Modern Microprocessor and Low-Level Optimizations

Ivan Sorokin

About the Talk

Sometimes program optimization requires knowledge of how CPU works:

- Why do these two similar functions differ so much in execution time?
- Why does this minor change affect execution time so much?

The purpose of the talk is to introduce people to how CPU works. In this talk we'll devise a model of a CPU that agrees with some effects observable on real CPUs.

This is an introductory talk.

- Prefer breadth to depth.
- Assembly usage is minimized.

Plan

The algorithm of the talk:

Start with some CPU model.
Repeat:
- Observe some effect that disagrees with the model.
- Refine the model.

Just a Model

Important note: This is just a model.

- It will neither predict nor explain all effects observable on real CPUs
- Intel and AMD publish details only of some aspects of their products.
- About most parts of a CPU little is know for sure.

This is not a problem in practice.

- The most precise model is not necessarily the easiest to work with.
- The most precise model of any CPU is that CPU itself.
- Most models need to be simplified to be operated effectively on.

Intel or AMD

Do we model Intel or AMD CPUs?

- Both!
- Although in 90s there were many different CPU architectures, now even CPUs of different vendor are remarkably similar.

In most cases there are strong reasons why a CPU behaves one way or another.

Initial Model

CPU executes instructions one by one. The duration of the computation is proportional to the number of instructions to be executed.

Example #1

```
char a[N][N];

for (size_t i = 0; i != N; ++i)
   for (size_t j = 0; j != N; ++j)
      a[i][j] = 0;
```

```
char a[N][N];

for (size_t i = 0; i != N; ++i)
   for (size_t j = 0; j != N; ++j)
      a[j][i] = 0;
```

Is there any difference between these two pieces of code?

Example #1

```
char a[N][N];

for (size_t i = 0; i != N; ++i)
   for (size_t j = 0; j != N; ++j)
      a[i][j] = 0;
```

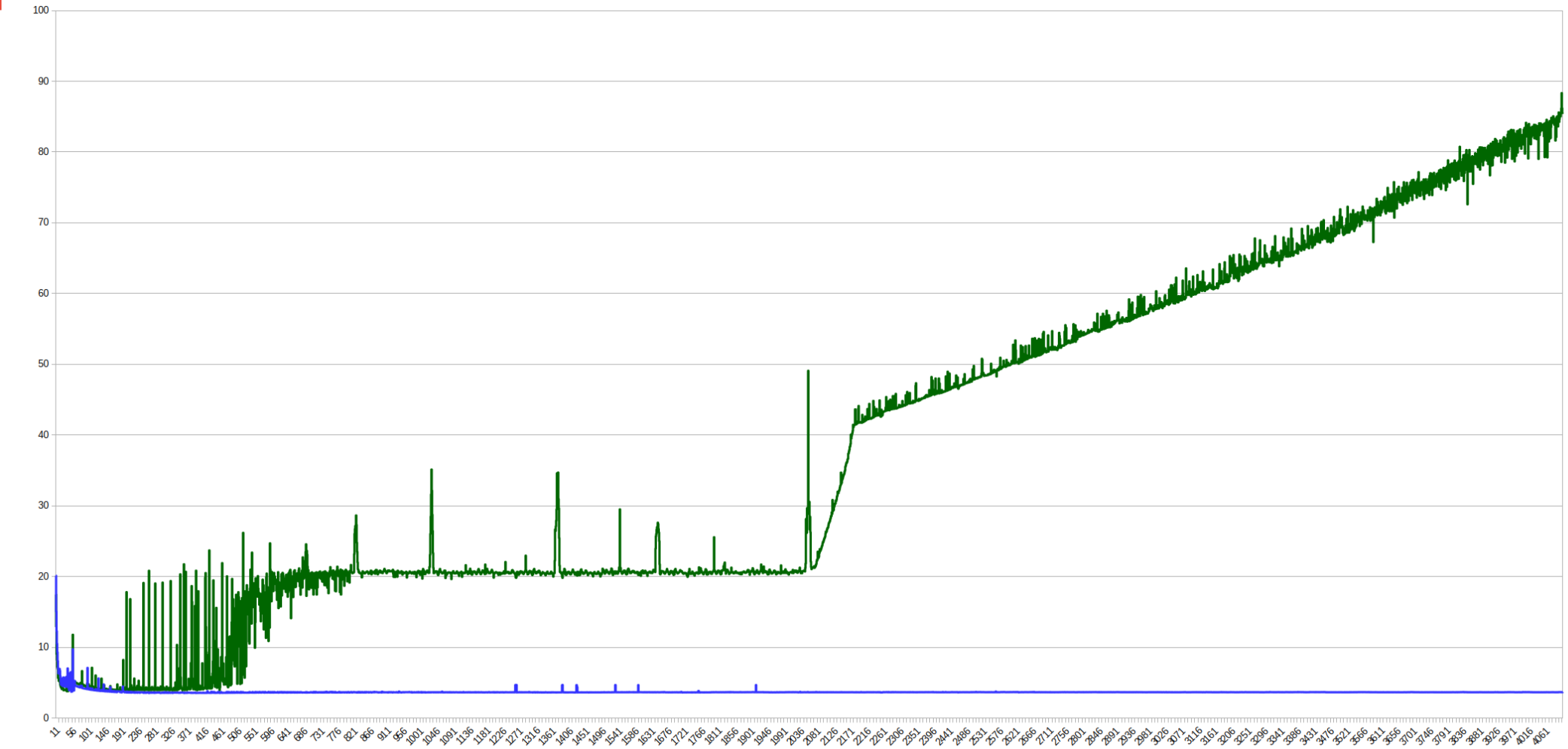
```
char a[N][N];

for (size_t i = 0; i != N; ++i)
   for (size_t j = 0; j != N; ++j)
      a[j][i] = 0;
```

Is there any difference between these two pieces of code?

It turned out there is. Let's plot a graph where:

- Y axis — time taken divided by N^2
- X axis — N

Example #1



X axis — N

Y axis — time taken divided by N^2

Example #1

Changing iteration order causes 40x execution time difference!

What is the cause?
No, it is not because compiler used special instructions:

```
.L5:
  test r9, r9
  je .L3
  mov rdx, QWORD PTR [rdi+16]
  lea rax, [rdx+rcx]
  add rdx, r9
  add rdx, rcx
.L4:
  mov BYTE PTR [rax], 0
  add rax, 1
  cmp rdx, rax
  jne .L4
.L3:
  add rsi, 1
  add rcx, r8
  cmp rsi, r8
  jne .L5
```

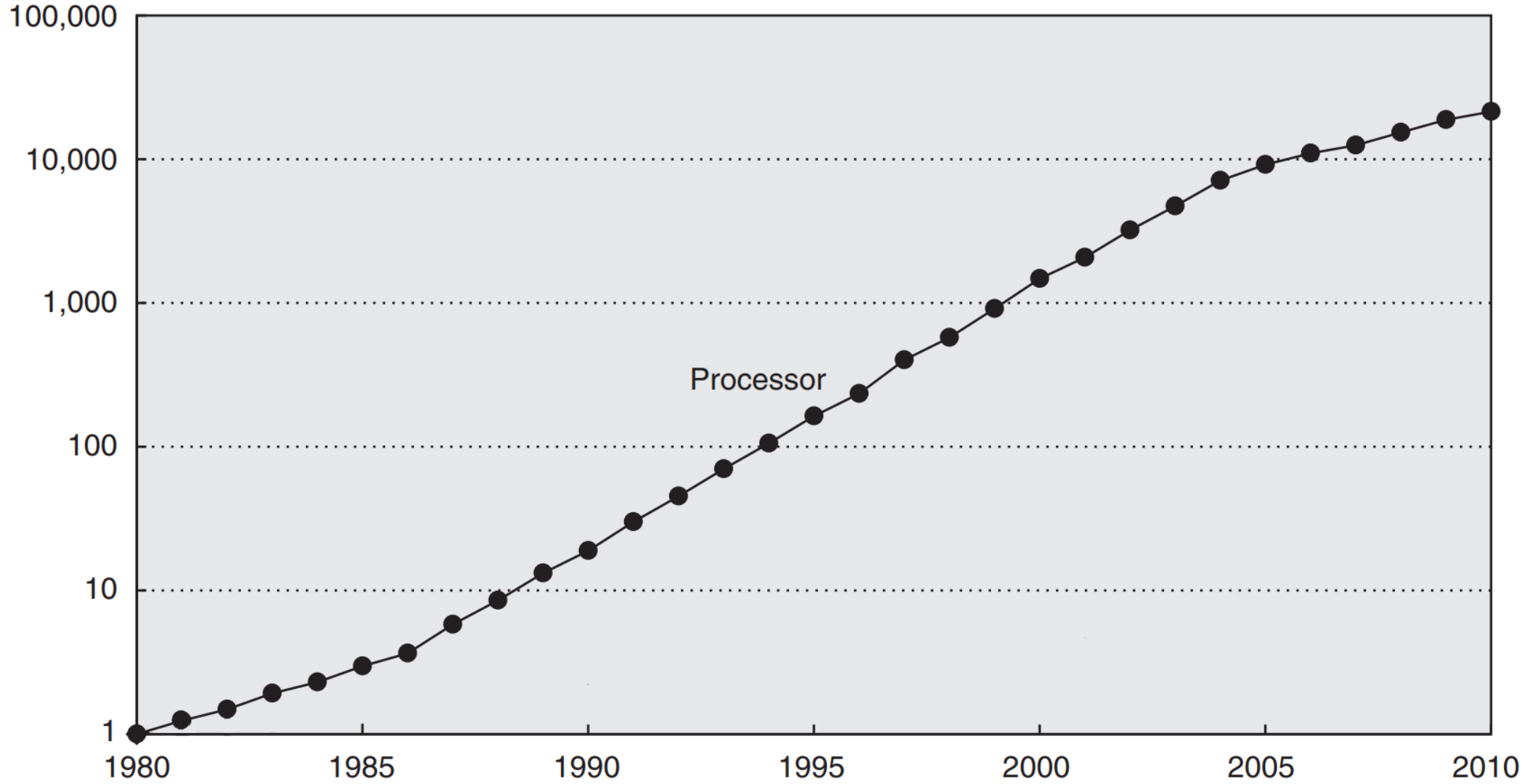
```
.L5:
  test rsi, rsi
  je .L3
  mov rdx, QWORD PTR [rdi+16]
  xor eax, eax
  add rdx, r8
.L4:
  add rax, 1
  mov BYTE PTR [rdx], 0
  add rdx, rcx
  cmp rax, rsi
  jne .L4
.L3:
  add r8, 1
  cmp r8, rcx
  jne .L5
```

Example #1

Changing iteration order causes 40x execution time difference.

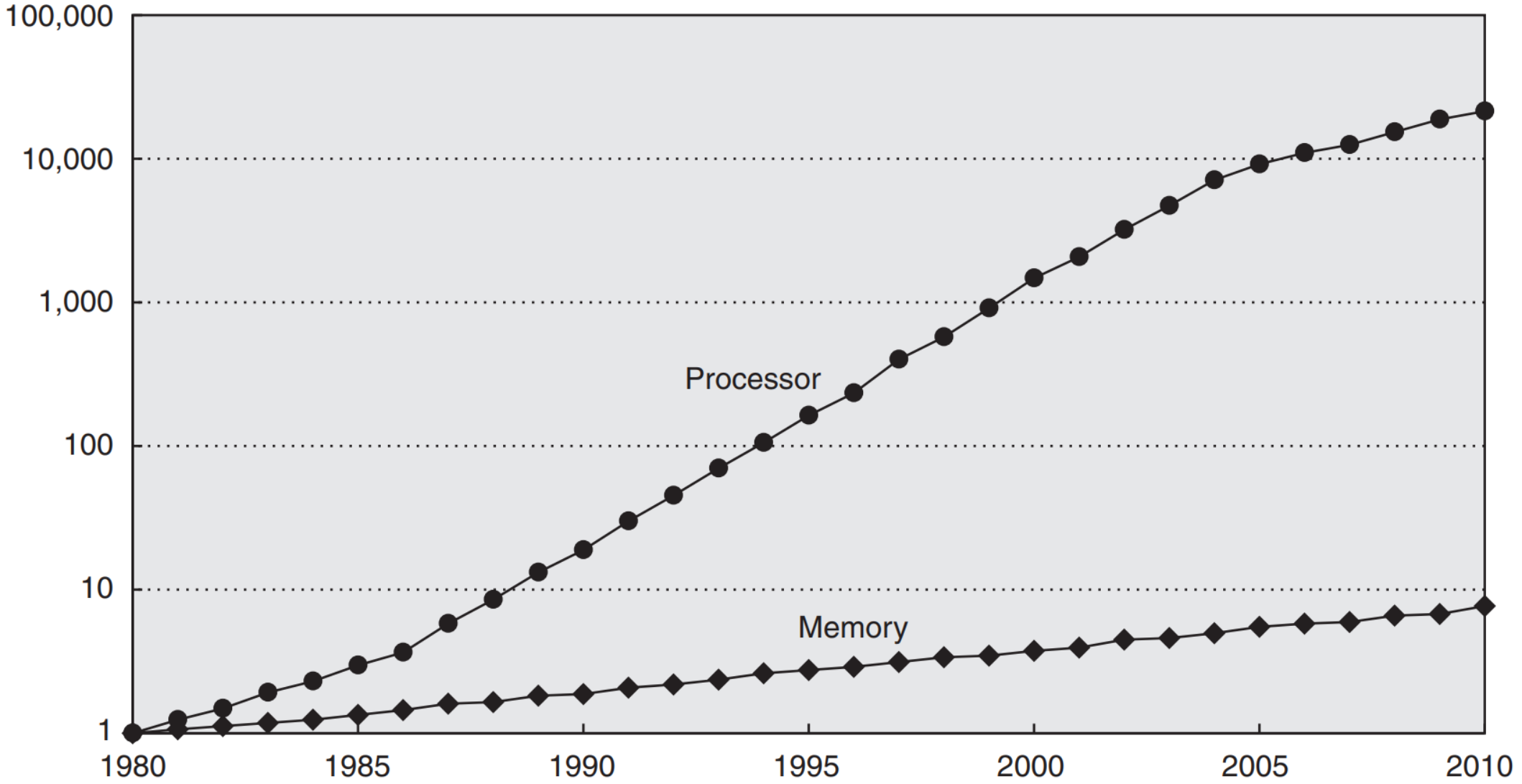
What is the cause?
Processor cache.

