corvalius

```
loop:
mov al, byte [rcx]
shl rax, 0xc
jz loop
mov rbx, qword [rbx + rax]
```

Scratched Metal

Federico Lois
Twitter: @federicolois
Github: redknightlois
Repo: performance-course

Not long ago in a city (not that) far, far away…

20.000.000.000

operations per day

and a few calculations later…

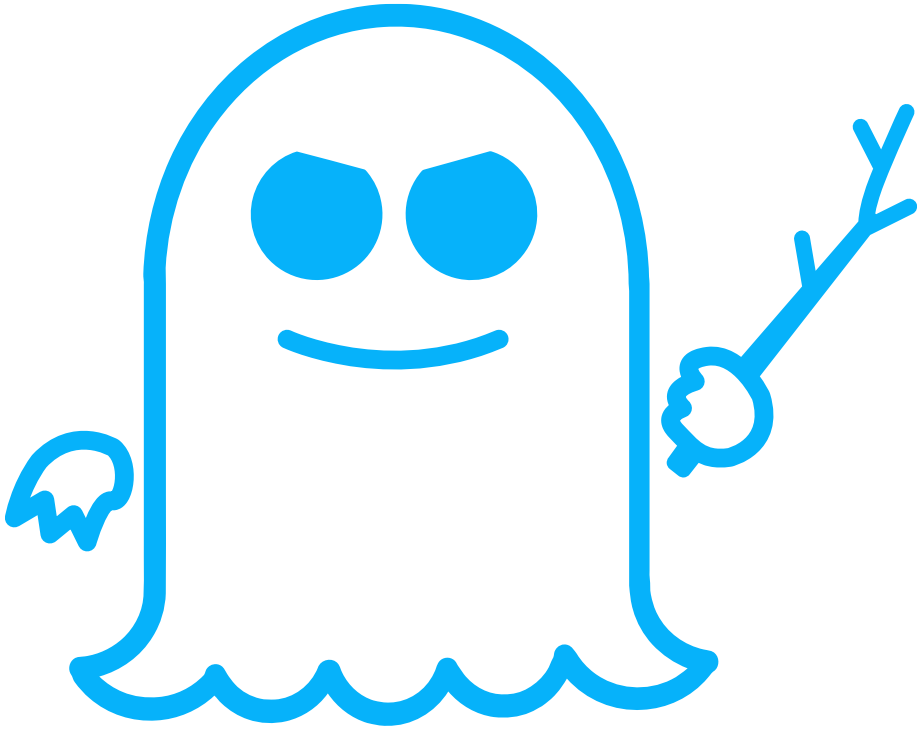
~ 5 ms

per operation
per node


DOTNEXT

On the eve of 2018…

MELTDOWN



SPECTRE

Federico Andres Lois @federicolois · 3 ene.

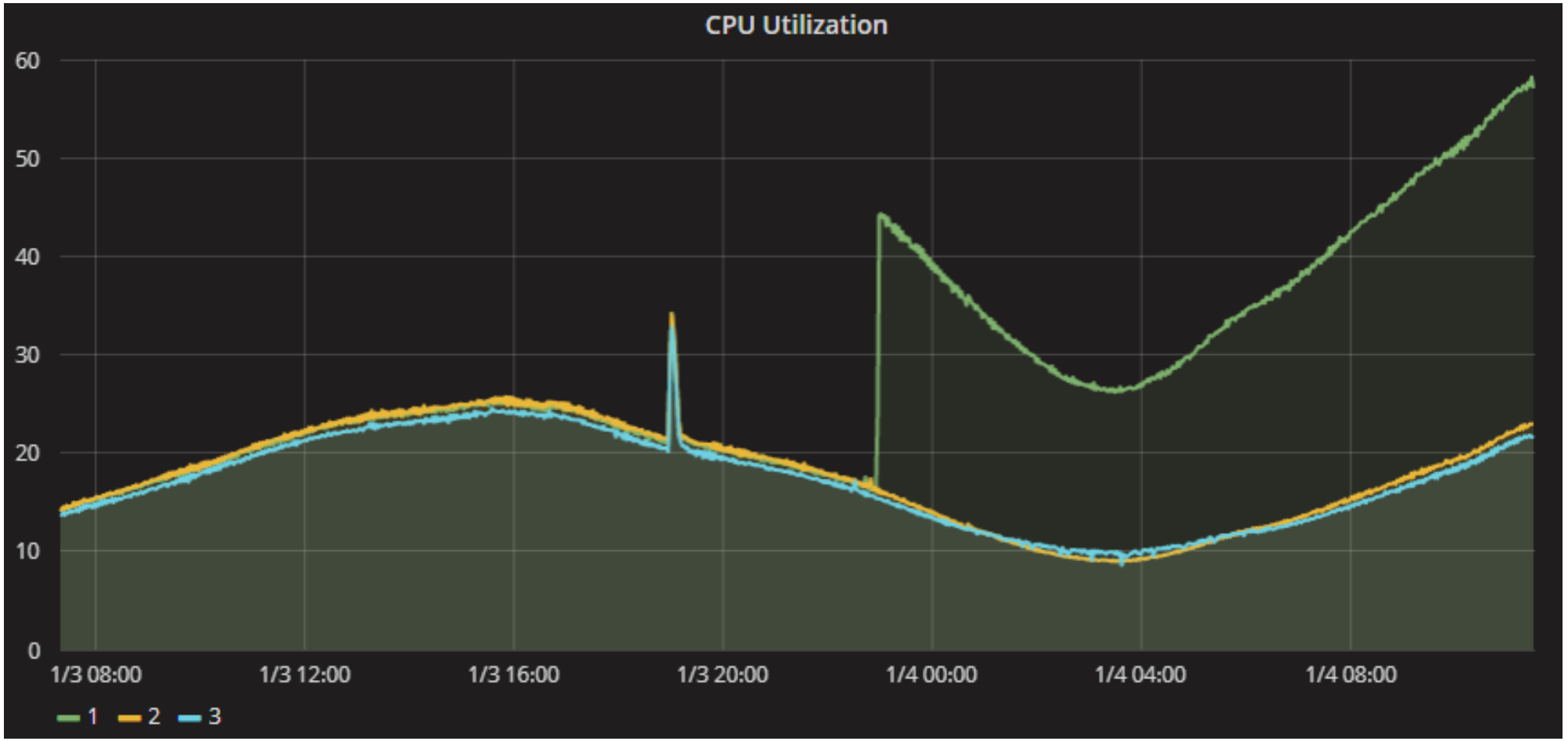
Holy Madness Batman!!! The monster is here looking to eat all my performance away.

