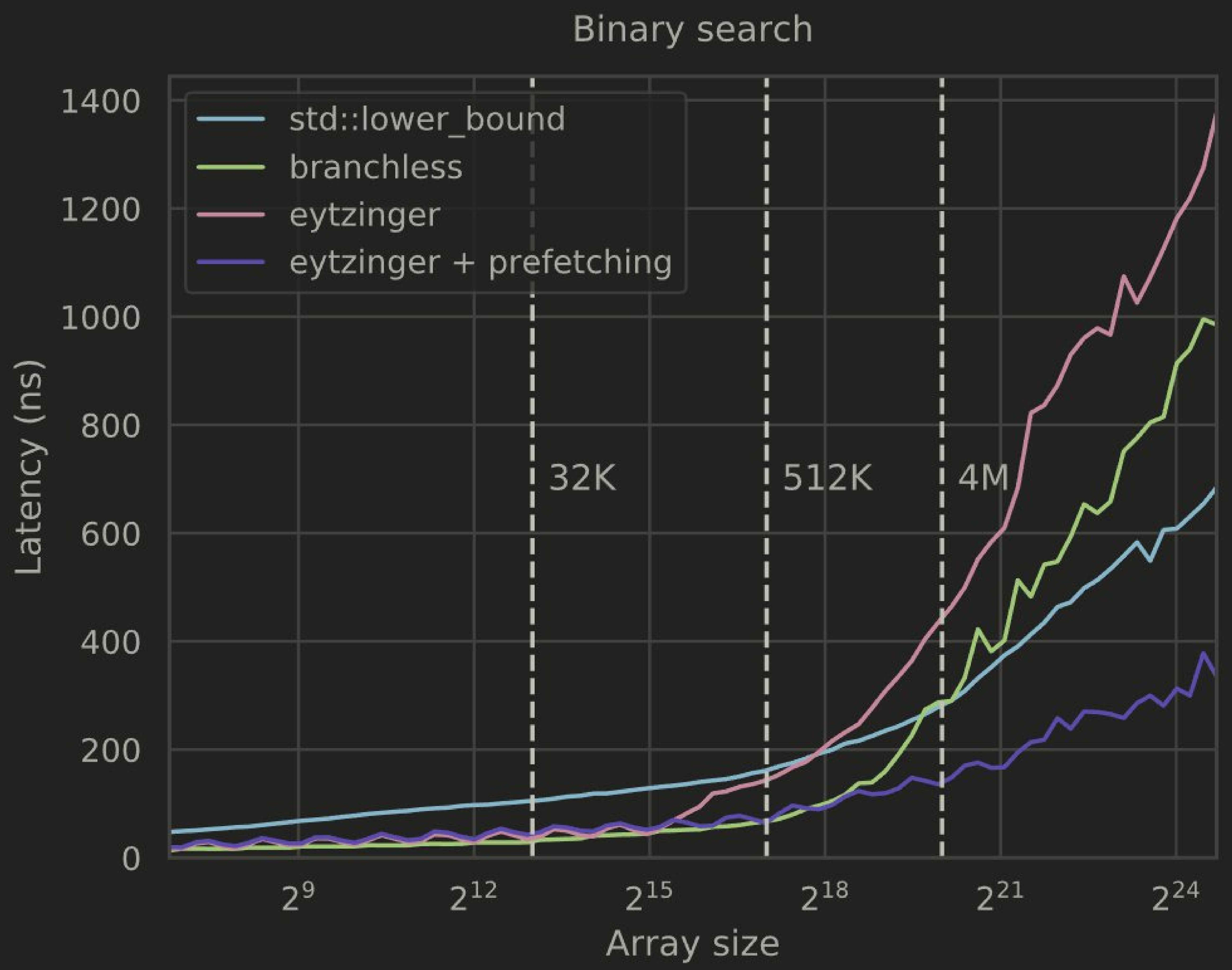
```
int lower_bound(int x) {
    int k = 1;
    while (k <= n) {
        __builtin_prefetch(&t[k * 4]);
        __builtin_prefetch(&t[k * 4 + 1]);
        __builtin_prefetch(&t[k * 4 + 2]);
        __builtin_prefetch(&t[k * 4 + 3]);
        k = 2 * k + (t[k] < x);
    }
    k >>= __builtin_ffs(~k);
    return t[k];
}
```

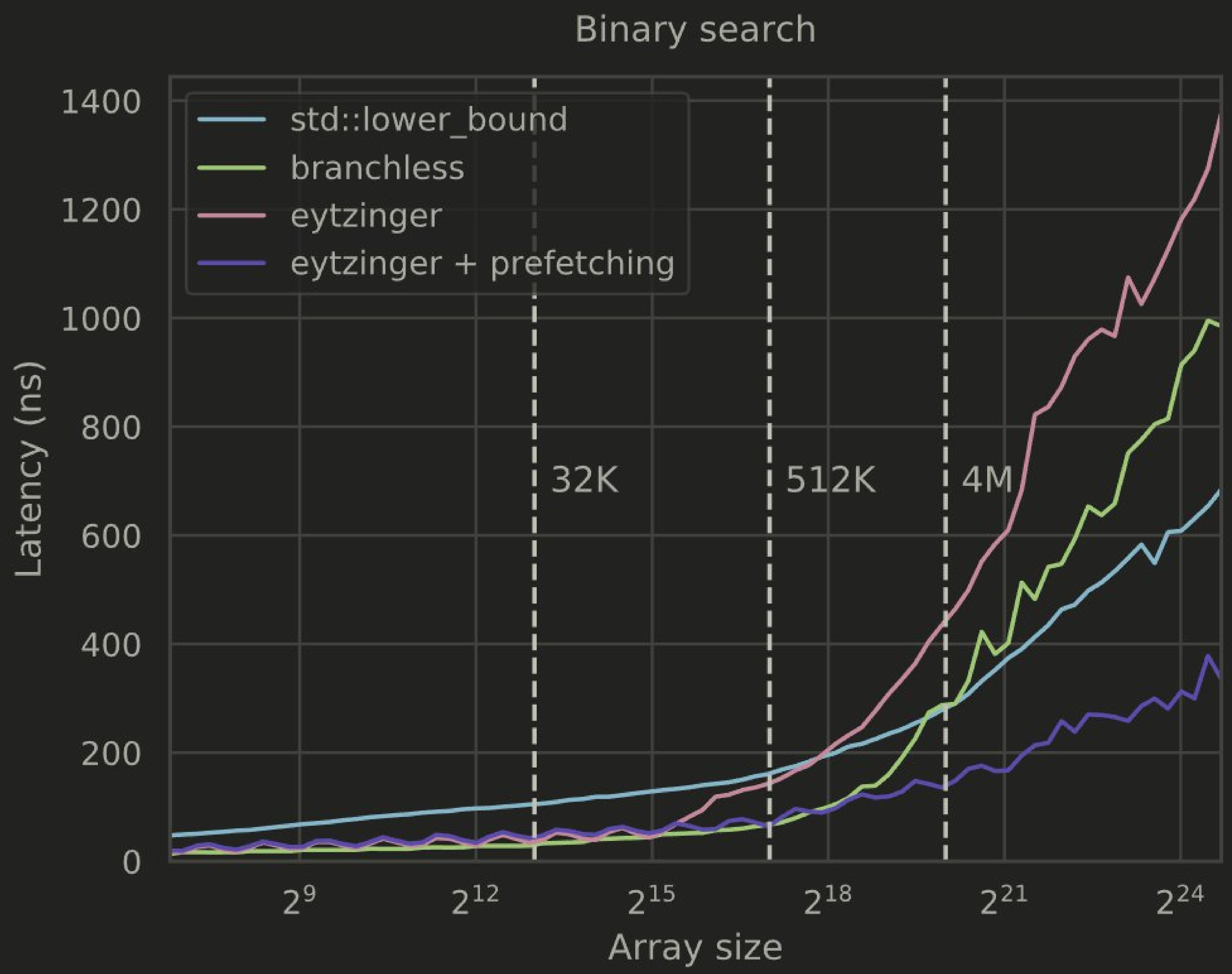
PREFETCHING

```
int lower_bound(int x) {
    int k = 1;
    while (k <= n) {
        __builtin_prefetch(&t[k * 4]);
        // __builtin_prefetch(&t[k * 4 + 1]);
        // __builtin_prefetch(&t[k * 4 + 2]);
        // __builtin_prefetch(&t[k * 4 + 3]);
        k = 2 * k + (t[k] < x);
    }
    k >>= __builtin_ffs(~k);
    return t[k];
}
```