CPU Performance over Time



Source: J. Hennessy, D. Patterson — Computer Architecture: A Quantitative Approach
Idea taken from: T. Albrecht — Pitfalls of Object Oriented Programming

CPU and Memory Performance over Time



Source: J. Hennessy, D. Patterson — Computer Architecture: A Quantitative Approach
Idea taken from: T. Albrecht — Pitfalls of Object Oriented Programming

Memory Access Time Comparison

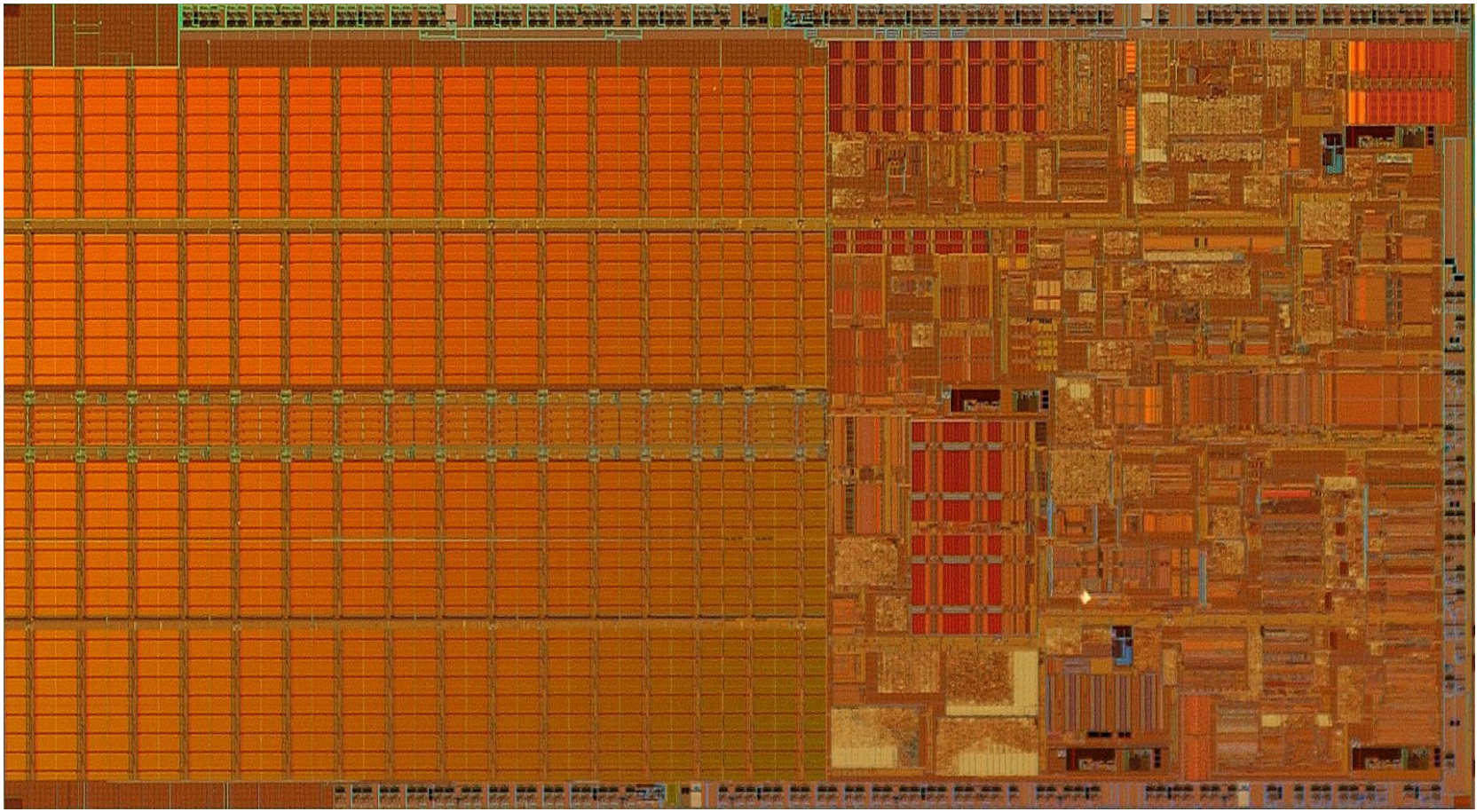
For modern Intel CPUs 4 instructions per cycle execution rate is achievable and sustainable.

Memory access normally costs 200+ cycles.

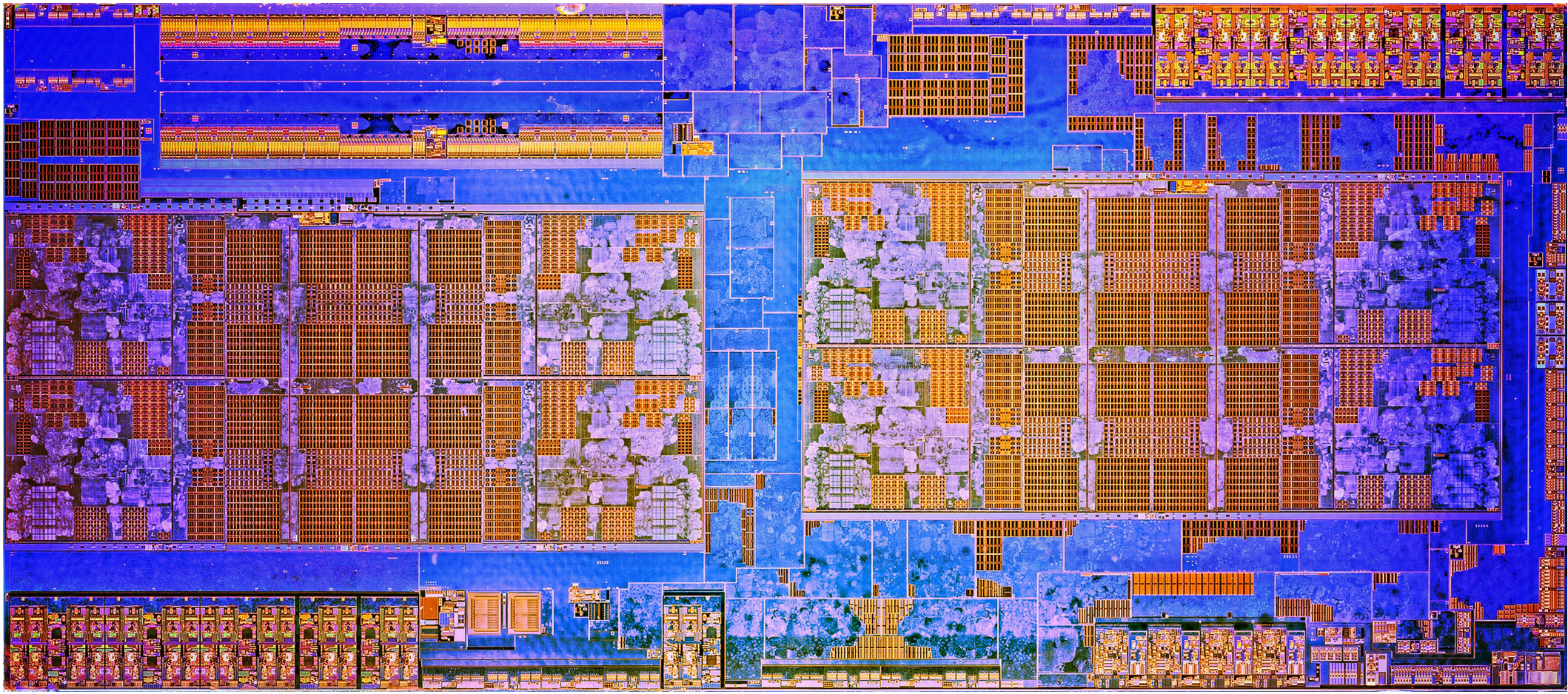
Based on http://www.7-cpu.com/cpu/Haswell.html:

For i7-4770 (Haswell), 3.4 GHz: 230 cycles.
For 3.6 GHz E5-2699 v3 dual: 422 cycles.

Pentium M Die



Ryzen Die



Cache Access Time

For modern Intel CPUs 4 instructions per cycle execution rate is achievable and sustainable.

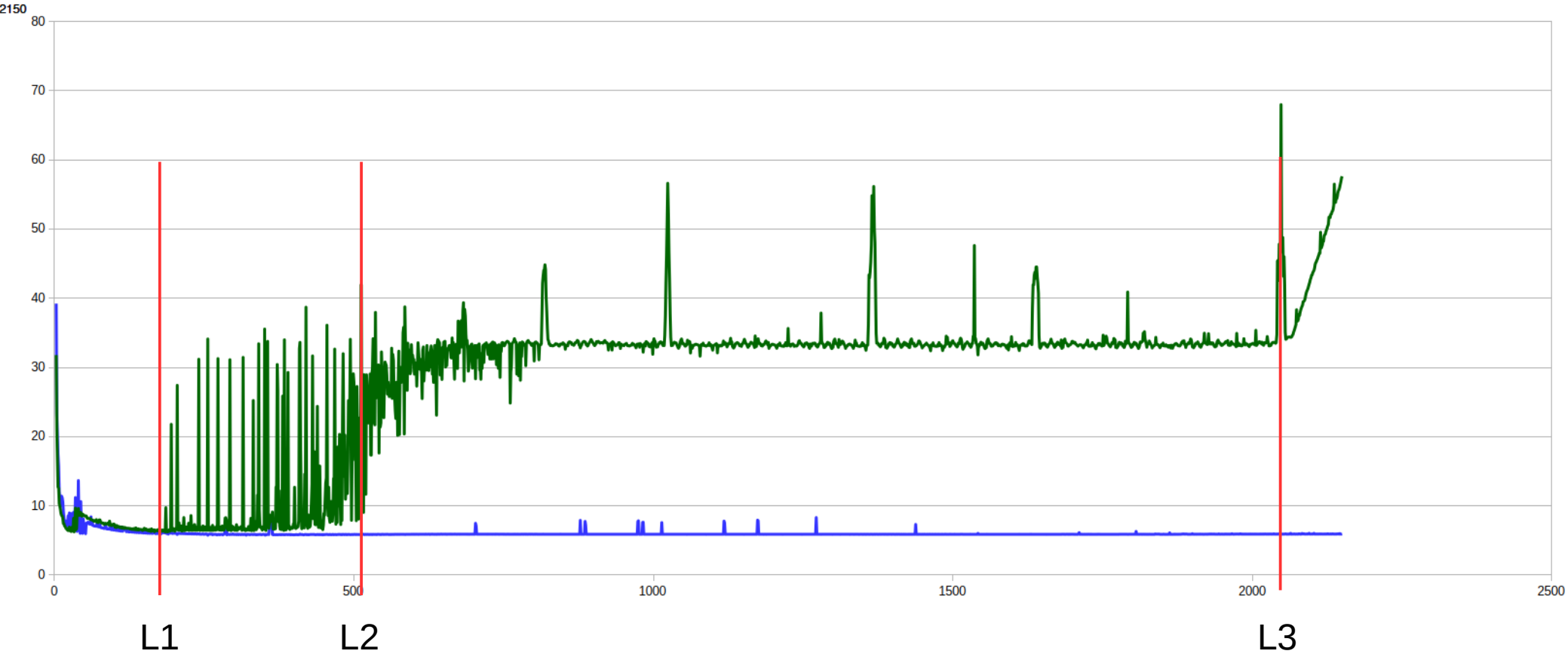
Based on http://www.7-cpu.com/cpu/Haswell.html:

For i7-4770 (Haswell), 3.4 GHz:

L1 access time: 4 cycles.
L2 access time: 12 cycles.
L3 access time: 36 cycles.
RAM access time: 230 cycles.

„Optimize for data first, then code. Memory access is probably going to be your biggest bottleneck“ — T. Albrecht

Example #1



Powers of Two

For some reason powers of two are especially bad:

```
...
255         6.46553       5.87923
256         34.1074       5.74615
257         6.47127       5.87773
...
511         8.17188       5.83852
512         41.9089       5.86711
513         26.1207       5.86645
...
2047        46.632        5.90516
2048        67.9766       5.91165
2049        48.3231       5.90636
...
```

CPU Caches

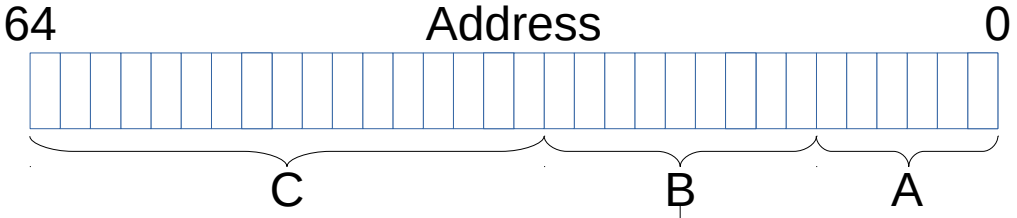
CPU caches are implemented as a hashtable.