Traducir del inglés

Kernel-memory-leaking Intel processor design flaw forces Linux, Win...

Speed hits loom, other OSes need fixes

theregister.co.uk



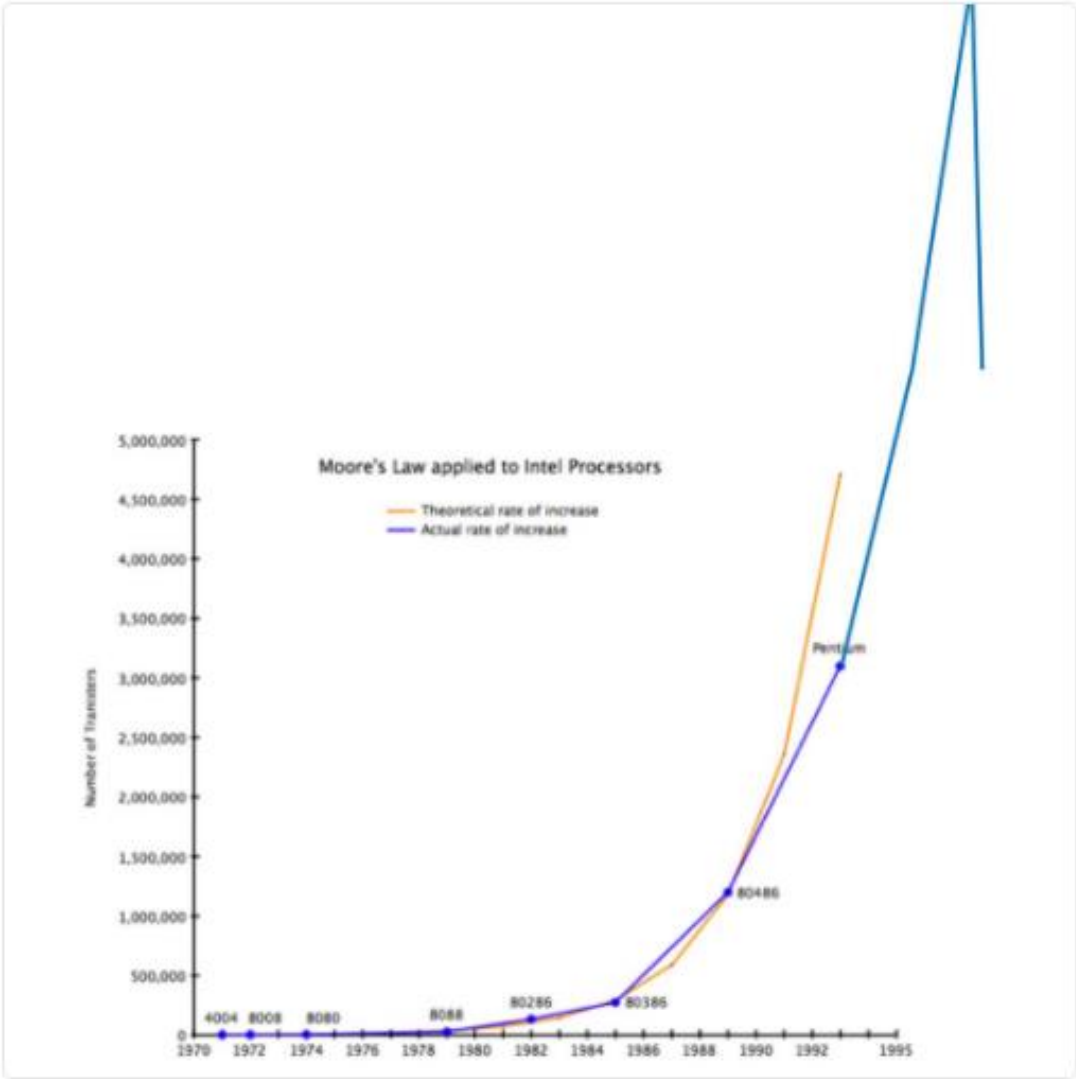
https://www.epicgames.com/fortnite/forums/news/announcements/132642-epic-services-stability-update

Retwitteado por ti

Roger Johansson @RogerAlsing · 6 ene.

.@intel I fixed this diagram for you #Meltdown

Traducir del inglés



We have been running on borrowed performance

And its time we pay our debt to the performance gods

Things we all should know (aka the ground truth)

- The compiler and JIT are usually good enough
  - Its our job to help them do “The Right Thing” ™
- Cache misses and memory effects usually dominate.
  - Cache pollution can slow you down to a crawl.
  - Its our job to keep the zoo in check.
- We are aware of data dependencies.
- The instruction set is huge, we use a cheatsheet
  - And don’t forget, install AsmDude ☺
- All tricks are allowed.

If you have to micro-optimize make sure there is NO other way.