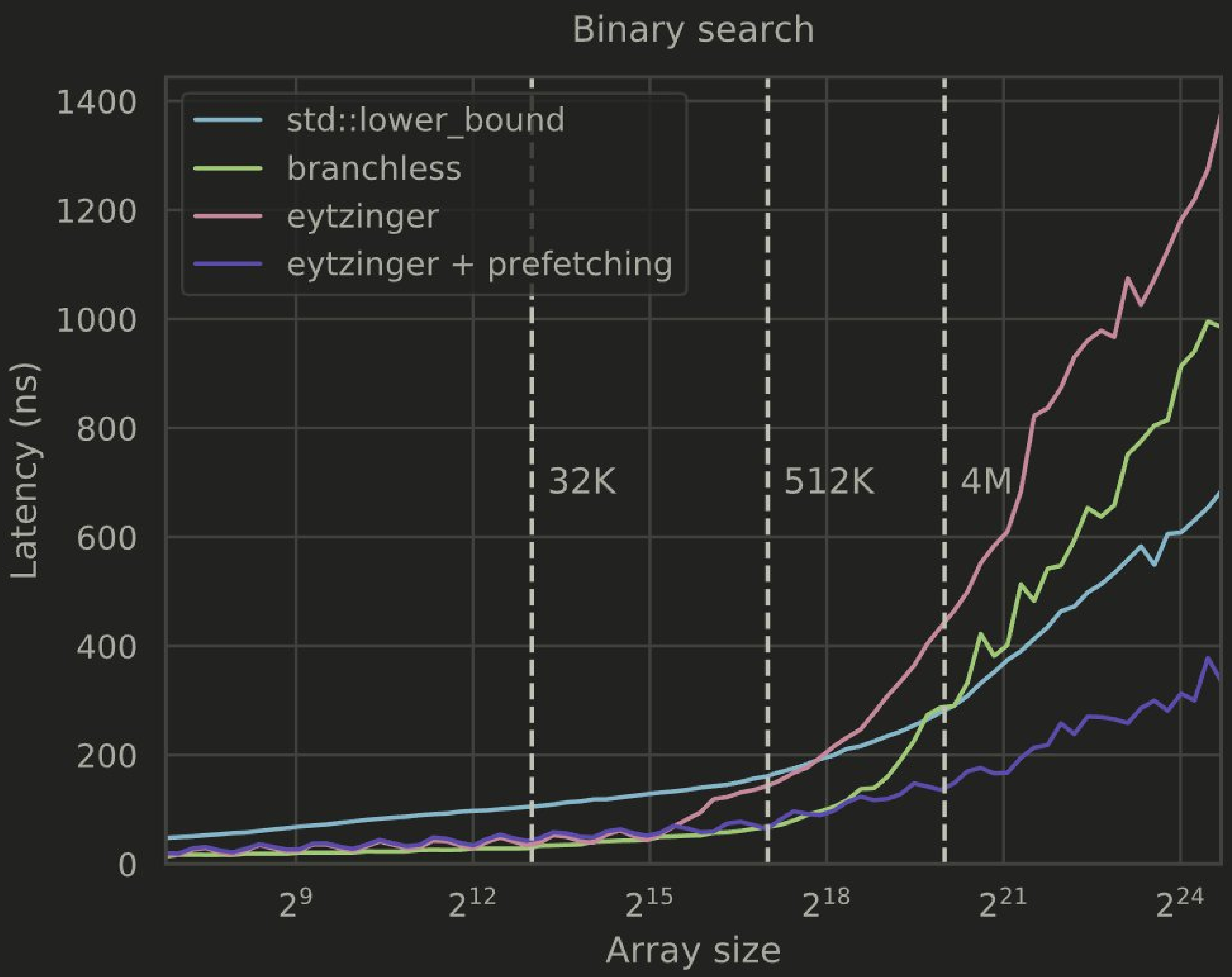
PREFETCHING

```
alignas(64) int t[n + 1];
//      ^ allocate the array on the beginning of a cache line

int lower_bound(int x) {
    int k = 1;
    while (k <= n) {
        __builtin_prefetch(&t[k * 16]);
        k = 2 * k + (t[k] < x);
    }
    k >>= __builtin_ffs(~k);
    return t[k];
}
```


Binary search
std::lower_bound
branchless
eytzinger
eytzinger + prefetching
32K
512K
4M
Latency (ns)
0
200
400
600
800
1000
1200
1400
2^{9}
2^{12}
2^{15}
2^{18}
2^{21}
2^{24}
Array size