The keys are memory addresses.
The values are memory content.

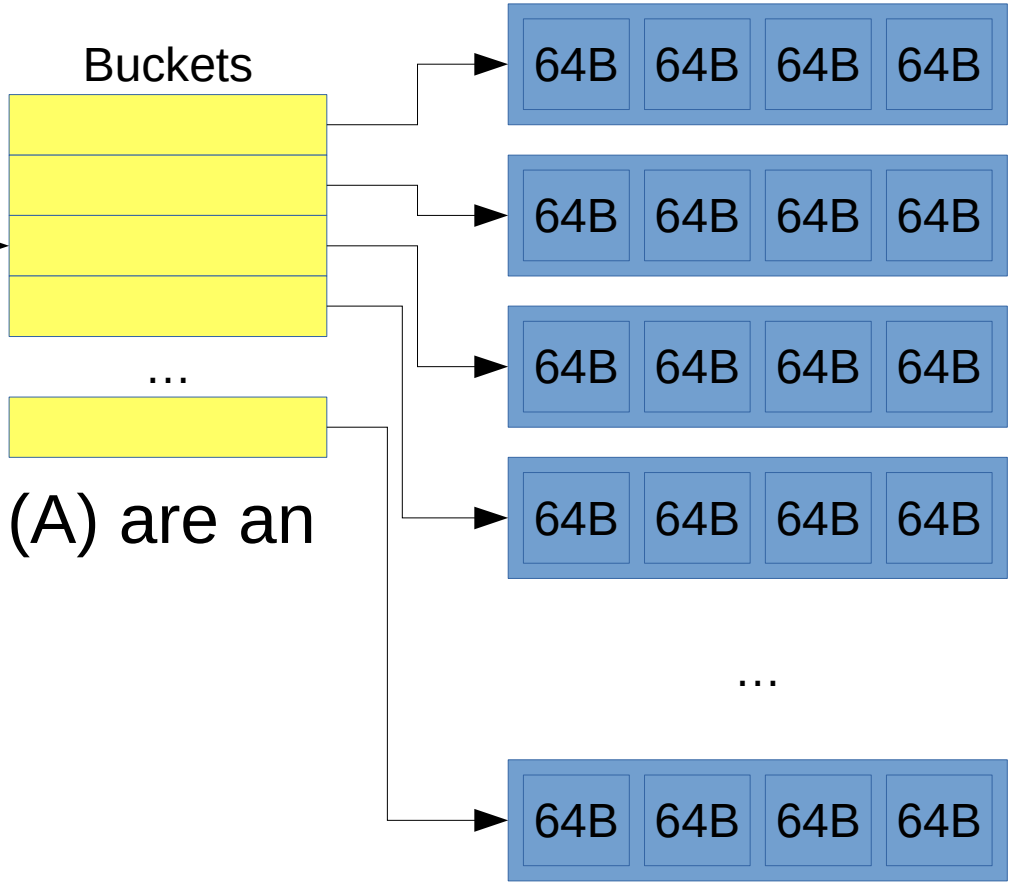
The caches don't work on individual bytes. Accessing one byte causes a whole block (called *cache line*) to be cached.

Typical cache line size is 64 bytes.

CPU Caches



The number of elements in a bucket is called *associativity*.

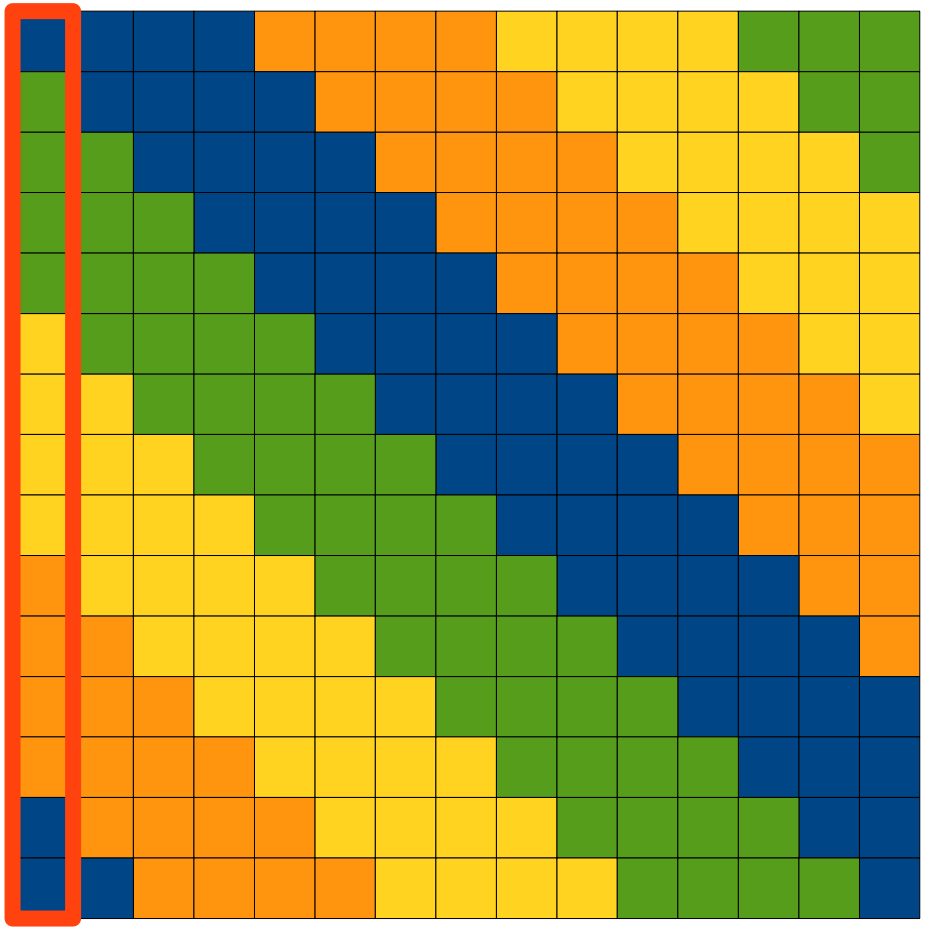


A typical hash function is just a few low bits of an address (B)

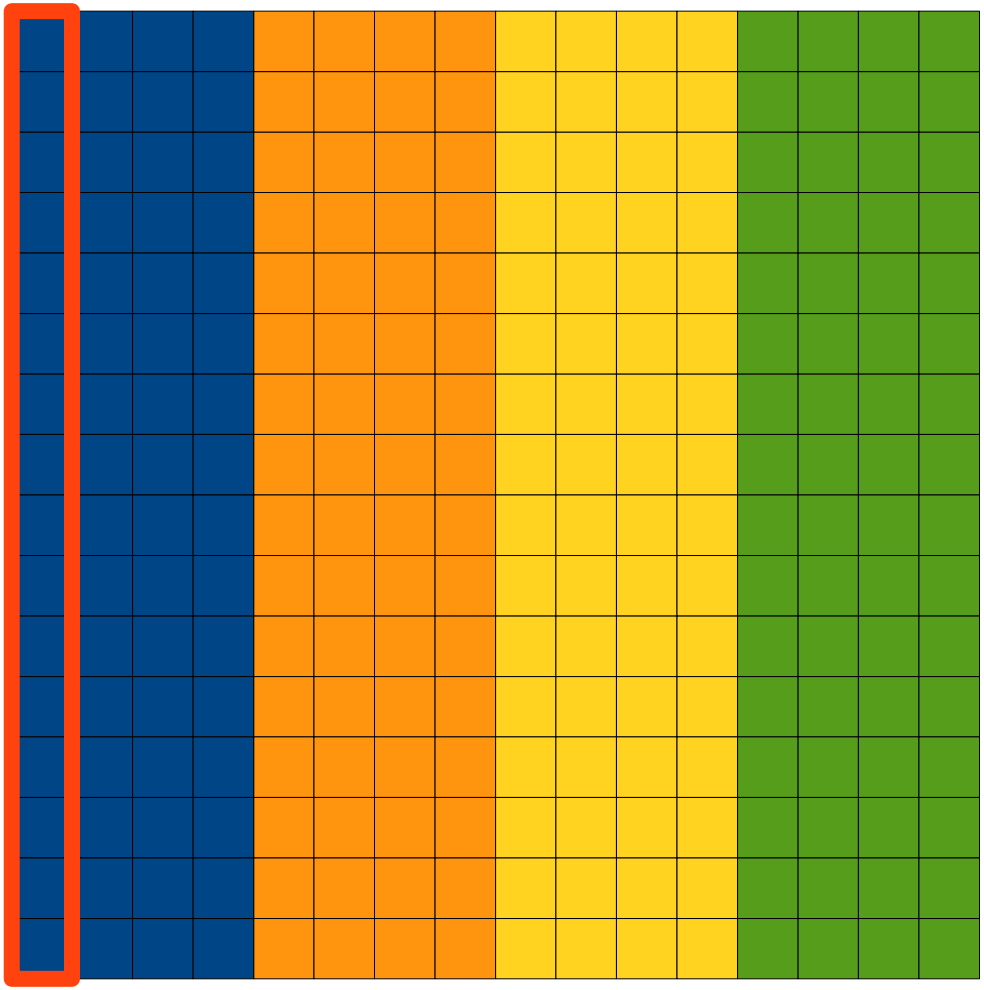
The lowest 6 bits of an address (A) are an offset in 64-byte blocks

Power of Two Matrices

15x15



16x16



Different colors correspond to different buckets.

Cache Hierarchy

Designing a cache subsystem requires finding a balance between:

- Cache line size
- Cache size (bigger caches have higher hit rate, but are slower)
- Associativity

Cache Hierarchy Example

Caches of some common CPUs:

	Haswell i7-4770	Skylake i7-6700	Ryzen 7
L1d size	32 KB	32 KB	32 KB
L1d line size	64 B	64 B	64 B
L1d associativity	8-way	8-way	8-way
L1d access time	4 cycles	4 cycles	4 cycles
L2 size (per core)	256 KB	256 KB	512 KB
L2 line size	64 B	64 B	64 B (?)
L2 associativity	8-way	4-way *	8-way
L2 access time	12 cycles	12 cycles	17 cycles
L3 size (per core)	2 MB	2 MB	2 MB
L3 line size	64 B	64 B	64 B
L3 associativity	16-way	16-way	16-way
L3 access time	36 cycles	42 cycles	39-40 cycles

* https://stackoverflow.com/questions/37957181/skylake-l2-cache-enhanced-by-reducing-associativity

Question

During array traversal only the first item of each cache line causes a stall. Accessing the first item reads whole cache line which make accessing other elements cheap.

Is this right?

Answer

During array traversal only the first item of each cache line causes a stall. Accessing the first item reads whole cache line which make accessing other elements cheap.

Is this right? **Wrong**.

Modern CPUs detect sequential memory access and actively fetch data that is expected to be used next. This technique is called *prefetching*.

Prefetching

„The Intel Pentium 4 can prefetch data into the second-level cache from up to eight streams from eight different 4 KB pages. Prefetching is invoked if there are two successive L2 cache misses to a page, and if the distance between those cache blocks is less than 256 bytes. (The stride limit is 512 bytes on some models of the Pentium 4.) It won't prefetch across a 4 KB page boundary.“

Source: J. Hennessy, D. Patterson — Computer Architecture: A Quantitative Approach

More recent Intel CPUs have 4 different prefetchers. Two prefetch into L1 cache, and two prefetch into L2 and LLC cache.

Source: Intel 64 and IA-32 Architectures Optimization Reference Manual, section 2.3.5.4 Data Prefetching

Prefetching

- Prefetching trades memory bandwidth for reduced latency.
  - Usually there is excessive memory bandwidth to spare and memory bandwidth is easier to scale.

- With prefetching enabled the speed of an algorithm is bounded by either CPU or RAM depending on which is slower.
  - Many simple SIMD-friendly algorithms are memory-bound, not CPU-bound.

Best Practices

- Minimize the size your data
  - The smaller the items are the more of them fits the cache/cache line.
  - The smaller the items are the less memory bandwidth is needed.
- Store data sequentially (if it is accessed sequentially)
  - Sequential access enables prefetching
  - Accessing one element causes others to be cached too
- Keep „cold“ and „hot“ data separately

Example: Hash Tables

Keep „cold“ and „hot“ data separately.

Consider a hash table with *open addressing*. Should we store key/value pair together or separately?

Example: Hash Tables

Keep „cold“ and „hot“ data separately.

Consider a hash table with *open addressing*. Should we store key/value pair together or separately?

It depends on the hit rate.