Diffing Memory at RAVENDB

- Memory diffing is core of the journaling system
- Write transactions are isolated, until they are written to disk.
  - Memory diffing allows us to know what to write ☺

Diffing Memory at 

- It is a very simple algorithm (on the surface)
- It is also in the critical path of our code
- It showcase many different bottlenecks
- Single loop takes 99.9% of the total execution time

HARDCORE

Assembly
Bit Twiddling
Microarchitecture

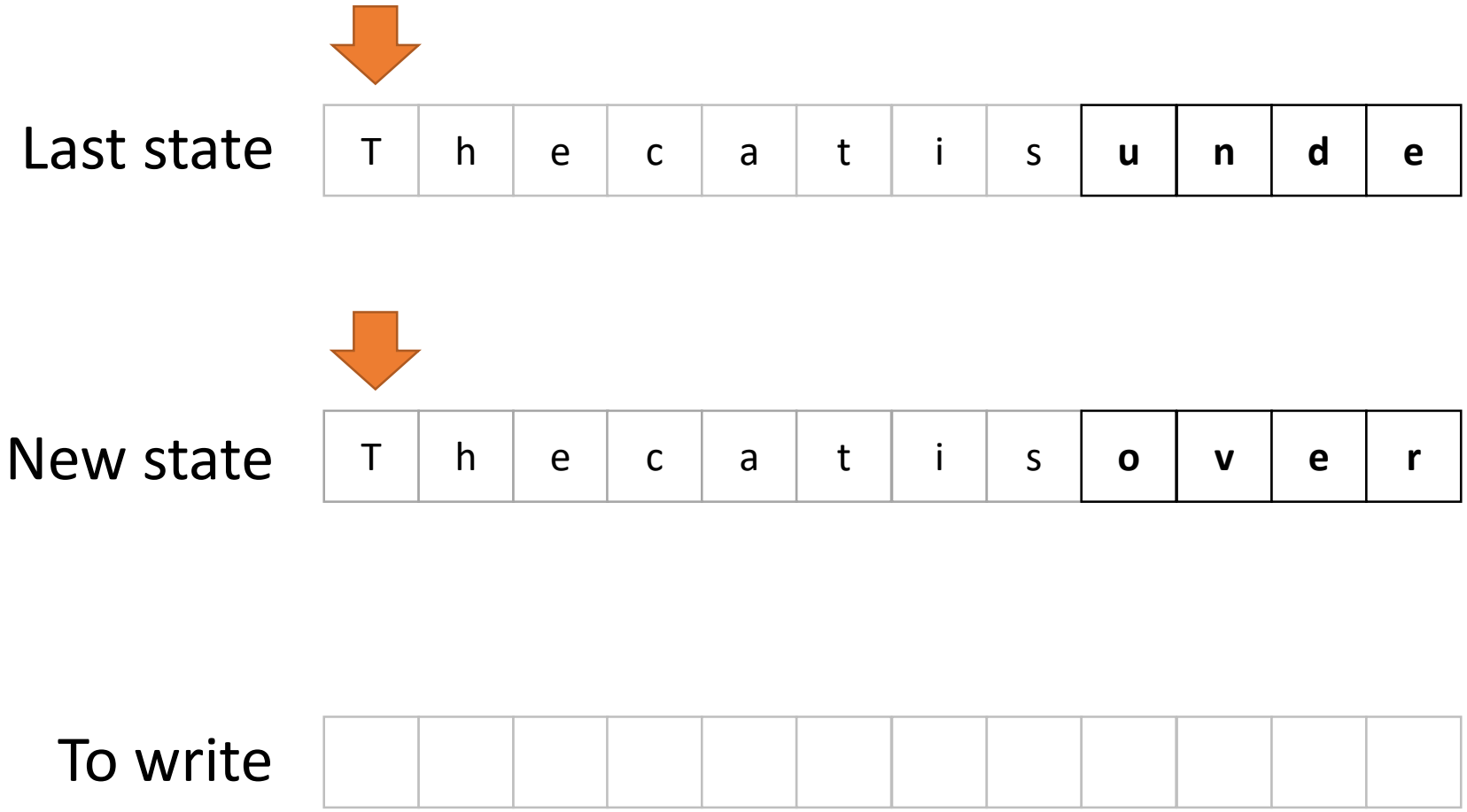
DOTNEXT CONTENT RATING

CoreCLR ♥ SIMD