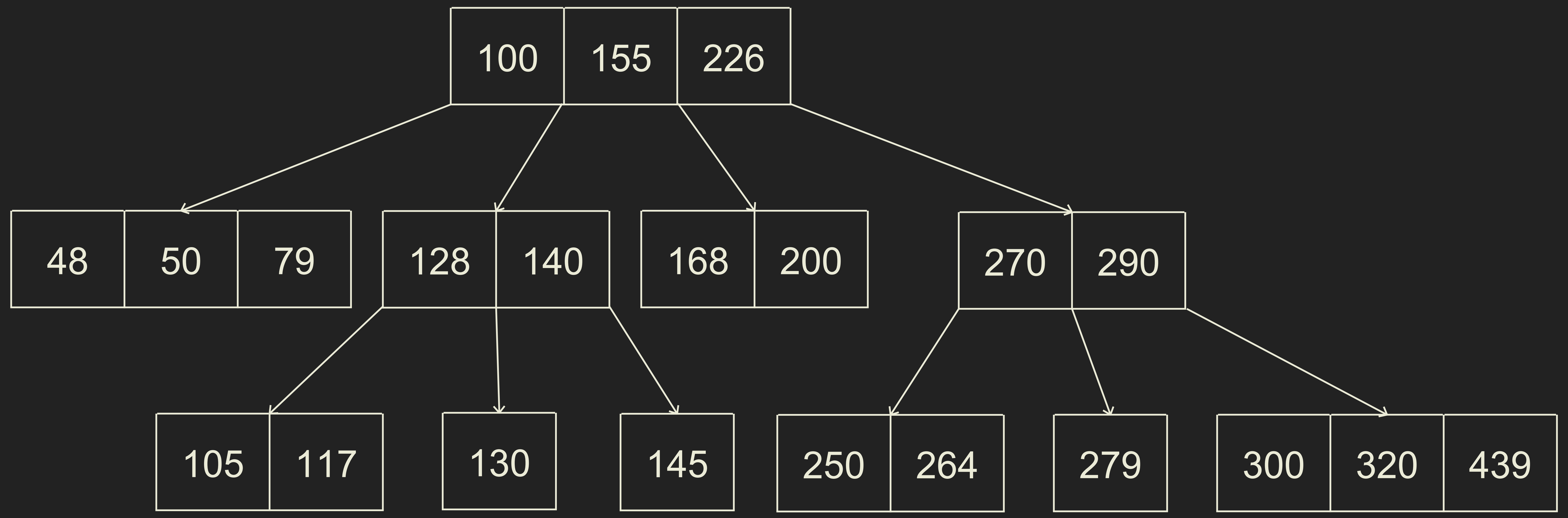


- In Eytzinger binary search, we trade off **bandwidth** for **latency** with prefetching
- This doesn't work well in bandwidth-constrained environments (e.g., multi-threading)



- In Eytzinger binary search, we trade off **bandwidth** for **latency** with prefetching
- This doesn't work well in bandwidth-constrained environments (e.g., multi-threading)
- Instead, we can fetch *fewer* cache lines

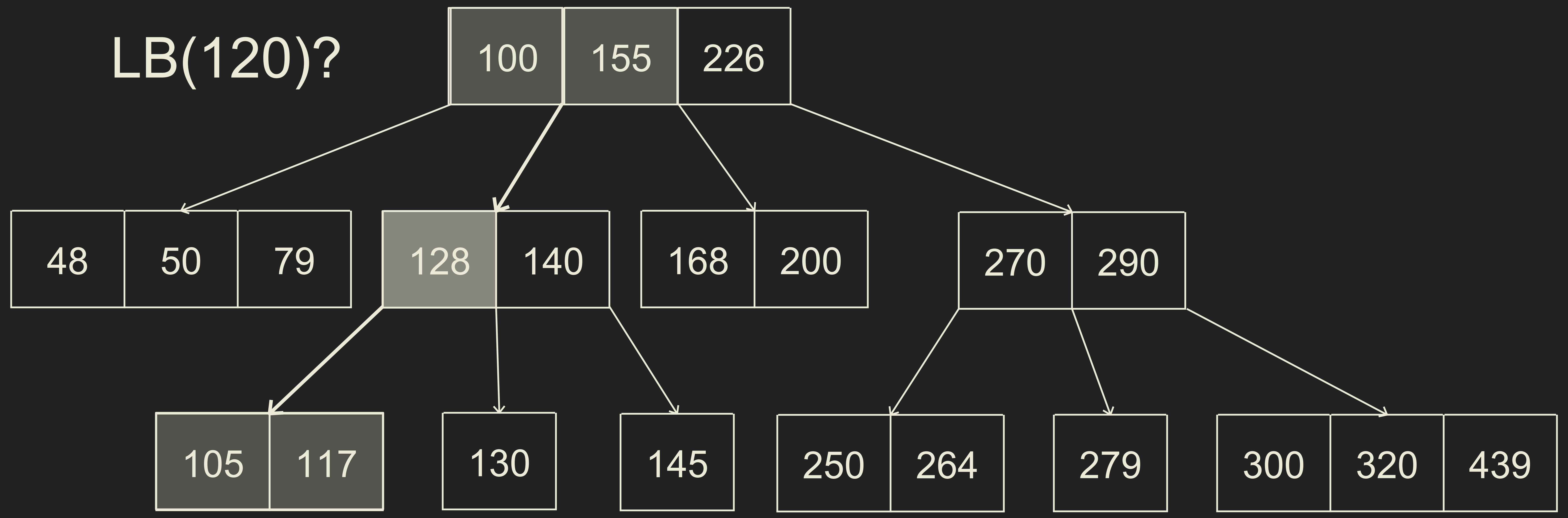
B-TREES



$B \geq 2$ keys in each node,
increasing branching factor and reducing tree height

(at the cost of having to perform B comparisons in each node)

B-TREES



$B \geq 2$ keys in each node,
increasing branching factor and reducing tree height

(at the cost of having to perform B comparisons in each node)

- For RAM, set B to be the cache line size so that we don't waste reads

- B = 16 reduces the tree height by $\frac{\log_2 n}{\log_{17} n} = \frac{\log 17}{\log 2} = \log_2 17 \approx 4.09$ times

We can make static B-trees **implicit**
by generalizing the Eytzinger numeration:

- The root node is numbered 0
- Node k has (B + 1) child nodes numbered $\{k \cdot (B + 1) + i + 1\}$ for $i \in [0, B]$

```
const int B = 16, nblocks = (n + B - 1) / B;
int btree[nblocks][B];

int go(int k, int i) { return k * (B + 1) + i + 1; }

void build(int k = 0) {
    static int t = 0;
    while (k < nblocks) {
        for (int i = 0; i < B; i++) {
            build(go(k, i));
            btree[k][i] = (t < n ? a[t++] : INT_MAX);
        }
        build(go(k, B));
    }
}
```

```c
// compute the "local" lower bound in a node
int rank(int x, int *node) {
    for (int i = 0; i < B; i++)
        if (node[i] >= x)
            return i;
    return B;
}
```

$B/2$ comparisons on average, unpredictable branching

```
int rank(int x, int *node) {
    int mask = (1 << B);

    for (int i = 0; i < B; i++)
        mask |= (btree[k][i] >= x) << i;

    return __builtin_ffs(mask) - 1;
}
```

No branching, but even more comparisons

```
    y = 4        17       65       103
    x = 42       42       42       42
y ≥ x = 00000000 00000000 11111111 11111111

movemask = 0011

     ffs = 3
```