- If a high hit rate is expected, storing values together with keys can minimize the number of cache misses
- If a low hit rate is expected, storing values separately can conserve cache capacity for keys

Best Practices

- Avoid indirections
  - The location of the dereferenced object is generally unpredictable and causes an extra cache miss
  - The more objects are allocated the more pressure is put on the the memory allocator
  - Extra pointers increase the size of the data structures

- In some cases the elimination of indirections is harmful
  - Avoid „inlining“ cold objects into hot
  - These cases are rare in practice

Pointers

- Pointers are probably the most common objects in any program
- Everything is made of pointers:
  - `std::vector` consists of 3 pointers
  - `std::list` consists of 2 pointers and one integer of pointer size
  - `std::set`, `std::map` consist of 1 pointer and one integer of pointer size
  - `size_t`, `ptrdiff_t`
- Most programs don't use more that 4GB of RAM. 64-bit pointers only waste extra space.
- Can something be done about it?

x86 vs x86-64

- x86 has 32-bit pointers, but using x86 has its own downsides:
  - Limited number of registers
  - Baroque calling conventions
  - Expensive PIC

x32 ABI

- With x32 ABI the program runs in 64-bit CPU mode using 32-bit pointers.

	x86	x32	x86-64
Pointers	4 bytes		8 bytes
Virtual memory	4 GB		128 TB
Integer registers	6 (PIC)	15	
FP registers	8	16	
64-bit arithmetic	No	Yes	
FP arithmetic	x87	SSE	
Calling convention	Stack	Registers	
PIC prologue	2-3 instructions	None	

Source: https://en.wikipedia.org/wiki/X32_ABI

x32 ABI

- x32 ABI is reported to be:
  - 5-8% faster than x86-64 on SPEC CPU INT
  - No difference with x86-64 on SPEC CPU FP
  - 40% faster than x86-64 on 181.mcf (memory bound)

  - 7-10% faster than x86 on SPEC CPU INT
  - 5-11% faster than x86 on SPEC CPU FP
  - 40% faster than x86 on 186.crafty (64-bit arithmetic)

Source: H. J. Lu, H. Peter Anvin, M. Girkar — X32 – A Native 32bit ABI For X86-64
http://www.linuxplumbersconf.net/2011/ocw//system/presentations/531/original/x32-LPC-2011-0906.pptx

https://sites.google.com/site/x32abi/

Best Practices

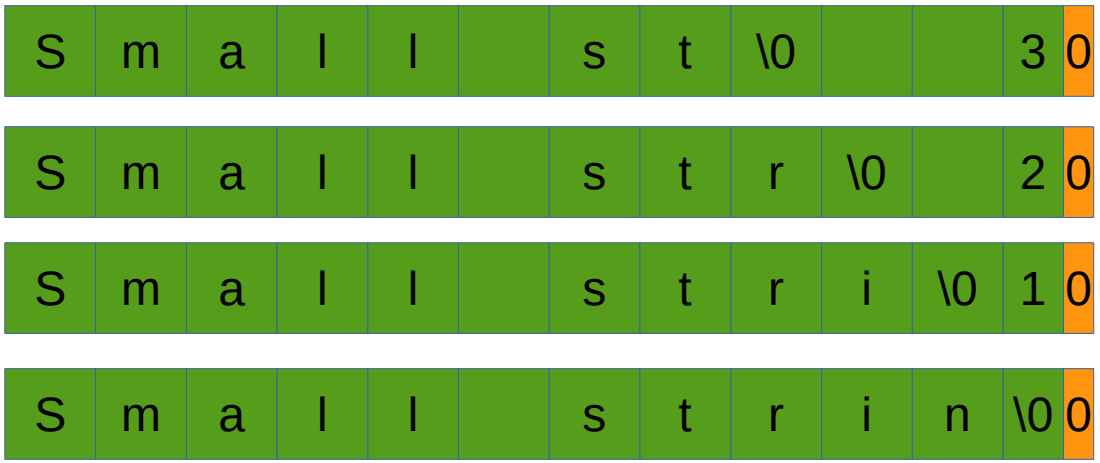
- Optimize for common case
- Consider using small-object-optimized containers
  - Improves data locality for common case (small number of items)
  - Reduces the number of allocations
- Be aware of the downsides of small-object-optimized containers
  - Both overcounting and undercounting the expected number of items waste some memory
  - Using small-object-optimized containers usually (but not always) increases the size of the enclosing objects
  - Usually small-object-optimized containers are slower to access in terms of CPU operations

fbstring

Small-object optimization doesn't necessarily increase the size of the object.

Big object:

Small object:



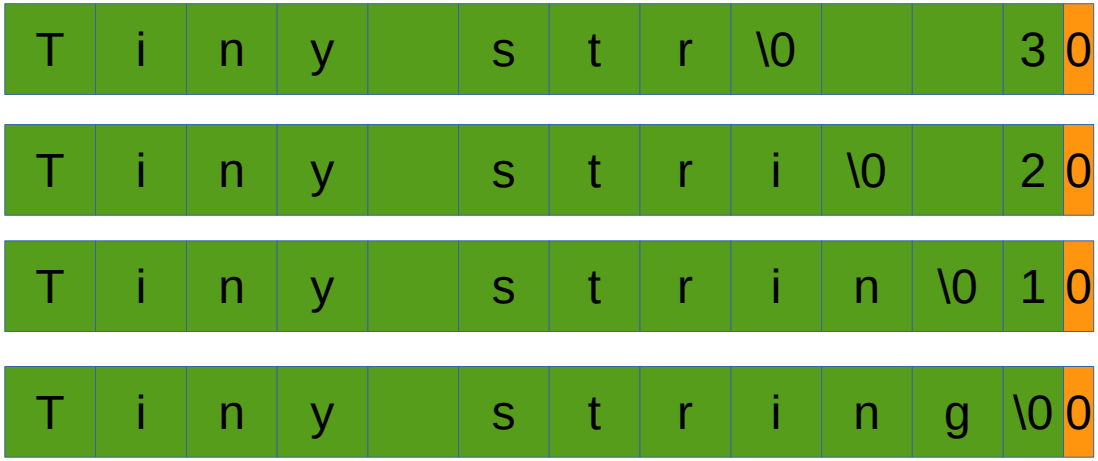
fbstring

The last byte serves triple purpose:

- Its highest bit is the selector whether the string is big
- Holds the number of free characters in the small case
- Serves as null-terminator if the string is small and full

Big object:

Small object:



fbstring

On 64-bit system fbstring can store up to 23 chars in-place.

For big endian systems the layout is different, but the idea is the same.

The real fbstring steals two highest bits of the capacity and distinguishes between three case:

- Small string — in-place storage (<24 bytes)
- Medium string — dynamic storage (24 to 256 bytes)
- Large string — dynamic storage, copy-on-write (>256 bytes)

Best Practices

- Consider using small-object-optimized polymorphic wrappers and discriminated unions
  - Have the same up-/downsides as small-object optimization
- Not all alternatives have to be stored in-place
  - Rare and big alternatives can have dynamic storage
  - The library Dyno by Louis Dionne allows storage type customization
- In some cases writing your own polymorphic wrappers is worth the trouble
  - Don't forget hiding the bit-trickery behind a type-safe interface

Example

As an example let's design a class to store C++ unqualified names.

There are multiple types of unqualified names:

• Identifier	`foo`	>99%
• Anonymous		<0.1%
• Destructor	`~foo`	<0.1%
• Operator	`operator+`	<0.1%
• Coversion	operator int	<0.1%
• ...		

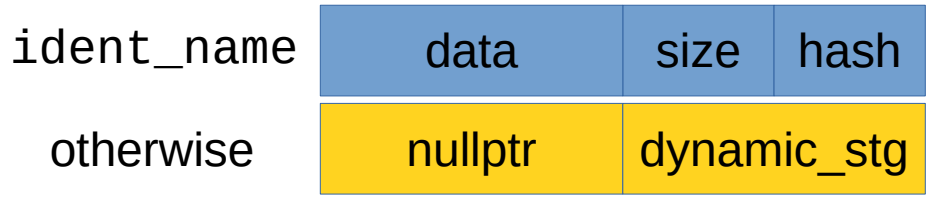
For simplicity let's omit template names: `foo<int>`.

Example

```
struct ident_name
{
   char const* data; // can not be nullptr
   uint32_t size;
   uint32_t hash;
};

struct destructor_name { /* same as ident_name */ };
struct anonymous_name {};
struct operator_name { enum operator_kind kind; };
struct conversion_name { qual_type type; };
```

Example

Polymorphic wrapper should be optimized for the most common case




```
struct name
{
    // non-nullptr for ident_name
    // nullptr otherwise
    char const* data;
    union
    {
        // ident_name's case
        struct { uint32_t size; uint32_t hash; };

        // anything else's case
        dynamic_name_storage* dynamic_storage;
    };
};
```

Virtual Memory

The address that a program uses to access the memory is not the same address that is used to access physical memory.

In this presentation let's call the address that the program uses a *virtual* address. (*)

A CPU provides and an OS uses a mechanism to specify the mapping from virtual to physical addresses.

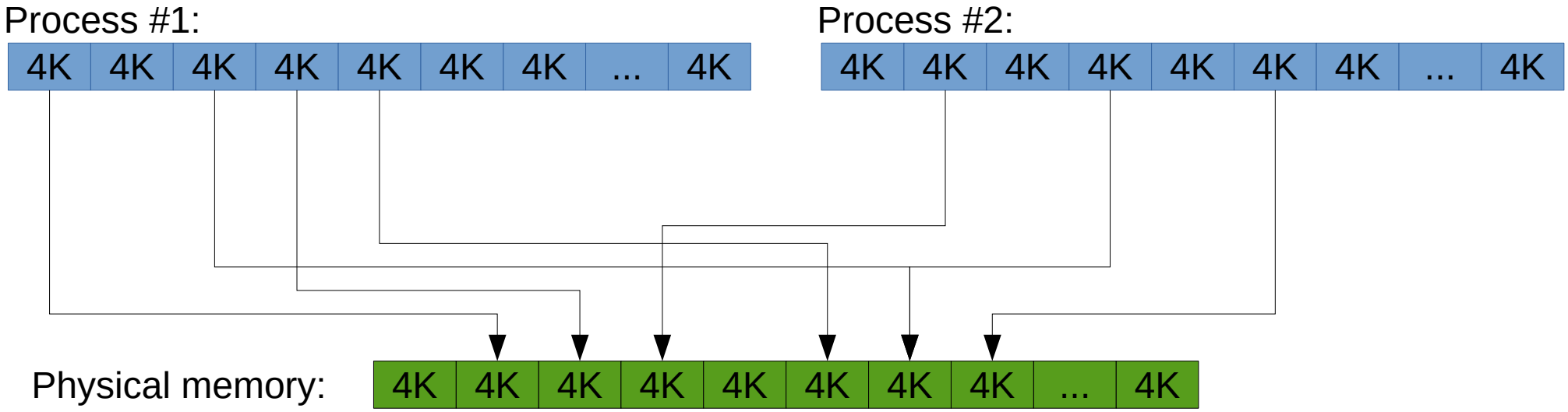
* This is technically incorrect and the address should be called a linear address. But for the sake of simplicity segmentation is ignored in this presentation.

Virtual Memory

The process of mapping virtual addresses to physical is called *paging*.

Paging involves grouping all virtual addresses and all physical addresses into groups called *pages*. The size of a page is 4 KB and pages are aligned to 4 KB.

Virtual Memory



CPU allows mapping any page of *virtual address space* to any page of *physical address space*.

Different pages of *virtual address space* can be mapped to the same page of *physical address space*.

Normally OS configures different mappings for different processes.

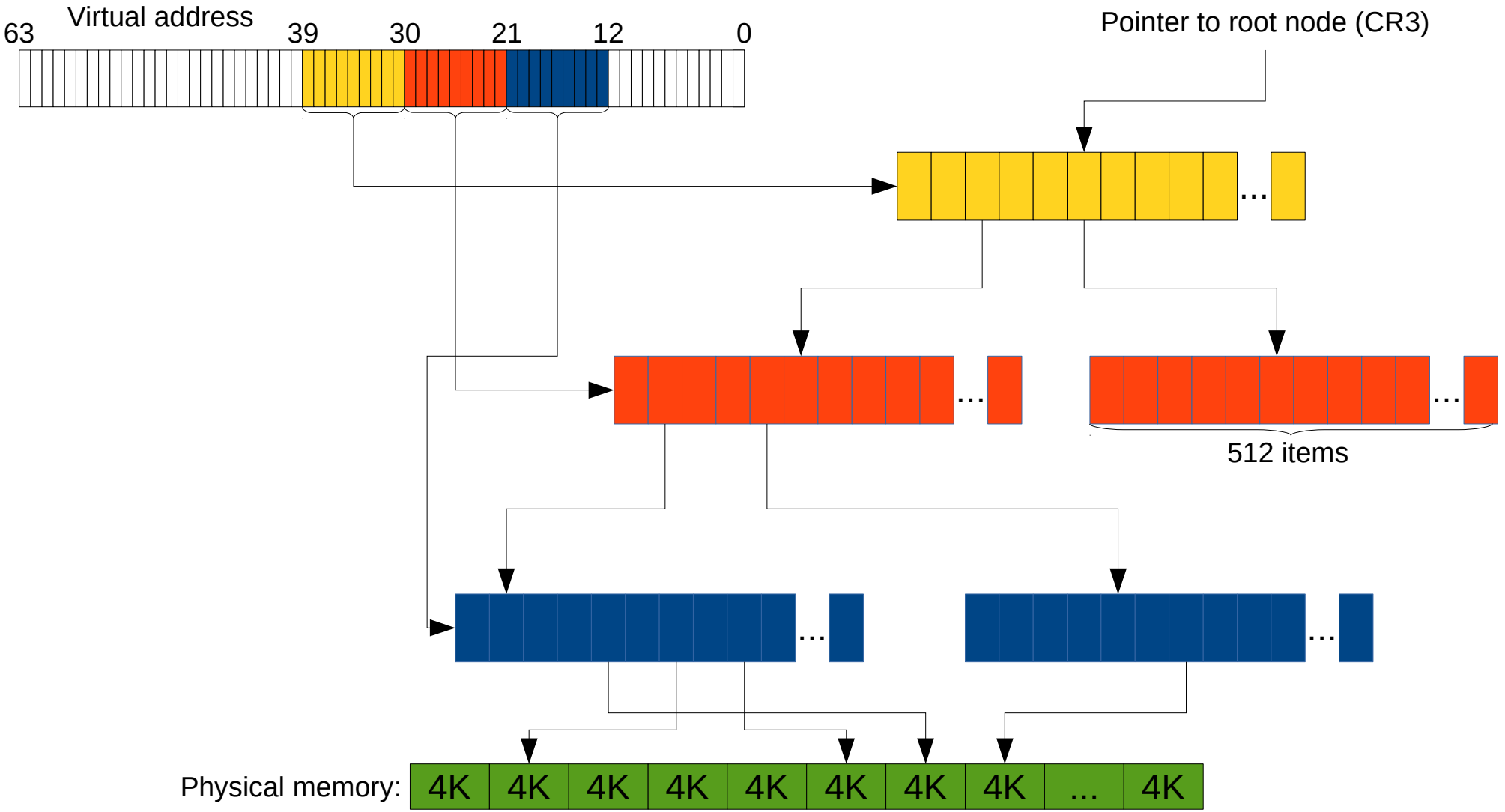
Page Tables

The mapping which page of virtual address space translates to which page of physical address space is stored in 512-ary tree of bounded depth (usually 3, but 2, 4 or 5 is possible).