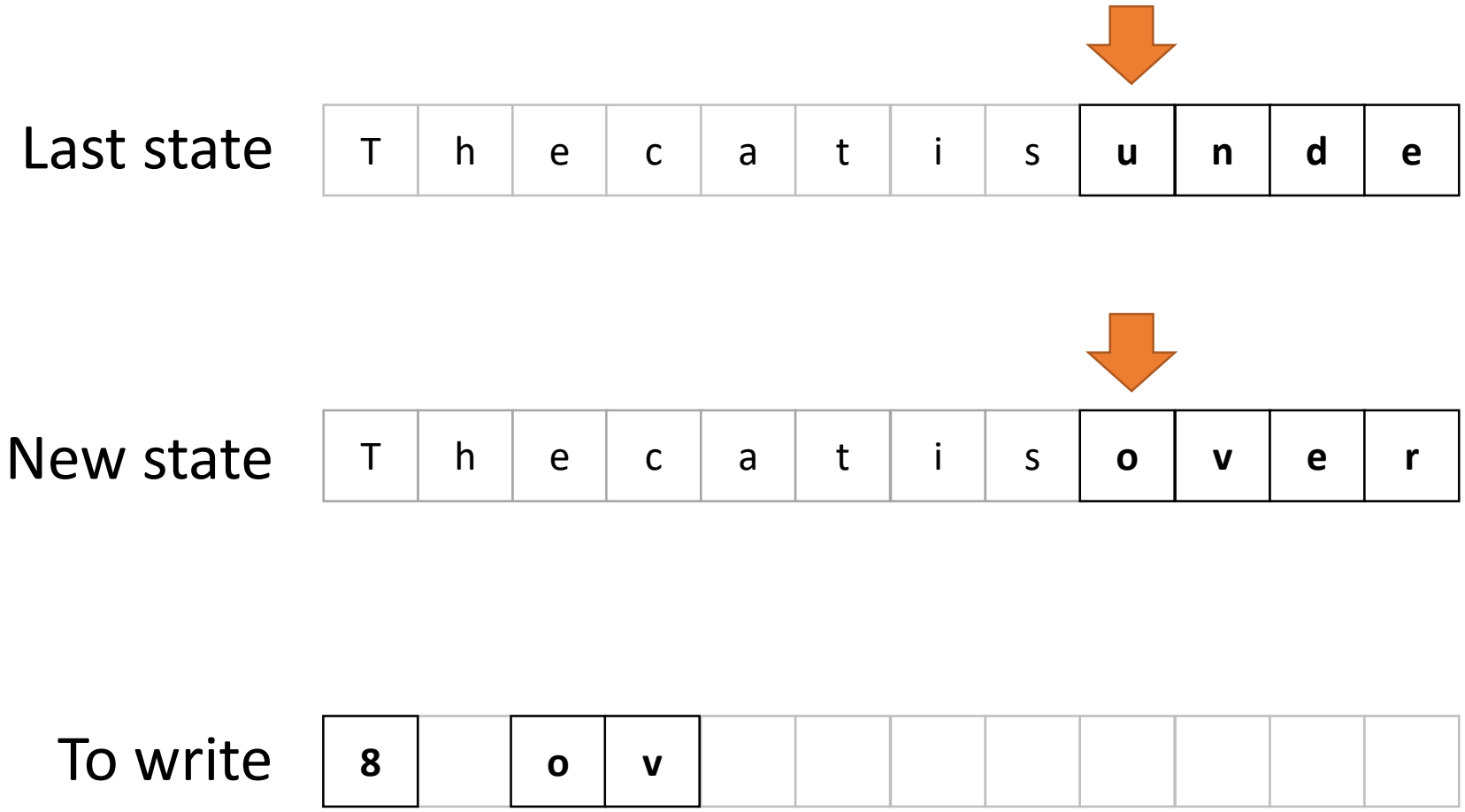
- “Recently” enabled through JIT intrinsecs in CoreCLR 2.1
  - Complete SSE and AVX instruction set
- API mimics C++ intrinsics
  - JIT will translate them to hardware instructions
- “Shoot yourself in the foot” style API (by design)
  - As it should be ☺
- Available on preview builds for x64 hardware

An entire new dimension of available optimizations

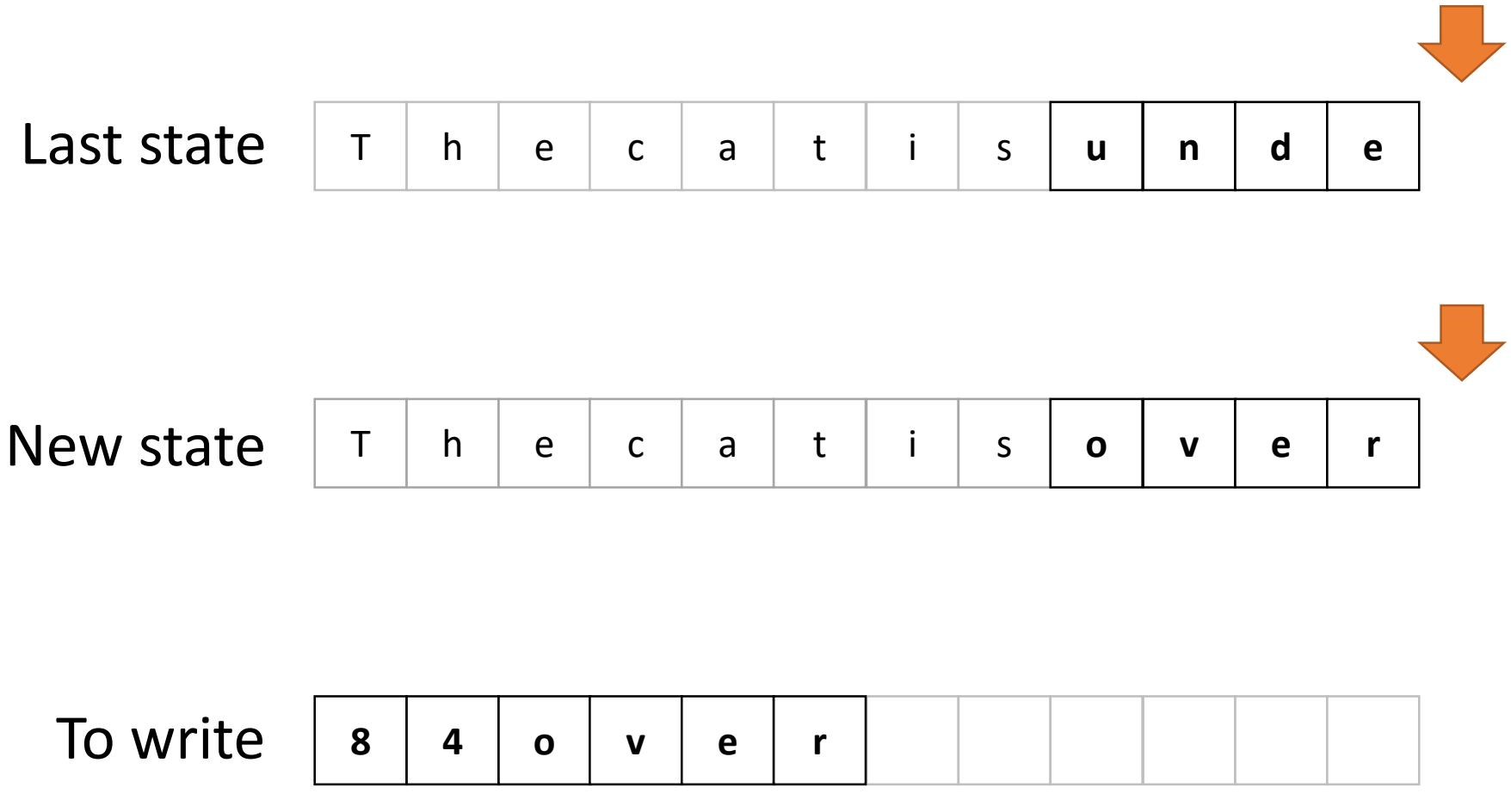
Naive Diff



Naive Diff


Last state
T h e c a t i s u n d e
New state
T h e c a t i s o v e r
To write
8 o v