SIMD: Single Instruction, Multiple Data

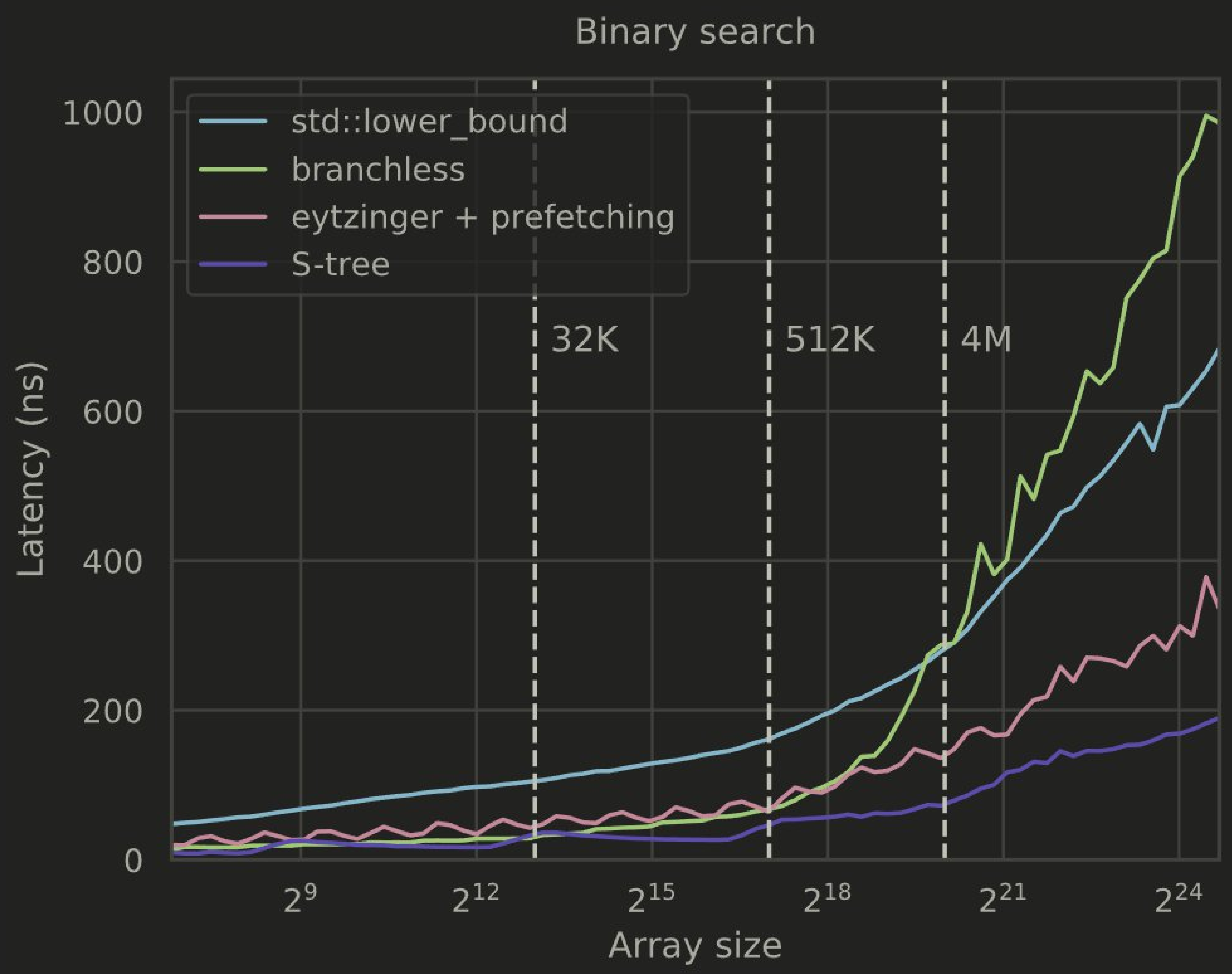
We can compare a contiguous block of node keys against the search key in one go

```
typedef __m256i reg;

// compute a 8-bit mask corresponding to "<" elements
int cmp(reg x_vec, int* y_ptr) {
    reg y_vec = _mm256_load_si256((reg*) y_ptr); // load 8 sorted elements
    reg mask = _mm256_cmpgt_epi32(x_vec, y_vec); // compare against the key
    return  _mm256_movemask_ps((__m256)  mask);  // extract the 8-bit mask
}
```

Zen 2 supports AVX2, so we can compare 256/32 = 8 keys at a time

```c
int rank(reg x_vec, int *node) {
    int mask = ~(
        cmp(x, node) +
        (cmp(x, node + 8) << 8)
    );

    return __builtin_ffs(mask) - 1; // alternative: popcount
}
```

```
int lower_bound(int _x) {
    int k = 0, res = INT_MAX;
    reg x = _mm256_set1_epi32(_x);
    while (k < nblocks) {
        int i = rank(x, btree[k]);
        if (i < B) // a local lower bound may not exist in the leaf node
            res = btree[k][i];
        k = go(k, i);
    }
    return res;
}
```



"S-tree": static, succinct, small, speedy, SIMDized

```
int lower_bound(int _x) {
    int k = 0, res = INT_MAX;
    reg x = _mm256_set1_epi32(_x);
    while (k < nblocks) {
        int i = rank(x, btree[k]);
        if (i < B)
            res = btree[k][i];
        k = go(k, i);
    }
    return res;
}
```

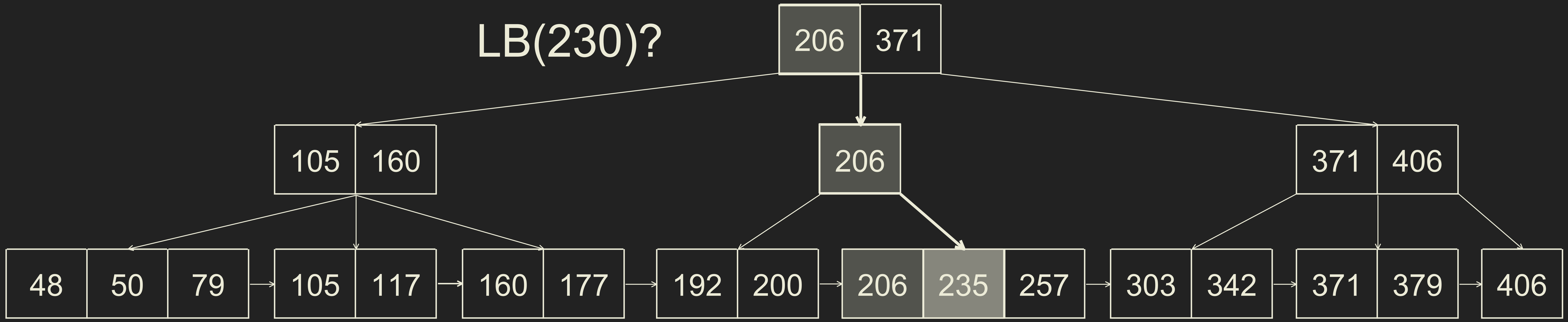
```
int lower_bound(int _x) {
    int k = 0, res = INT_MAX;
    reg x = _mm256_set1_epi32(_x);
    while (k < nblocks) {
        int i = rank(x, btree[k]);
        if (i < B)
            res = btree[k][i];
        k = go(k, i);
    }
    return res;
}
```