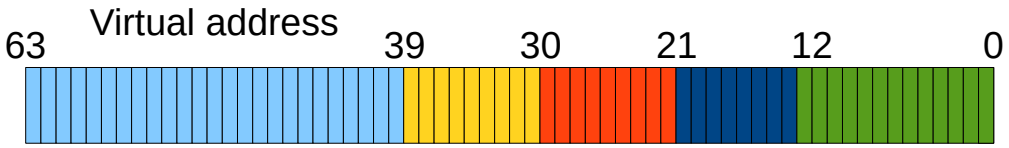
Each level of this tree has its own name, but we'll call this tree collectively *page tables*.

Each child of a node is indexed from 0 to 511.

Page Tables


Virtual address
63
39
30
21
12
0
Pointer to root node (CR3)
512 items
Physical memory:
4K
4K
4K
4K
4K
4K
4K
4K
...
4K

Page Tables



The lowest 12 bits of address are an offset inside a page. They are the same for physical and virtual addresses.

The highest 25 bits are unused and must be the same as the 38th bit of the address (an address must be sign-extended on x86-64).

Besides the address of the child node each item holds flags (whether the child is present, is the page writable, and etc).

OS keeps separate page tables for each process. During process switching OS updates a pointer to the root (called CR3 register).

Page Tables

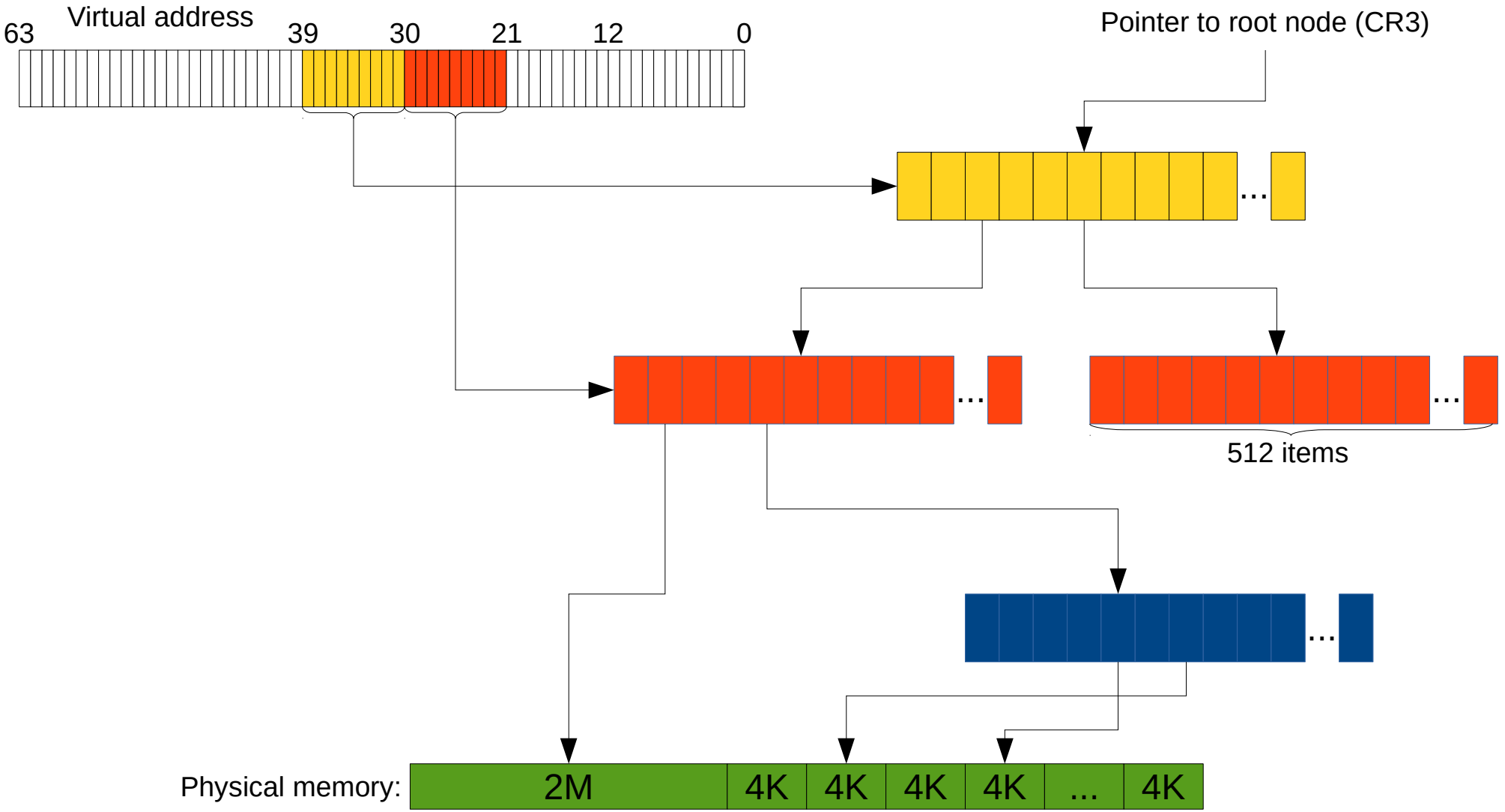
With paging enabled each memory access is turned into 4(!) memory accesses. Obviously a 4-fold increase in number of memory accesses is impractical.

CPUs have a special cache for virtual to physical translations, called *TLB (translation lookaside buffer)*.

On each memory access CPU queries both TLB and caches.

As other caches TLB has limited size and associativity.

Huge Pages


Virtual address
63
39
30
21
12
0
Pointer to root node (CR3)
512 items
Physical memory:
2M
4K
4K
4K
4K
...
4K

Huge Pages

Large pages improve performance in three ways:

- They make TLB misses cheaper by decreasing the number of accessed nodes
- They consume fewer resources of TLB
- The OS kernel spends less time in handling page faults

Different sources report different speedups ranging from 6% to 22%. (*)

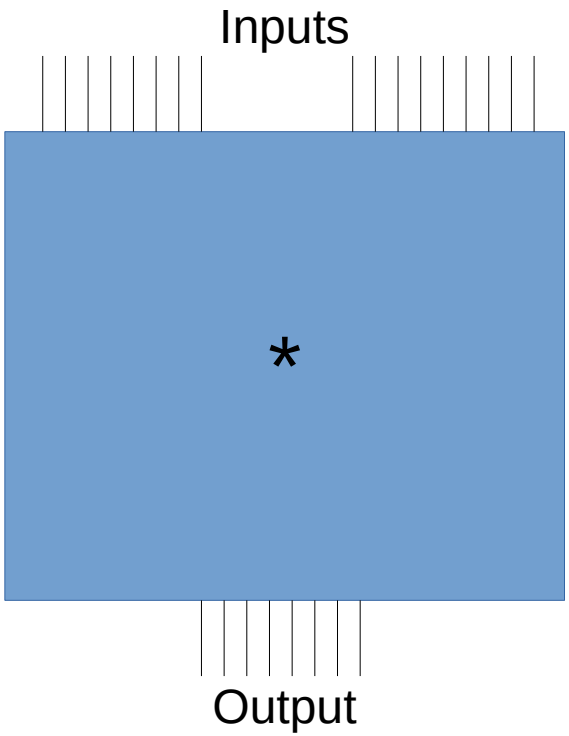
* Andrea Arcangeli — Transparent Hugepage Support
https://events.static.linuxfound.org/slides/2011/lfcs/lfcs2011_hpc_arcangeli.pdf
Tobiasrieper — Improve your CPU Monero mining...
https://steemit.com/monero/@tobiasrieper/improve-your-cpu-monero-xmr-mining-up-to-20-with-huge-pages

TLBs

	Haswell i7-4770	Skylake i7-6700	Ryzen 7
4K pages, L1	64 items, 4-way	64 items, 4-way	64 items, full assoc.
4K pages, L2	1024 items, 8-way	1536 items, 12-way	1536 items, 8-way
2M pages, L1	32 items, 4-way	32 items, 4-way	64 items, full assoc.
2M pages, L2	1024 items, 8-way	1536 items, 12-way	1536 items, 2-way (?)
1G pages, L1	4 items	4 items, 4-way	64 items, full assoc.
1G pages, L2		16 items, 4-way	

Pipelining

Assume that we have some boolean circuit.

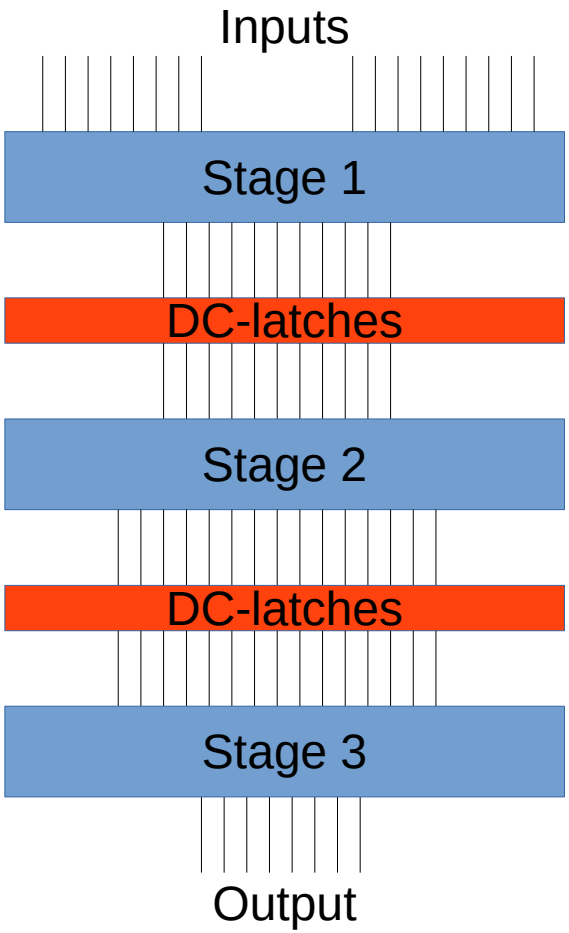


Pipelining

Assume that we have some boolean circuit.

Sometimes the circuit can be split into several stages connected by DC-latches.

The clock rate is determined by the stage with the slowest signal propagation delay. If the circuit can be split into 3 stages with similar propagation delay the clock the clock rate can be increased 3 times.



Pipelining

Sequential Execution:

Pipelined Execution:

With pipelining latency is increased only slightly while the throughput is increased many times.

Pipelining

Although modern CPUs are not pure pipelines, they share many properties with pipelines.

Many instructions are in fly simultaneously at different stages of execution. To some extent modern CPUs can be modelled as pipelines of ~20 stages deep.

The problem is the instructions called „*conditional branches*“. The conditional branch instruction is an instruction that selects depending on some condition what is executed after it.

Usually it is not known if the condition is true until the late stage of the execution of the condition branch.

Pipelining

Although modern CPUs are not pure pipelines, they share many properties with pipelines.

Many instructions are in fly simultaneously at different stages of execution. To some extent modern CPUs can be modelled as pipelines of ~20 stages deep.