Naive Diff


Last state
T h e c a t i s u n d e
New state
T h e c a t i s o v e r
To write
8 4 o v e r

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 1, originalPtr += 1, modifiedPtr += 1)
{
    byte m = *modifiedPtr;
    byte o = *originalPtr;

    if (m != o)
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        *(writePtr + writePtrOffset) = m;
        writePtrOffset++;
    }
    else if (started) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 1, originalPtr += 1, modifiedPtr += 1)
{
    byte m = *modifiedPtr;
    byte o = *originalPtr;

    if (m != o)
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        *(writePtr + writePtrOffset) = m;
        writePtrOffset++;
    }
    else if (started) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

‘THE’ Main Loop

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 1, originalPtr += 1, modifiedPtr += 1)
{
    byte m = *modifiedPtr;
    byte o = *originalPtr;

    if (m != o)
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        *(writePtr + writePtrOffset) = m;
        writePtrOffset++;
    }
    else if (started) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

Iteration when memory is unchanged

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 1, originalPtr += 1, modifiedPtr += 1)
{
    byte m = *modifiedPtr;
    byte o = *originalPtr;

    if (m != o)
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        *(writePtr + writePtrOffset) = m;
        writePtrOffset++;
    }
    else if (started) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

Iteration when memory has changed

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 1, originalPtr += 1, modifiedPtr += 1)
{
    byte m = *modifiedPtr;
    byte o = *originalPtr;

    if (m != o)
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        *(writePtr + writePtrOffset) = m;
        writePtrOffset++;
    }
    else if (started) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

Starting a new changed block

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 1, originalPtr += 1, modifiedPtr += 1)
{
    byte m = *modifiedPtr;
    byte o = *originalPtr;

    if (m != o)
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        *(writePtr + writePtrOffset) = m;
        writePtrOffset++;
    }
    else if (started) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

Ending a changed block

- Works byte-by-byte
- Is able to process 700Mb/sec on the reference machine
  - Intel i7-7700 (Kaby-Lake)
  - 32Gb DDR4-2400 (1200Mhz) on dual channel

(Warning, trick questions ahead ☺)

Can we do better?

Clockticks:	1,141,248,240,000	
Instructions Retired:	3,698,555,400,000	
CPI Rate:	0.309	
MUX Reliability:	0.842	
Front-End Bound:	4.4%	of Pipeline Slots
Front-End Latency:	3.0%	of Pipeline Slots
Front-End Bandwidth:	1.4%	of Pipeline Slots
Bad Speculation:	2.3%	of Pipeline Slots
Branch Mispredict:	1.8%	of Pipeline Slots
Machine Clears:	0.5%	of Pipeline Slots
Back-End Bound:	40.3%	of Pipeline Slots
Memory Bound:	5.3%	of Pipeline Slots
Core Bound:	35.0%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	33.4%	of Clockticks
Cycles of 0 Ports Utilized:	4.4%	of Clockticks
Cycles of 1 Port Utilized:	2.1%	of Clockticks
Cycles of 2 Ports Utilized:	11.8%	of Clockticks
Cycles of 3+ Ports Utilized:	29.9%	of Clockticks
Vector Capacity Usage (FPU):	12.5%	
Retiring:	53.0%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

Clockticks:	1,141,248,240,000	
Instructions Retired:	3,698,555,400,000	
CPI Rate:	0.309	
MUX Reliability:	0.842	
Front-End Bound:	4.4%	of Pipeline Slots
Front-End Latency:	3.0%	of Pipeline Slots
Front-End Bandwidth:	1.4%	of Pipeline Slots
Bad Speculation:	2.3%	of Pipeline Slots
Branch Mispredict:	1.8%	of Pipeline Slots
Machine Clears:	0.5%	of Pipeline Slots
Back-End Bound:	40.3%	of Pipeline Slots
Memory Bound:	5.3%	of Pipeline Slots
Core Bound:	35.0%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	33.4%	of Clockticks
Cycles of 0 Ports Utilized:	4.4%	of Clockticks
Cycles of 1 Port Utilized:	2.1%	of Clockticks
Cycles of 2 Ports Utilized:	11.8%	of Clockticks
Cycles of 3+ Ports Utilized:	29.9%	of Clockticks
Vector Capacity Usage (FPU):	12.5%	
Retiring:	53.0%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

CPI Rate: 0.309
Where Theoretical Best is 0.25

Clockticks:	1,141,248,240,000	
Instructions Retired:	3,698,555,400,000	
CPI Rate:	0.309	
MUX Reliability:	0.842	
Front-End Bound:	4.4%	of Pipeline Slots
Front-End Latency:	3.0%	of Pipeline Slots
Front-End Bandwidth:	1.4%	of Pipeline Slots
Bad Speculation:	2.3%	of Pipeline Slots
Branch Mispredict:	1.8%	of Pipeline Slots
Machine Clears:	0.5%	of Pipeline Slots
Back-End Bound:	40.3%	of Pipeline Slots
Memory Bound:	5.3%	of Pipeline Slots
Core Bound:	35.0%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	33.4%	of Clockticks
Cycles of 0 Ports Utilized:	4.4%	of Clockticks
Cycles of 1 Port Utilized:	2.1%	of Clockticks
Cycles of 2 Ports Utilized:	11.8%	of Clockticks
Cycles of 3+ Ports Utilized:	29.9%	of Clockticks
Vector Capacity Usage (FPU):	12.5%	
Retiring:	53.0%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