1. The update is unnecessary 16 times out of 17

```
int lower_bound(int _x) {
    int k = 0, res = INT_MAX;
    reg x = _mm256_set1_epi32(_x);
    while (k < nblocks) {
        int i = rank(x, btree[k]);
        if (i < B)
            res = btree[k][i];
        k = go(k, i);
    }
    return res;
}
```

1. The update is unnecessary 16 times out of 17
2. Non-constant number of iterations causes branch mispredictions

B+ TREES



- *Internal nodes* store up to B keys and (B + 1) pointers to child nodes
- *Data nodes* or *leaves* store up to B keys (and the pointer to the next leaf node)
- Keys are **copied** to upper layers (the structure is not succinct)

```
int lower_bound(int _x) {
    unsigned k = 0;
    reg x = _mm256_set1_epi32(_x - 1);
    for (int h = H - 1; h > 0; h--) {
        unsigned i = rank(x, btree + offset(h) + k);
        k = k * (B + 1) + i * B;
    }
    unsigned i = rank(x, btree + k);
    return btree[k + i];
}
```

```
int lower_bound(int _x) {
    unsigned k = 0;
    reg x = _mm256_set1_epi32(_x - 1);
    for (int h = H - 1; h > 0; h--) {
        unsigned i = rank(x, btree + offset(h) + k);
        k = k * (B + 1) + i * B;
    }
    unsigned i = rank(x, btree + k);
    return btree[k + i];
}
```

1. The last node or its next neighbor has the local lower bound (no update)

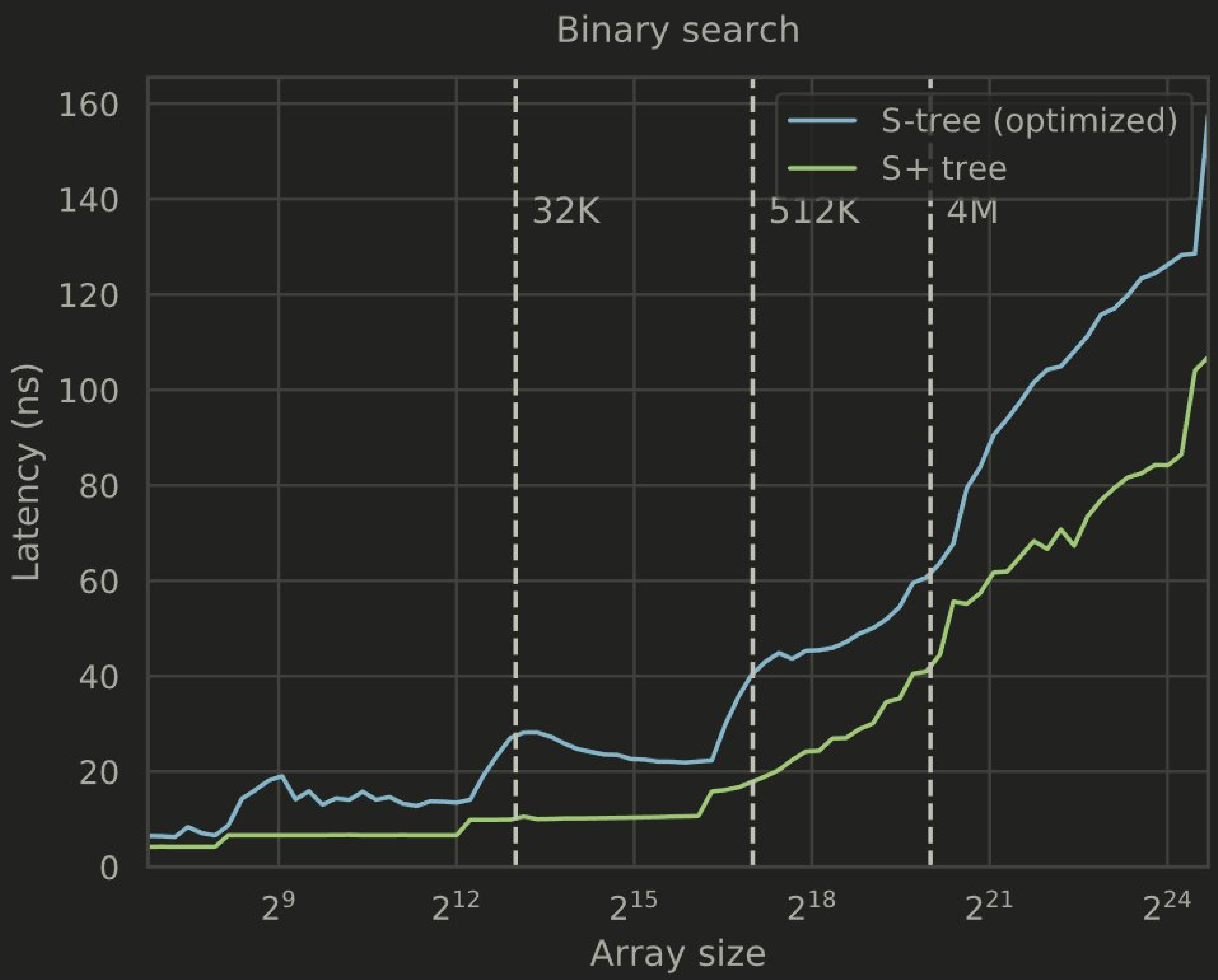
```
int lower_bound(int _x) {
    unsigned k = 0;
    reg x = _mm256_set1_epi32(_x - 1);
    for (int h = H - 1; h > 0; h--) {
        unsigned i = rank(x, btree + offset(h) + k);
        k = k * (B + 1) + i * B;
    }
    unsigned i = rank(x, btree + k);
    return btree[k + i];
}
```

1. The last node or its next neighbor has the local lower bound (no update)
2. The depth is constant as B+ trees grow upwards (no branching)