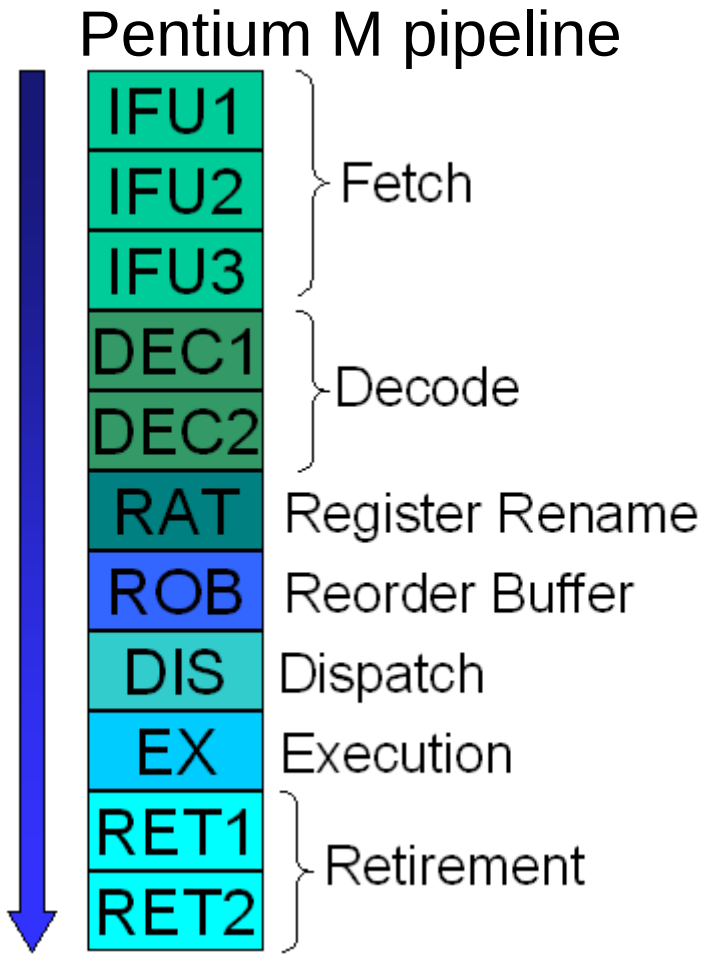
Example: Pentium M Pipeline

Although modern CPUs are not pure pipelines, to some extent they can be modelled as pipelines.

According to this definition Pentium M had 11 stages.

Pentium 4 (Willamette, Northwood) — 20.
Pentium 4 (Prescott, Cedar Mill) — 31.

Haswell — 14-19 stages depending on how you count.



Conditional Branches

The problem is the instructions called „*conditional jumps*“. The conditional branch instruction is an instruction that selects which instruction is executed next after it depending on some condition.

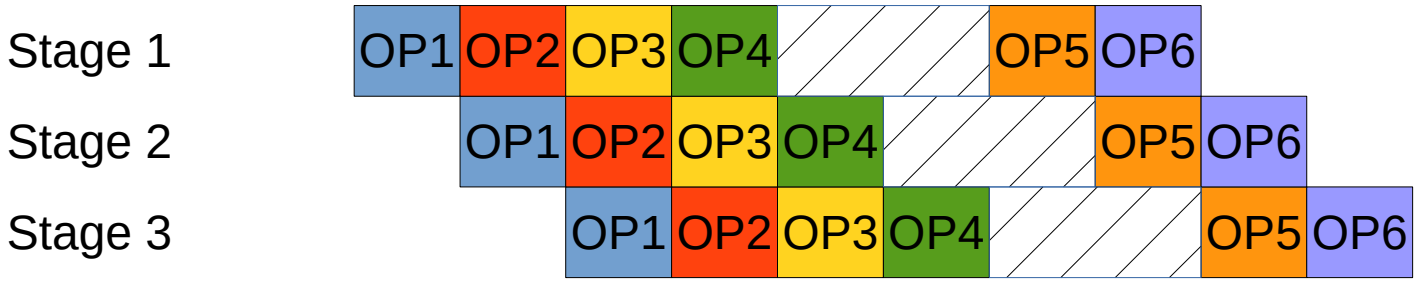
Usually it is not known if the condition is true until the late stage of execution of the condition jump.

Conditional jumps have two options depending on which instruction is executed after next:

- The next instruction in memory order is executed (in this case the jump is called *not taken*)
- The execution starts with the instruction specified in the conditional jump (in this case the jump is called *taken*)

Conditional Branches

Pipelined Execution:



Just waiting for a conditional branch to complete causes a pipeline stall for as many cycles as the pipeline depth is.

What if instead of stalling CPU some branch is executed and then after conditional jump instruction is completed either:

- The instructions in wrong branch were executed, the results of the computation need to be discarded, and the execution need to be restarted
- The instructions in the right branch were executed, no special action is needed.

Branch Prediction

A branch predictor is the part of a CPU that predicts if the specific branch should be taken or not.

If branch is predicted correctly no (or very little) time is wasted. If branch is mispredicted the number of cycles corresponding to the depth of pipeline is wasted.

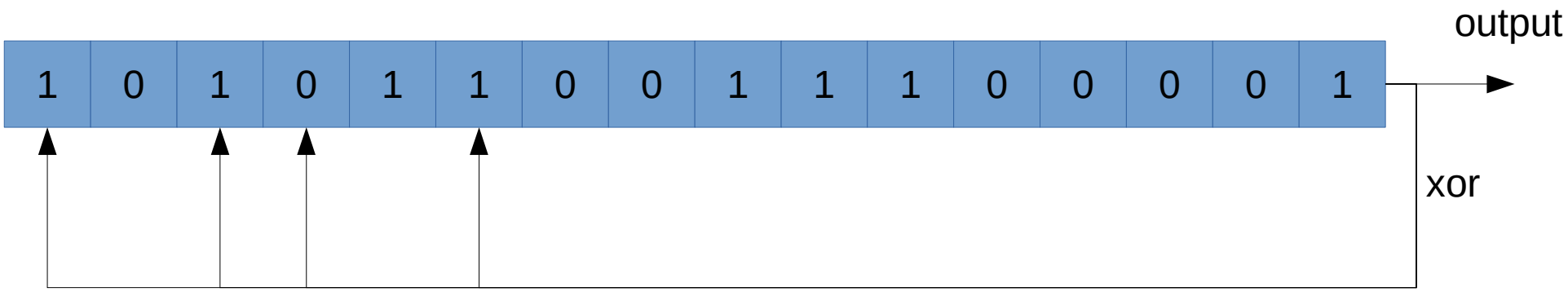
Both Intel and AMD publish little information on how exactly their branch predictors work.

The branch predictors of modern CPUs work in a such way that no simple short sequence of branches exhibit bad behavior.

Branch Prediction

To circumvent branch predictor I'll base the branching behavior of a program on a random number generator.

Linear Feedback Shift Register:



Branch Prediction

To circumvent branch predictor I'll base the branching behavior of a program on a random number generator.

Linear Feedback Shift Register:

```
unsigned lfsr = 0xace1;

for (unsigned i = 0; i != 1000000000; ++i)
{
   unsigned lsb = lfsr & 1;  /* Get LSB (i.e., the output bit). */
   lfsr >>= 1;               /* Shift register */
   if (lsb)                  /* This branch is expected to be */
      lfsr ^= 0xb400;        /* often mispredicted */
}
```

Branch Prediction

Let's compare the version with a branch with two branchless versions.

```
unsigned lsb = lfsr & 1;
lfsr >>= 1;
if (lsb)
    lfsr ^= 0xb400;
```

```
unsigned lsb = lfsr & 1;
lfsr >>= 1;
lfsr ^= 0xb400 * lsb;
```

```
unsigned lsb = lfsr & 1;
lfsr >>= 1;
lfsr ^= 0xB400 & -lsb;
```

```
$ chrt -f 99 perf stat ./a.out
```

1.97 seconds	1.64 seconds	1.32 seconds

Branches are 20% slower than multiplication and 50% slower than bitwise and and negation. Anyone disagrees with the result?

Branch Prediction

Wait, only 1 billion of branches?

```
$ sudo chrt -f 99 perf stat ./demo-branch-o2

 Performance counter stats for './demo-branch-o2':

      1977,494719      task-clock (msec)         #    0,993 CPUs utilized
               18      context-switches          #    0,009 K/sec
                0      cpu-migrations            #    0,000 K/sec
              115      page-faults               #    0,058 K/sec
    6 056 788 736      cycles                    #    3,063 GHz
    9 012 566 316      instructions              #    1,49  insn per cycle
    1 002 097 766      branches                  #  506,751 M/sec
           58 221      branch-misses             #    0,01% of all branches

      1,990458396 seconds time elapsed
```

Branch Prediction

```
$ sudo chrt -f 99 perf stat ./demo-branch-o2
       1977,494719      task-clock (msec)         #    0,993 CPUs utilized
     6 056 788 736      cycles                    #    3,063 GHz
     9 012 566 316      instructions              #    1,49  insn per cycle
     1 002 097 766      branches                  #  506,751 M/sec
            58 221      branch-misses             #    0,01% of all branches

$ sudo chrt -f 99 perf stat ./demo-mul-o2
       1638,604757      task-clock (msec)         #    0,997 CPUs utilized
     5 031 149 137      cycles                    #    3,070 GHz
     7 010 510 218      instructions              #    1,39  insn per cycle
     1 001 738 182      branches                  #  611,336 M/sec
            31 210      branch-misses             #    0,00% of all branches

$ sudo chrt -f 99 perf stat ./demo-negand-o2
       1320,836315      task-clock (msec)         #    0,997 CPUs utilized
     4 047 430 739      cycles                    #    3,064 GHz
     8 009 734 001      instructions              #    1,98  insn per cycle
     1 001 594 794      branches                  #  758,303 M/sec
            31 070      branch-misses             #    0,00% of all branches
```

Conditional Move

The compiler replaced the if with a conditional move:

```
.L3:
    mov ecx, ebx
    shr ebx
    mov edx, ebx
    and ecx, 1
    xor dh, 180
    test    ecx, ecx
    cmovne  ebx, edx
    sub eax, 1
    jne .L3
```

```
.L2:
    mov edx, ebx
    and ebx, 1
    imul    ebx, ebx, 46080
    shr edx
    xor ebx, edx
    sub eax, 1
    jne .L2
```

```
.L2:
    mov edx, ebx
    and ebx, 1
    neg ebx
    shr edx
    and ebx, 46080
    xor ebx, edx
    sub eax, 1
    jne .L2
```

Conditional Move

GCC has a flag to disable conditional moves:

```
$ g++ -O2 -fno-if-conversion -fno-if-conversion2 demo.cpp

$ sudo chrt -f 99 perf stat ./demo-branch-noifconv

 Performance counter stats for './demo-branch-noifconv':

       3823,392765      task-clock (msec)         #    1,000 CPUs
                 0      context-switches          #    0,000 K/sec
                 0      cpu-migrations            #    0,000 K/sec
               102      page-faults               #    0,027 K/sec
    11 804 910 900      cycles                    #    3,088 GHz
     7 517 019 848      instructions              #    0,64  insn per cycle
     2 002 890 565      branches                  #  523,852 M/sec
       500 409 199      branch-misses             #   24,98% of all branches

       3,823970930 seconds time elapsed
```

Jump Tables

Often writing interpreters and lexers/parsers involves a pattern like this:

```
for (;;)
{
    switch (*c++)
    {
    case OP_1: /* ... */ break;
    case OP_2: /* ... */ break;
    case OP_3: /* ... */ break;
    case OP_4: /* ... */ break;
    }
}
```

It turned out this is not the most efficient way to execute interpreter on modern CPUs.

Jump Tables

Often the input is not a random sequence of items. Some items are more likely to be followed by others.

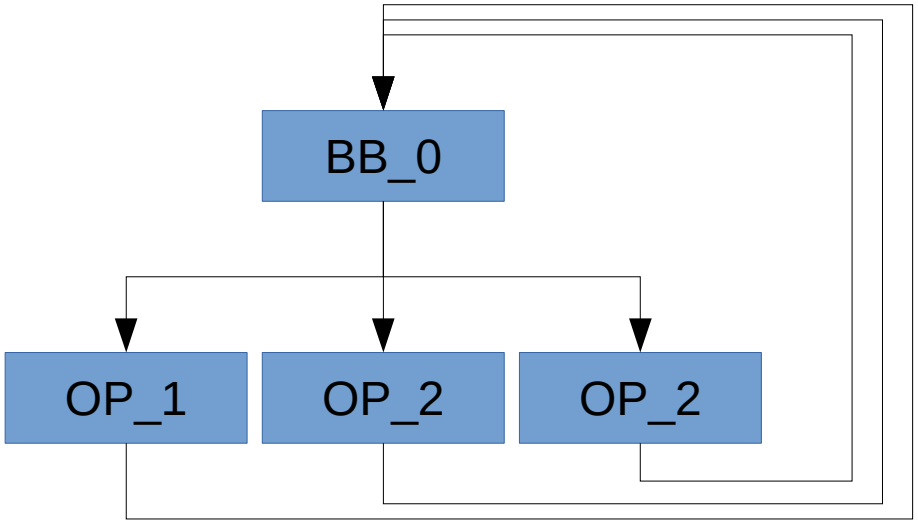
```
static void* const dispatch_table[] =
    {&&do_1, &&do_2, &&do_3, &&do_4};
#define dispatch() goto *dispatch_table[*c++];

do_1:
    /* ... */
    dispatch();
do_2:
    /* ... */
    dispatch();
do_3:
    /* ... */
    dispatch();
do_4:
    /* ... */
    dispatch();
```

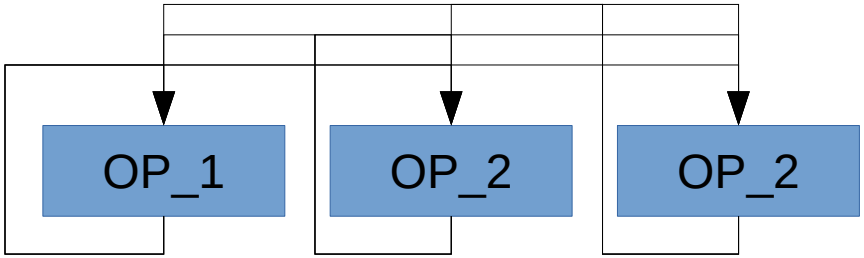
http://www.emulators.com/docs/nx25_nostradamus.htm
E. Bendersky — Computed goto for efficient dispatch tables
https://eli.thegreenplace.net/2012/07/12/computed-goto-for-efficient-dispatch-tables

Jump Tables

Jump tables helps in two ways

- They reduce the number of jumps
- They helps the branch predictor to detect patterns in the input data





Best Practices

Use PGO

- Unless you write an interpreter (or lexer), let the compiler figure out what to do with branches

Some projects use `__builtin_expect`:

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

There is some controversy around it, but it has some value in case PGO is not available.