CPI Rate: 0.309
Where Theoretical Best is 0.25

Core Bound: 35%
It's not you, it's the CPU

Clockticks:	1,141,248,240,000	
Instructions Retired:	3,698,555,400,000	
CPI Rate:	0.309	
MUX Reliability:	0.842	
Front-End Bound:	4.4%	of Pipeline Slots
Front-End Latency:	3.0%	of Pipeline Slots
Front-End Bandwidth:	1.4%	of Pipeline Slots
Bad Speculation:	2.3%	of Pipeline Slots
Branch Mispredict:	1.8%	of Pipeline Slots
Machine Clears:	0.5%	of Pipeline Slots
Back-End Bound:	40.3%	of Pipeline Slots
Memory Bound:	5.3%	of Pipeline Slots
Core Bound:	35.0%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	33.4%	of Clockticks
Cycles of 0 Ports Utilized:	4.4%	of Clockticks
Cycles of 1 Port Utilized:	2.1%	of Clockticks
Cycles of 2 Ports Utilized:	11.8%	of Clockticks
Cycles of 3+ Ports Utilized:	29.9%	of Clockticks
Vector Capacity Usage (FPU):	12.5%	
Retiring:	53.0%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

CPI Rate: 0.309
Where Theoretical Best is 0.25

Core Bound: 35%
It’s not you, it’s the CPU

Retiring: 53%
50% of the slots doing actual work

Clockticks:	1,141,248,240,000	
Instructions Retired:	3,698,555,400,000	
CPI Rate:	0.309	
MUX Reliability:	0.842	
Front-End Bound:	4.4%	of Pipeline Slots
Front-End Latency:	3.0%	of Pipeline Slots
Front-End Bandwidth:	1.4%	of Pipeline Slots
Bad Speculation:	2.3%	of Pipeline Slots
Branch Mispredict:	1.8%	of Pipeline Slots
Machine Clears:	0.5%	of Pipeline Slots
Back-End Bound:	40.3%	of Pipeline Slots
Memory Bound:	5.3%	of Pipeline Slots
Core Bound:	35.0%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	33.4%	of Clockticks
Cycles of 0 Ports Utilized:	4.4%	of Clockticks
Cycles of 1 Port Utilized:	2.1%	of Clockticks
Cycles of 2 Ports Utilized:	11.8%	of Clockticks
Cycles of 3+ Ports Utilized:	29.9%	of Clockticks
Vector Capacity Usage (FPU):	12.5%	
Retiring:	53.0%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

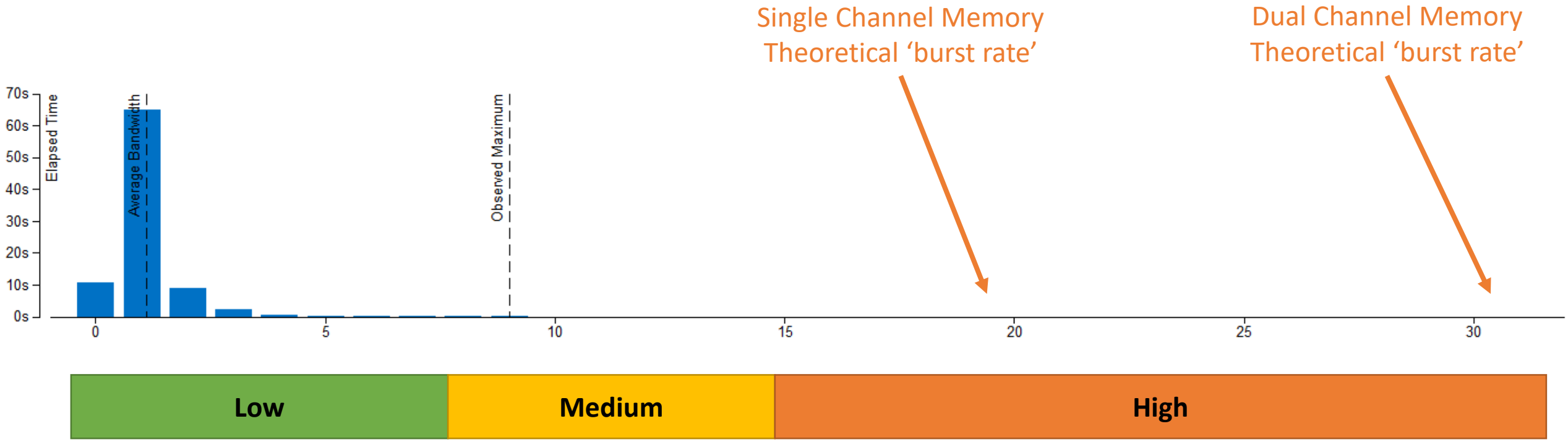
CPI Rate: 0.309
Where Theoretical Best is 0.25