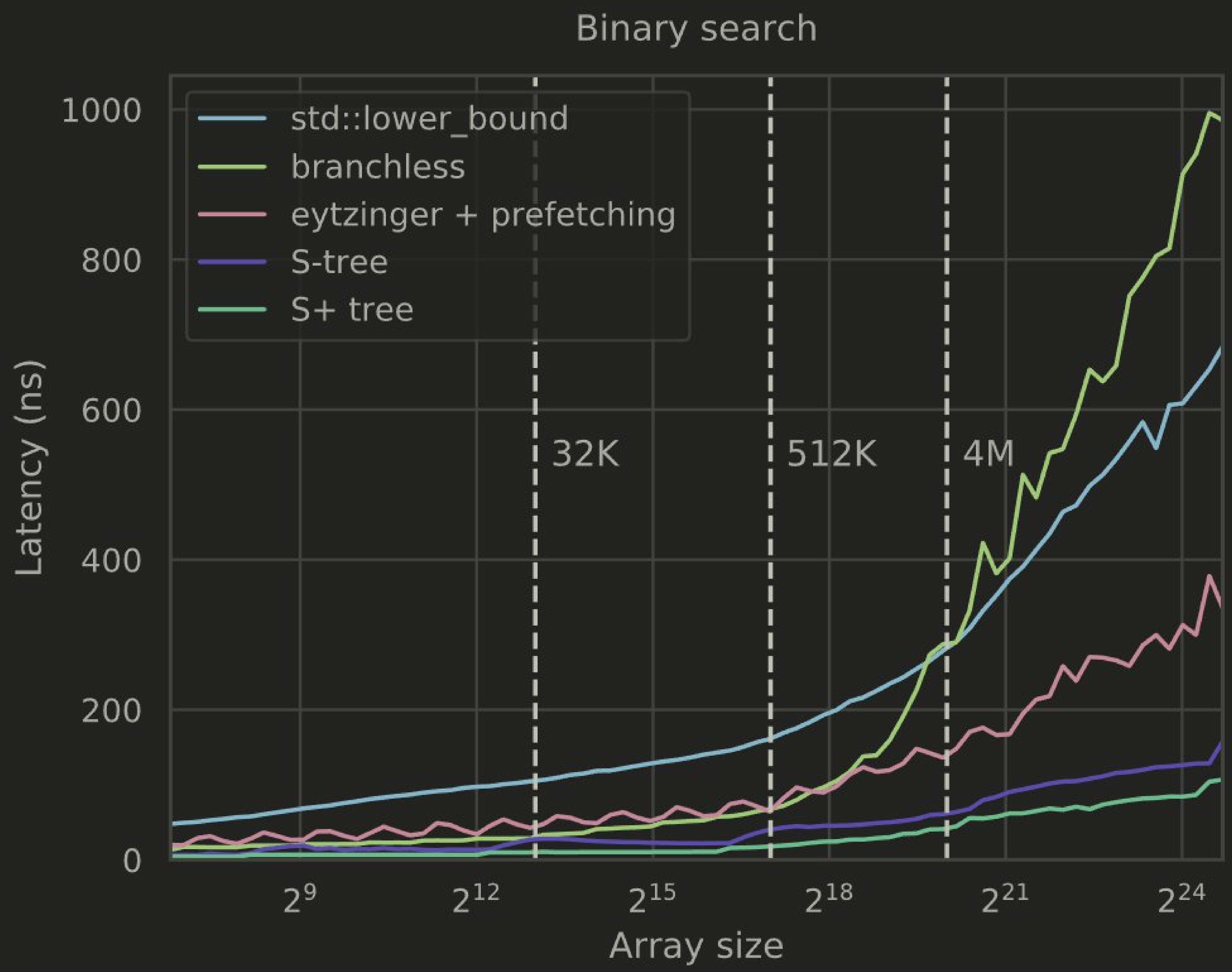
```
int lower_bound(int _x) {
    unsigned k = 0;
    reg x = _mm256_set1_epi32(_x - 1);
    for (int h = H - 1; h > 0; h--) {
        unsigned i = rank(x, btree + offset(h) + k);
        k = k * (B + 1) + i * B;
    }
    unsigned i = rank(x, btree + k);
    return btree[k + i];
}
```

1. The last node or its next neighbor has the local lower bound (no update)
2. The depth is constant as B+ trees grow upwards (no branching)

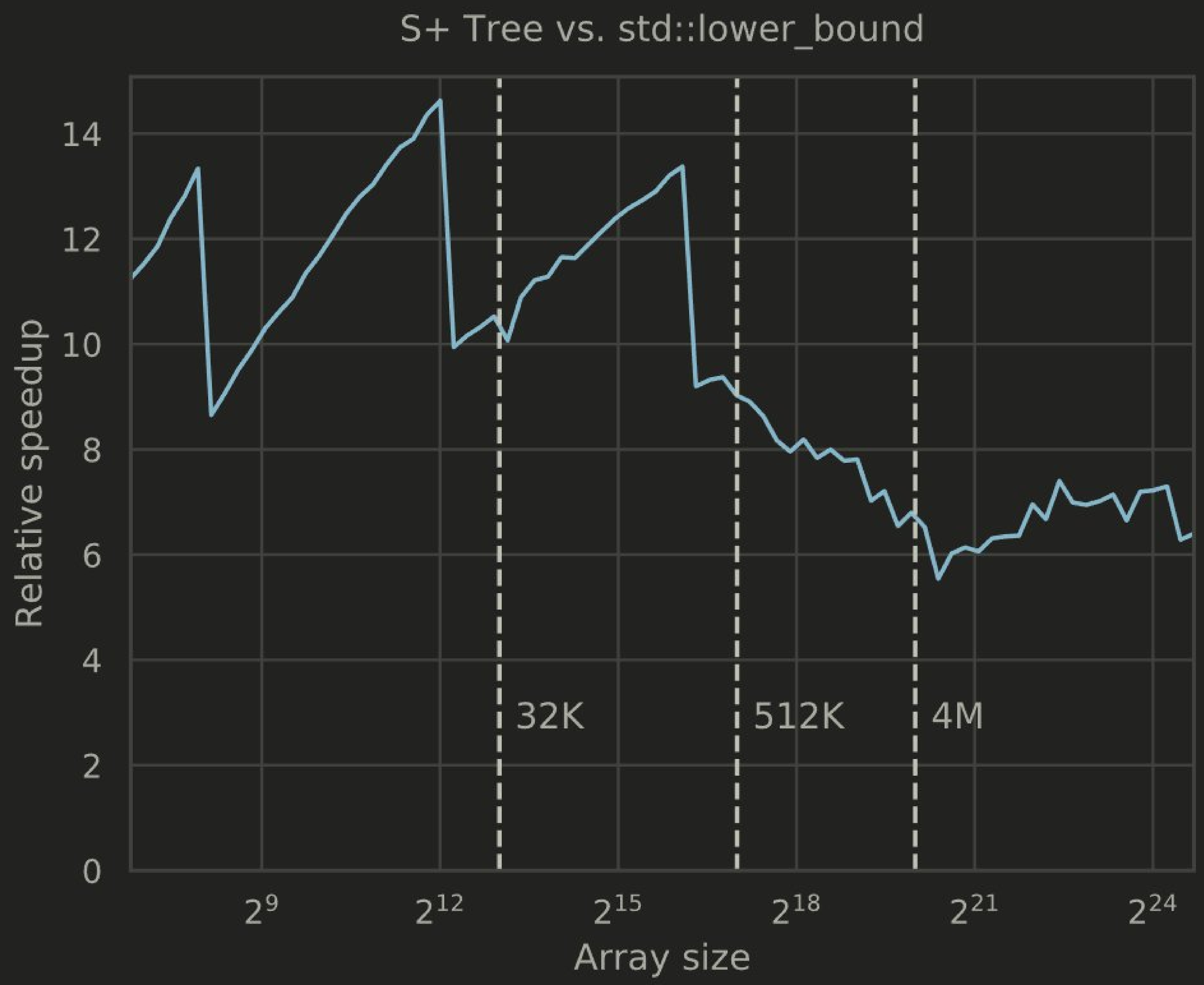
Also, we can now *reuse* the sorted array and easily return the *index* of the lower bound



(Spikes at the end due to L1 TLB spill)