There is a proposed C++20 attribute `[[likely]]`.

Clang provides `__builtin_unpredictable`, to mark the branch conditions that can not be predicted by hardware logic.

Example

Consider the following code:

```
void count_huffman_weights(char const* src, size_t size)
{
    uint32_t count[256] = {};

    for (size_t i = 0; i != size; ++i)
        ++count[src[i]];
}
```

Source: Counting bytes fast — little trick from FSE
http://fastcompression.blogspot.ru/2014/09/counting-bytes-fast-little-trick-from.html

Example

Consider the following code:

```
void count_huffman_weights(char const* src, size_t size)
{
    uint32_t count[256] = {};

    for (size_t i = 0; i != size; ++i)
        ++count[src[i]];
}
```

It turned out that the runtime of this function is highly sensitive to the input values.

Example

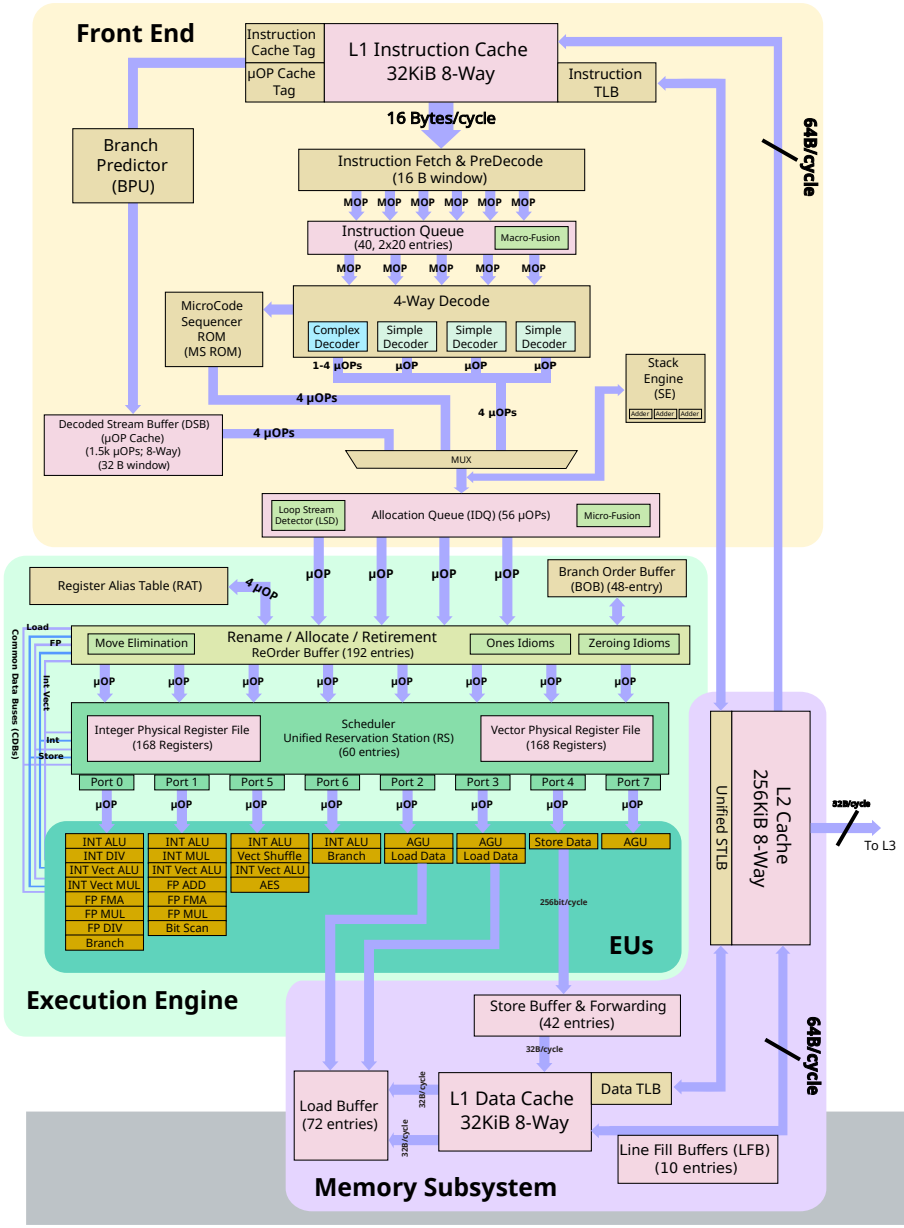
Running it on uniform data we get:

```
$ sudo chrt -f 99 ./a.out -b100 -P0
*** FSE speed analyzer  64-bits, by Yann Collet (Feb 18 2017) ***
Generating 48 KB with P=0.50%
^C-trivialCount                 :   1613.9 MB/s
```

While running it on the data where all values are the same we get:

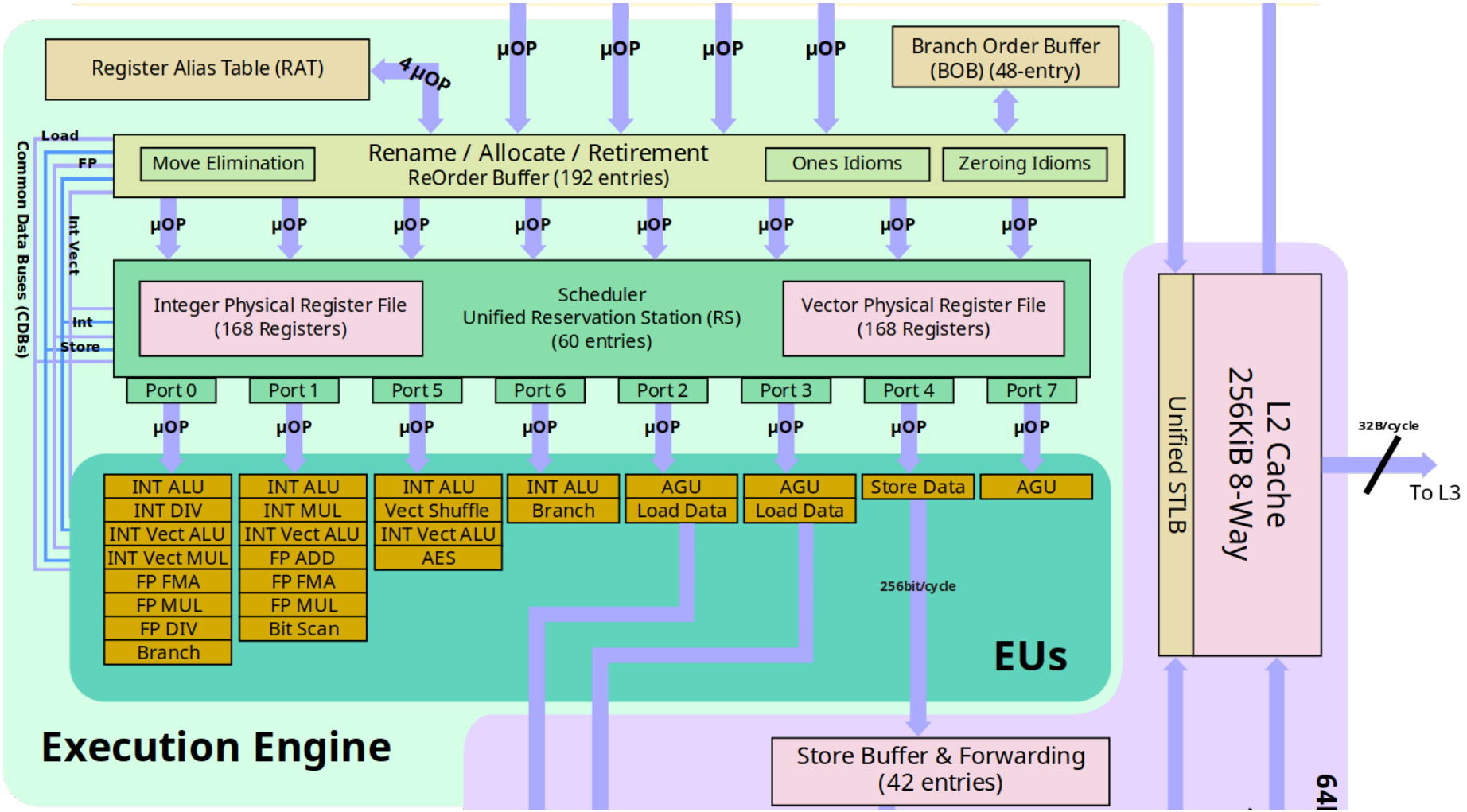
```
$ sudo chrt -f 99 ./a.out -b100 -P100
*** FSE speed analyzer  64-bits, by Yann Collet (Feb 18 2017) ***
Generating 48 KB with P=100.00%
^C-trivialCount                 :    512.7 MB/s
```

Haswell Microarchitecture



Source: https://en.wikichip.org/wiki/intel/microarchitectures/haswell

Haswell Execution Engine



Source: https://en.wikichip.org/wiki/intel/microarchitectures/haswell

Execution Engine

CPU detects dependencies between instructions and execute them in right order.

Slow instructions like memory accesses and division can be overtaken by fast ones.

The execution time is determined by
- The critical path
- The availability of execution units

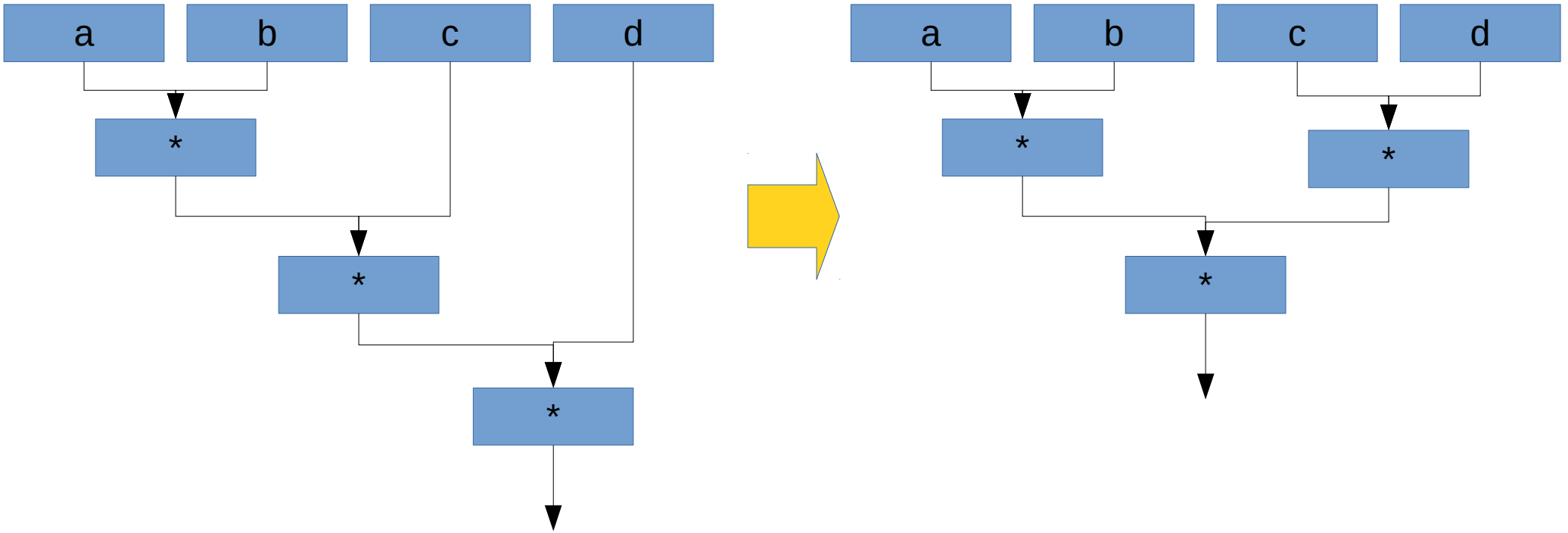
Dependency DAG

Compilers try to reassociate operations to enable more instruction level parallelism and to shorten critical path:

```
int f(int a, int b, int c, int d)
{
    return a * b * c * d;
}

f:
  imul edi, esi
  imul edx, ecx
  imul edx, edi
  mov eax, edx
  ret
```

Dependency DAG

Compilers try to reassociate operations to enable more instruction level parallelism and to shorten critical path:



Example

```
void count_huffman_weights(char const* src, size_t size)
{
    uint32_t count[256] = {0};

    for (size_t i = 0; i != size; ++i)
        ++count[src[i]];
}
```

Let me rewrite our original example into individual operations:

```
inc i
load src[i] → val
load count[val] → tmp
inc tmp
store tmp → count[val]
compare_and_jump
```

Example

Obviously the branch is predictable. So the execution engine get the following steam of instructions:

```
inc i
load src[i] → val
load count[val] → tmp
inc tmp
store tmp → count[val]
compare_and_jump
inc i
load src[i] → val
load count[val] → tmp
inc tmp
store tmp → count[val]
compare_and_jump
inc i
...
```

Example

Can the load of the next iteration be reordered with the store of the previous? It depends on whether they references the same address.