Core Bound: 35%
It’s not you, it’s the CPU

Retiring: 53%
50% of the slots doing actual work

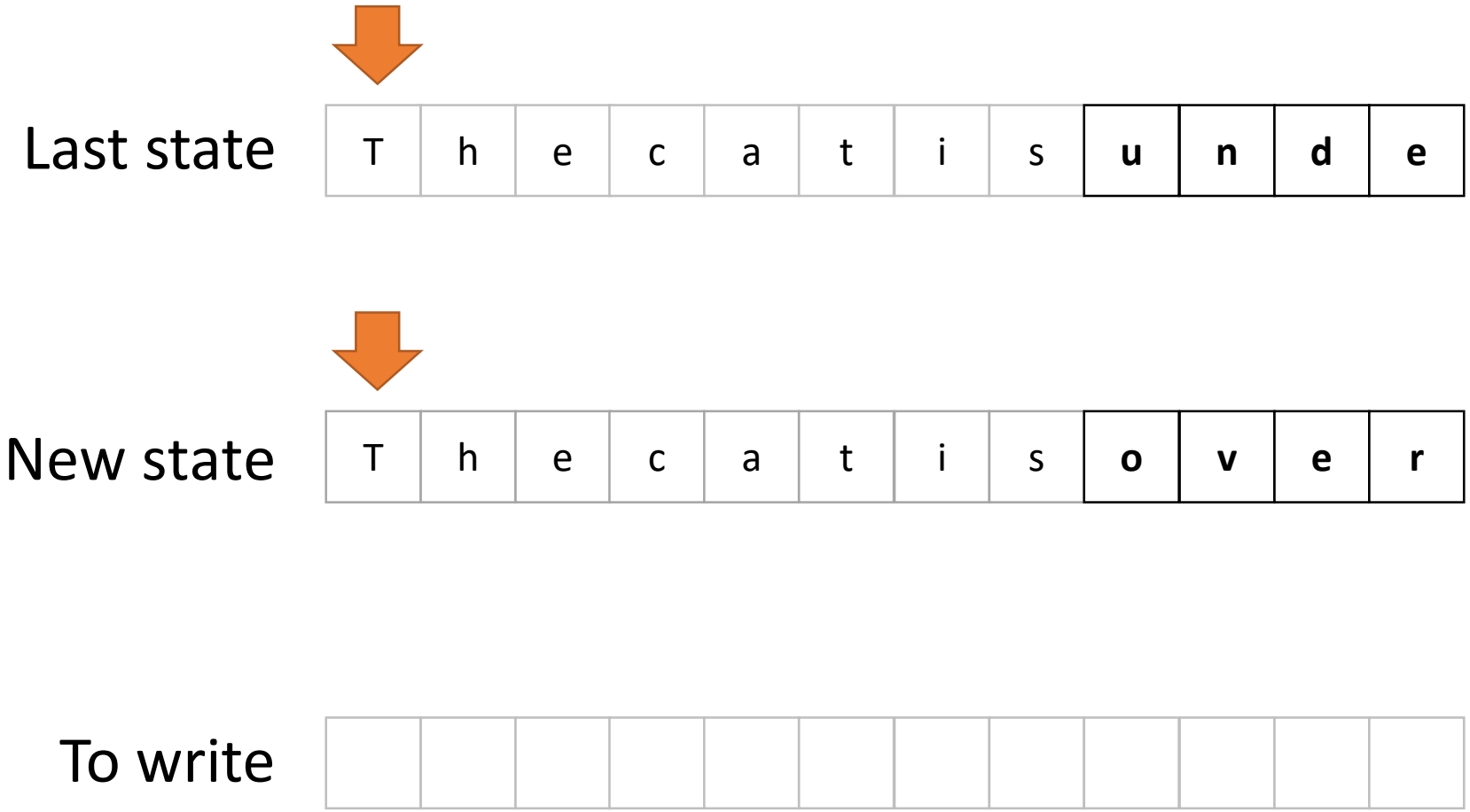
Cycles of 3+ Ports Utilized: 29.9%
We have lots of instruction parallelism

What could possibly go wrong?