Binary search
std::lower_bound
branchless
eytzinger + prefetching
S-tree
S+ tree
32K
512K
4M
Latency (ns)
0
200
400
600
800
1000
2^9
2^{12}
2^{15}
2^{18}
2^{21}
2^{24}
Array size


S+ Tree vs. std::lower_bound
Relative speedup
0
2
4
6
8
10
12
14
32K
512K
4M
2^9
2^12
2^15
2^18
2^21
2^24
Array size

FUTURE WORK

- Tweak the layout (non-uniform node sizes, “hierarchical blocking”)

FUTURE WORK

- Tweak the layout (non-uniform node sizes, “hierarchical blocking”)
- Partial prefetching

FUTURE WORK

- Tweak the layout (non-uniform node sizes, “hierarchical blocking”)
- Partial prefetching
- Different vector comparison kernels

FUTURE WORK

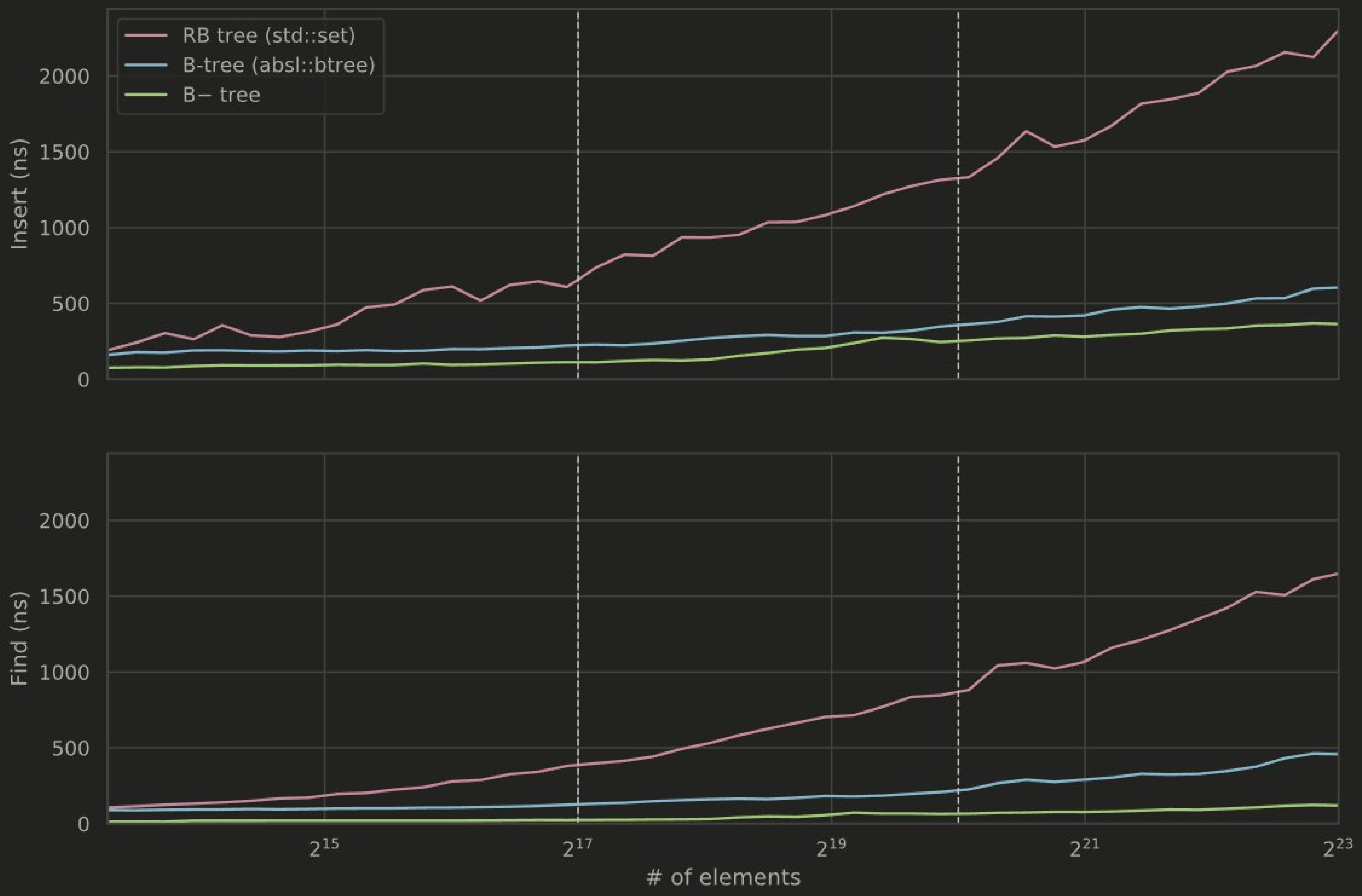
- Tweak the layout (non-uniform node sizes, “hierarchical blocking”)
- Partial prefetching
- Different vector comparison kernels
- Consider non-numerical data types and custom comparators