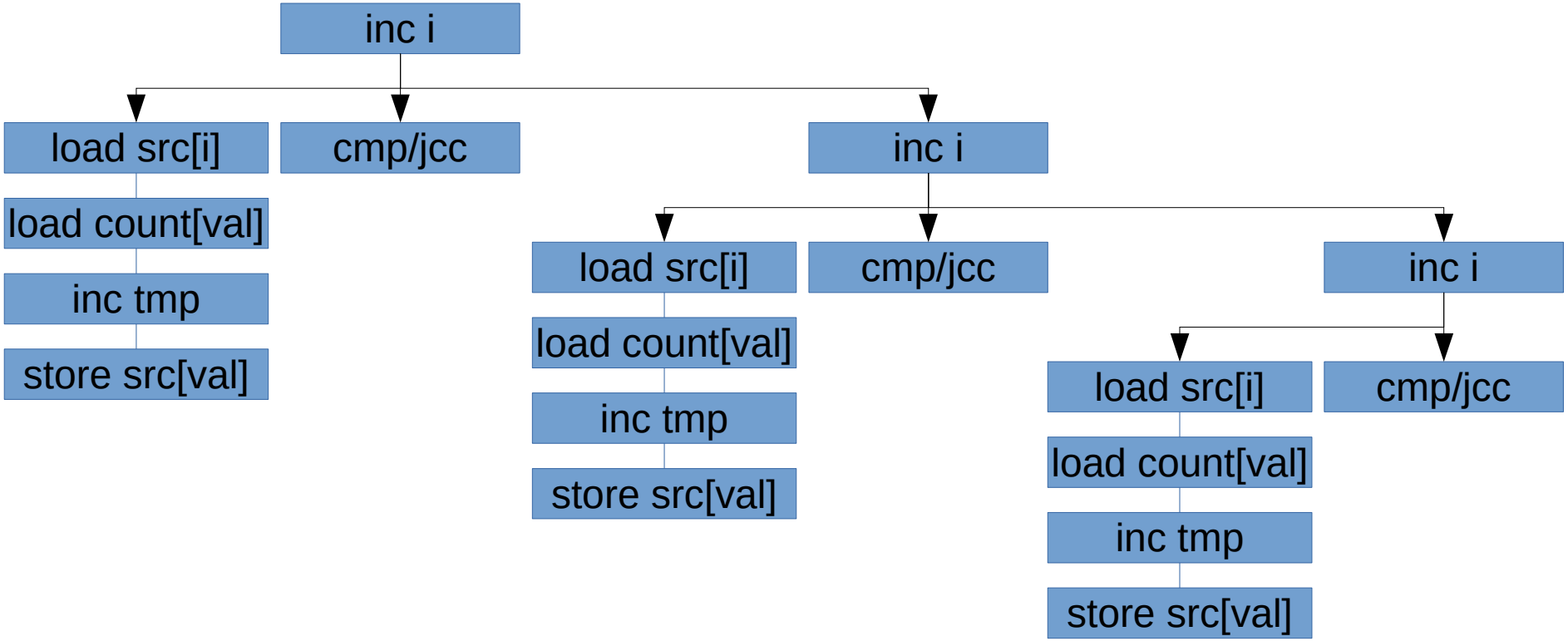
```
inc i
load src[i] → val
load count[val] → tmp
inc tmp
store tmp → count[val]
compare_and_jump
inc i
load src[i] → val
load count[val] → tmp
inc tmp
store tmp → count[val]
compare_and_jump
inc i
...
```

Speculative Loads

Usually loads doesn't alias preceding stores. Therefore CPU tries to start executing them earlier. It does so speculatively. In case the load does alias the preceding store execution need to be restarted.

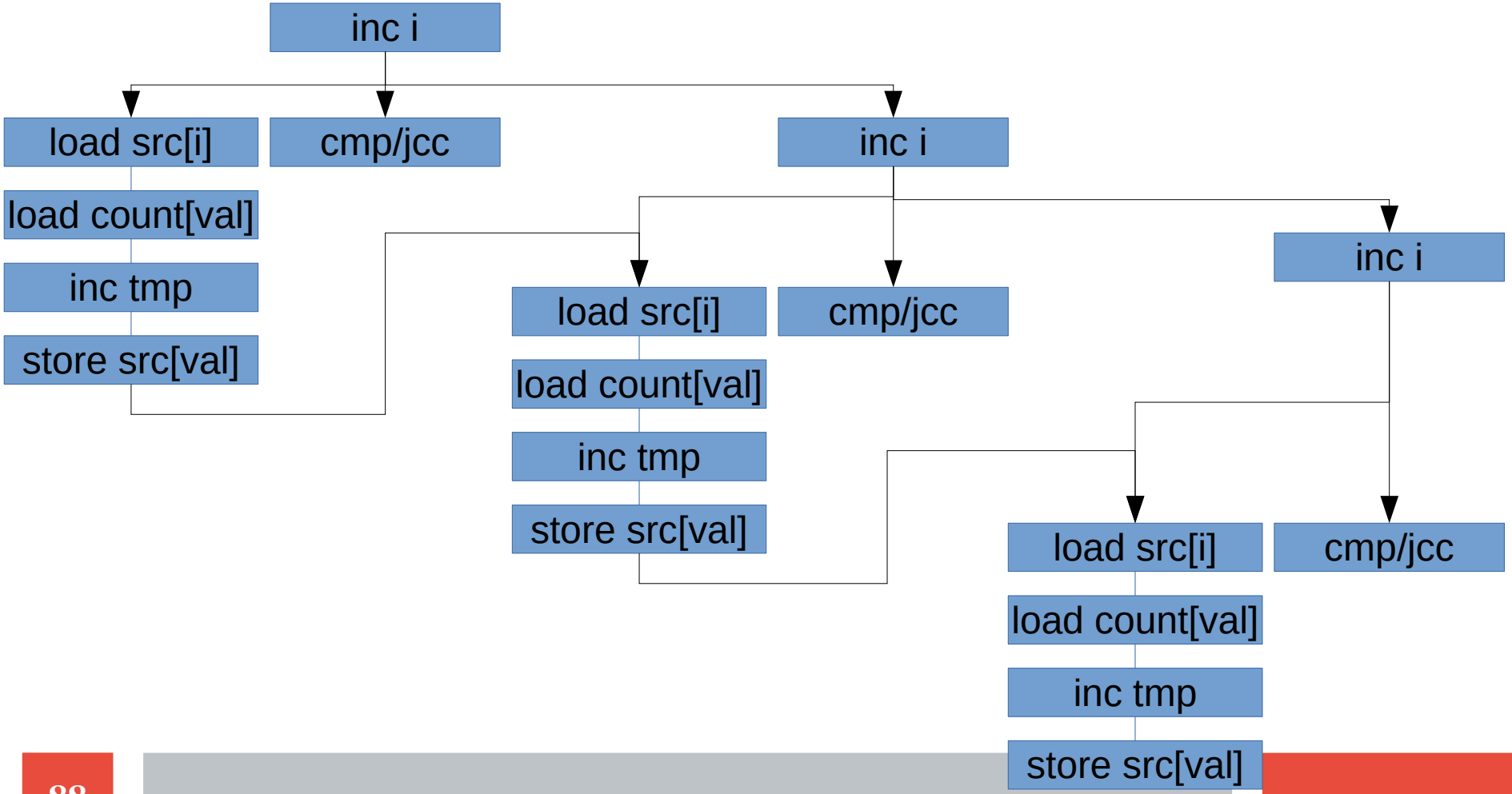
Dependency chain

In non-aliasing case the dependency chain looks like this:



Dependency chain

In non-aliasing case the dependency chain looks like this:



Improved Version

Having this information in mind the original algorithm can be improved:

```
void count_huffman_weights_improved(char const* src, size_t size)
{
    uint32_t count[8][256] = {};

    size = size / 8 * 8;
    for (size_t i = 0; i < size;)
    {
        ++count[0][src[i++]]; ++count[1][src[i++]]; ++count[2][src[i++]];
        ++count[3][src[i++]]; ++count[4][src[i++]]; ++count[5][src[i++]];
        ++count[6][src[i++]]; ++count[7][src[i++]];
    }
}
```

Improved Version

```
$ sudo chrt -f 99 ./a.out -b101 -P100
*** FSE speed analyzer  64-bits, by Yann Collet (Jan 24 2018) ***
Generating 48 KB with P=100.00%
101#count8                      :   1486.7 MB/s   (49152)

$ sudo chrt -f 99 ./a.out -b101 -P0
*** FSE speed analyzer  64-bits, by Yann Collet (Jan 24 2018) ***
Generating 48 KB with P=0.50%
101#count8                      :   1766.4 MB/s   (267)
```

Best Practices

Instruction scheduling is the task of the compiler.
- There is little we can or should do

In some cases compiler assume that pointers can alias while in fact they can not. This causes creating a new critical path.
- It's difficult to give a guidance where this may happen.
- Usually can be found by profiling.

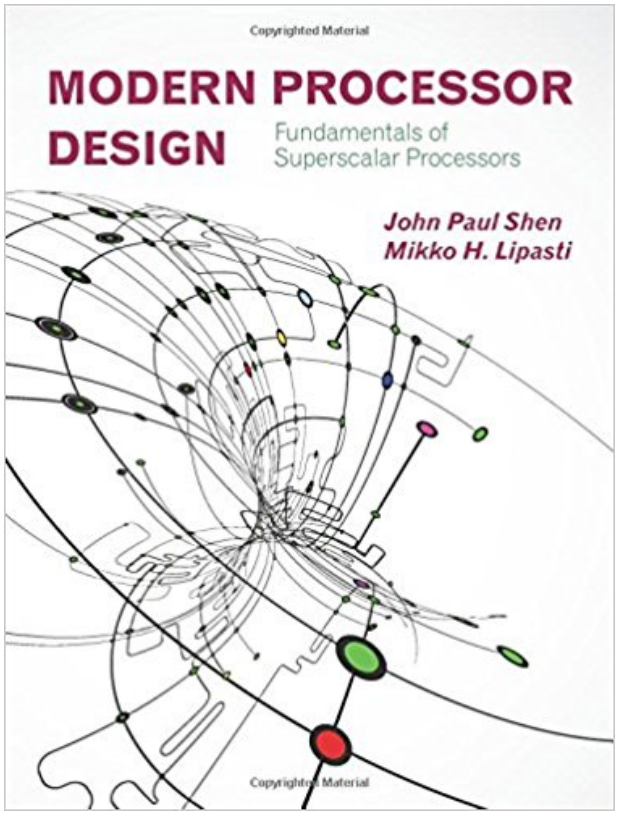
Final Thought

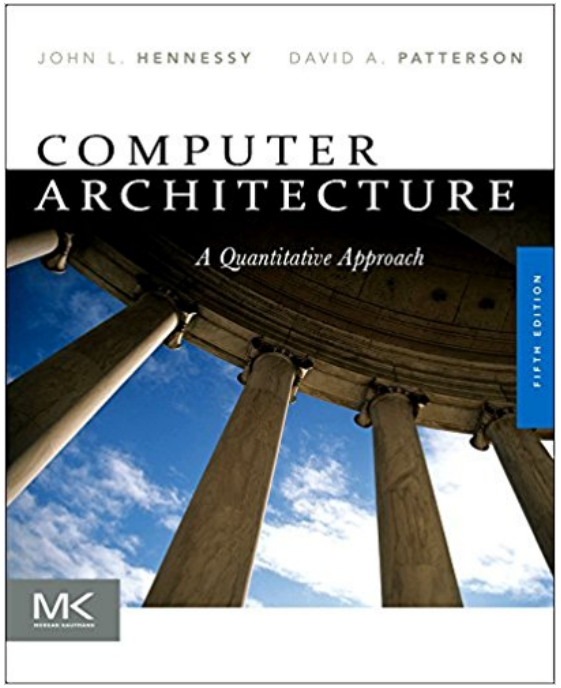
The fastest operation is no operation.

- Often it is easier to eliminate some operations completely than to optimize them.

References

J. Shen, M. Lipasti — Modern Processor Design: Fundamentals of Superscalar Processors





J. Hennessy, D. Patterson — Computer Architecture: A Quantitative Approach

References

- T. Albrecht — Pitfalls of Object Oriented Programming
- https://stackoverflow.com/questions/25078285/replacing-a-32-bit-loop-count-variable-with-64-bit-introduces-crazy-performance
- http://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-an-unsorted-array
- http://info.prelert.com/blog/cpp-stdstring-implementations
- Peter Steinbach — The Performance Addict's Toolbox (Meeting C++ 2017)
- Nicholas Ormrod — The strange details of std::string at Facebook (CppCon 2016)
- Chandler Carruth — Efficiency with Algorithms, Performance with Data Structures (CppCon 2014)

References

- Sean Parent — Inheritance Is The Base Class of Evil https://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil
- Louis Dionne — Runtime Polymorphism: Back to the Basics (CppCon 2017)
- https://github.com/ldionne/dyno

References

- E. Bendersky — Computed goto for efficient dispatch tables https://eli.thegreenplace.net/2012/07/12/computed-goto-for-efficient-dispatch-tables
- D. Michoka — The Common CPU Interpreter Loop Revisited http://www.emulators.com/docs/nx25_nostradamus.htm

Thank you!

Thank you!