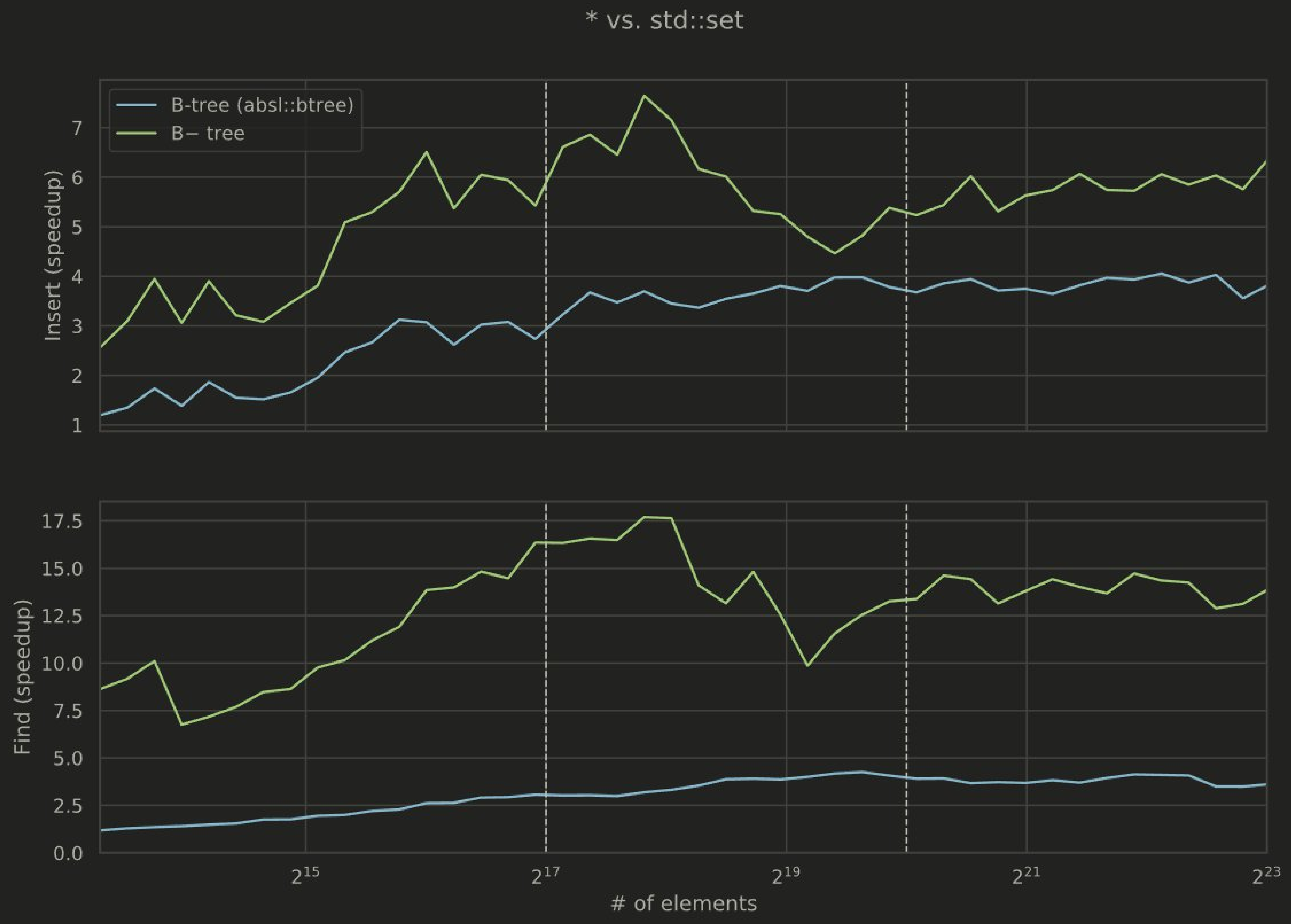
FUTURE WORK

- Tweak the layout (non-uniform node sizes, “hierarchical blocking”)
- Partial prefetching
- Different vector comparison kernels
- Consider non-numerical data types and custom comparators
- Generalize to *dynamic* trees and other tree-like structures

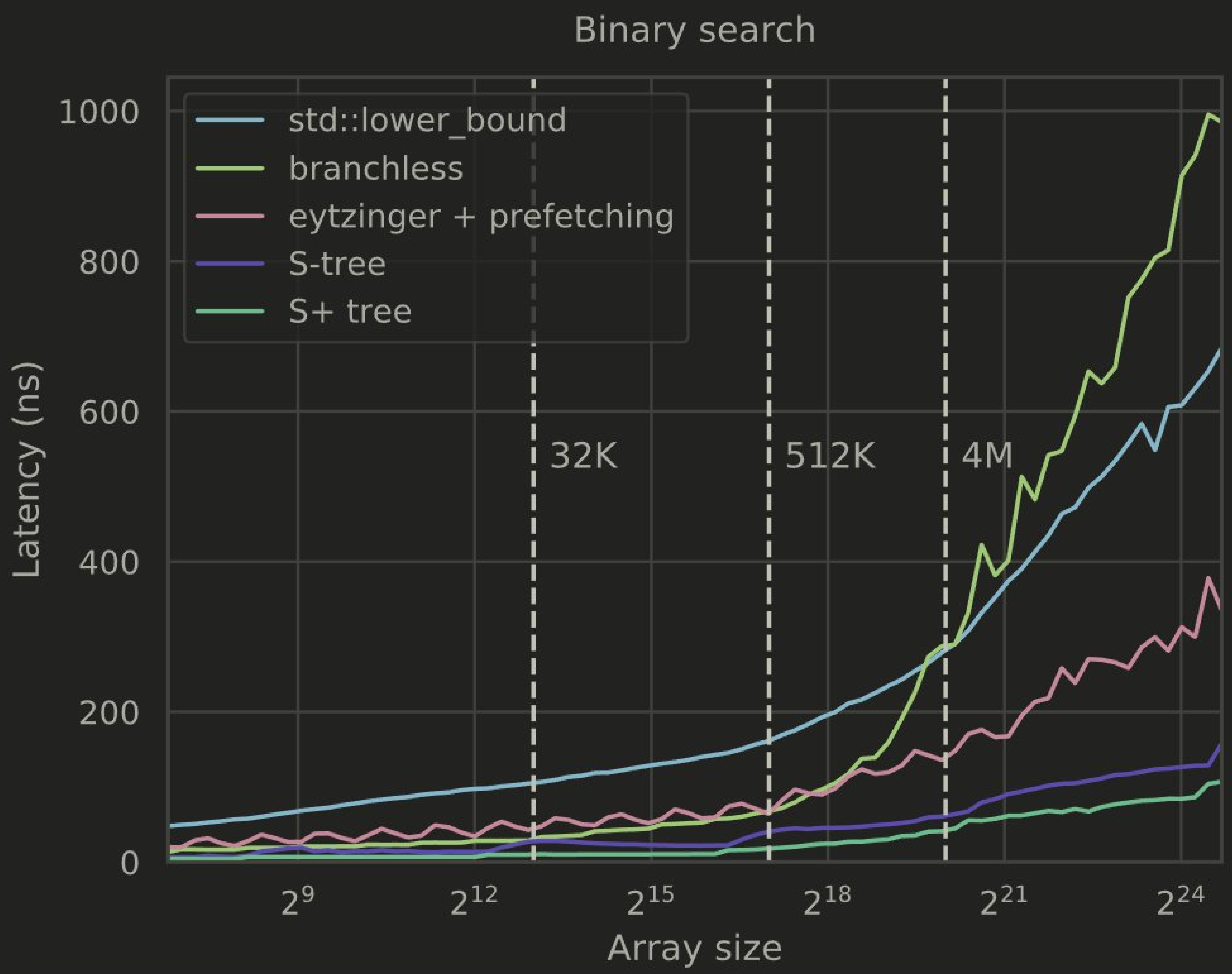
```
int *tree; // internal nodes store (B - 2) keys and (B - 1) pointers
           // leafs store (B - 1) keys

int lower_bound(int _x) {
    unsigned k = root;
    reg x = _mm256_set1_epi32(_x);

    for (int h = 0; h < H - 1; h++) {
        unsigned i = rank(x, &tree[k]);
        k = tree[k + B + i]; // fetch the next pointer
    }
    unsigned i = rank(x, &tree[k]);
    return tree[k + i];
}
```

“B− Tree”


RB tree (std::set)
B-tree (absl::btree)
B− tree
Insert (ns)
0
500
1000
1500
2000
Find (ns)
2^{15}
2^{17}
2^{19}
2^{21}
2^{23}
# of elements


* vs. std::set
B-tree (absl::btree)
B− tree
Insert (speedup)
7
6
5
4
3
2
1
Find (speedup)
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
2^{15}
2^{17}
2^{19}
2^{21}
2^{23}
# of elements

B− tree vs. absl::btree



Article: en.algorithmica.org/hpc/data-structures/binary-search

Code: github.com/sslotin/amh-code/tree/main/binsearch