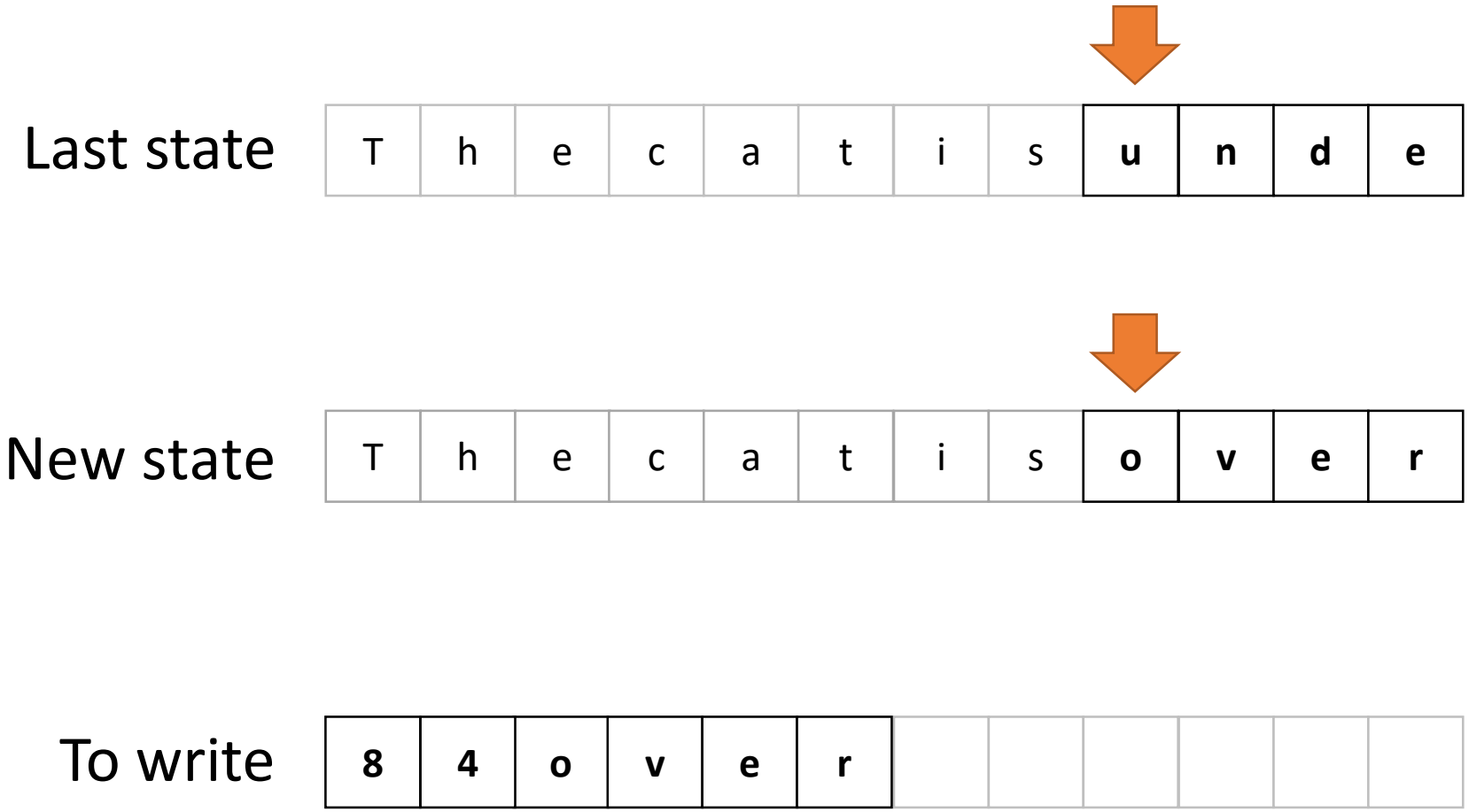


Bandwidth Utilization of Naïve Byte-by-byte

(A better) Naive Diff



(A better) Naive Diff



Which ‘word’ size then?

- CoreCLR 1.0 -> 2.0
  - Scalar Long – 8 bytes
  - System.Numerics – 16 to 32bytes depending runtime
    - No control!!!
    - Only arithmetic instructions
- CoreCLR 2.1:
  - Scalar Long – 8 bytes
  - SSE Vector – 16 bytes
  - AVX Vector – 32 bytes
- Future:
  - AVX512 Vector – 64 bytes

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 1, originalPtr += 1, modifiedPtr += 1)
{
    byte m = *modifiedPtr;
    byte o = *originalPtr;

    if (m != o)
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        *(writePtr + writePtrOffset) = m;
        writePtrOffset++;
    }
    else if (started) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

Strides over the words

Load and Compare

Write the modified data on the buffer

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 32, originalPtr += 32, modifiedPtr += 32)
{
    Vector256<byte> m = Avx.LoadVector256(modifiedPtr);
    Vector256<byte> o = Avx.LoadVector256(originalPtr);

    o = Avx2.Xor(o, m);
    if (!Avx.TestZ(o, o))
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        Avx.Store((writePtr + writePtrOffset), m);
        writePtrOffset++;
    }
    else if (started) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

Strides over the words

Load and Compare

Write the modified data on the buffer

Clockticks:	137,118,960,000	
Instructions Retired:	161,174,520,000	
CPI Rate:	0.851	
MUX Reliability:	0.990	
Front-End Bound:	2.6%	of Pipeline Slots
Front-End Latency:	1.8%	of Pipeline Slots
Front-End Bandwidth:	0.8%	of Pipeline Slots
Bad Speculation:	1.0%	of Pipeline Slots
Branch Mispredict:	1.0%	of Pipeline Slots
Machine Clears:	0.0%	of Pipeline Slots
Back-End Bound:	70.3%	of Pipeline Slots
Memory Bound:	51.9%	of Pipeline Slots
Core Bound:	18.4%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	17.3%	of Clockticks
Cycles of 0 Ports Utilized:	26.8%	of Clockticks
Cycles of 1 Port Utilized:	8.9%	of Clockticks
Cycles of 2 Ports Utilized:	7.7%	of Clockticks
Cycles of 3+ Ports Utilized:	8.8%	of Clockticks
Vector Capacity Usage (FPU):	0.0%	
Retiring:	26.1%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

Clockticks:	137,118,960,000	
Instructions Retired:	161,174,520,000	
CPI Rate:	0.851	
MUX Reliability:	0.990	
Front-End Bound:	2.6%	of Pipeline Slots
Front-End Latency:	1.8%	of Pipeline Slots
Front-End Bandwidth:	0.8%	of Pipeline Slots
Bad Speculation:	1.0%	of Pipeline Slots
Branch Mispredict:	1.0%	of Pipeline Slots
Machine Clears:	0.0%	of Pipeline Slots
Back-End Bound:	70.3%	of Pipeline Slots
Memory Bound:	51.9%	of Pipeline Slots
Core Bound:	18.4%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	17.3%	of Clockticks
Cycles of 0 Ports Utilized:	26.8%	of Clockticks
Cycles of 1 Port Utilized:	8.9%	of Clockticks
Cycles of 2 Ports Utilized:	7.7%	of Clockticks
Cycles of 3+ Ports Utilized:	8.8%	of Clockticks
Vector Capacity Usage (FPU):	0.0%	
Retiring:	26.1%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

CPI Rate (was): 0.309
Still HPC worth, but we regressed

Clockticks:	137,118,960,000	
Instructions Retired:	161,174,520,000	
CPI Rate:	0.851	
MUX Reliability:	0.990	
Front-End Bound:	2.6%	of Pipeline Slots
Front-End Latency:	1.8%	of Pipeline Slots
Front-End Bandwidth:	0.8%	of Pipeline Slots
Bad Speculation:	1.0%	of Pipeline Slots
Branch Mispredict:	1.0%	of Pipeline Slots
Machine Clears:	0.0%	of Pipeline Slots
Back-End Bound:	70.3%	of Pipeline Slots
Memory Bound:	51.9%	of Pipeline Slots
Core Bound:	18.4%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	17.3%	of Clockticks
Cycles of 0 Ports Utilized:	26.8%	of Clockticks
Cycles of 1 Port Utilized:	8.9%	of Clockticks
Cycles of 2 Ports Utilized:	7.7%	of Clockticks
Cycles of 3+ Ports Utilized:	8.8%	of Clockticks
Vector Capacity Usage (FPU):	0.0%	
Retiring:	26.1%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

CPI Rate (was): 0.309
Still HPC worth, but we regressed

Retiring (was): 53%
We regressed HARD

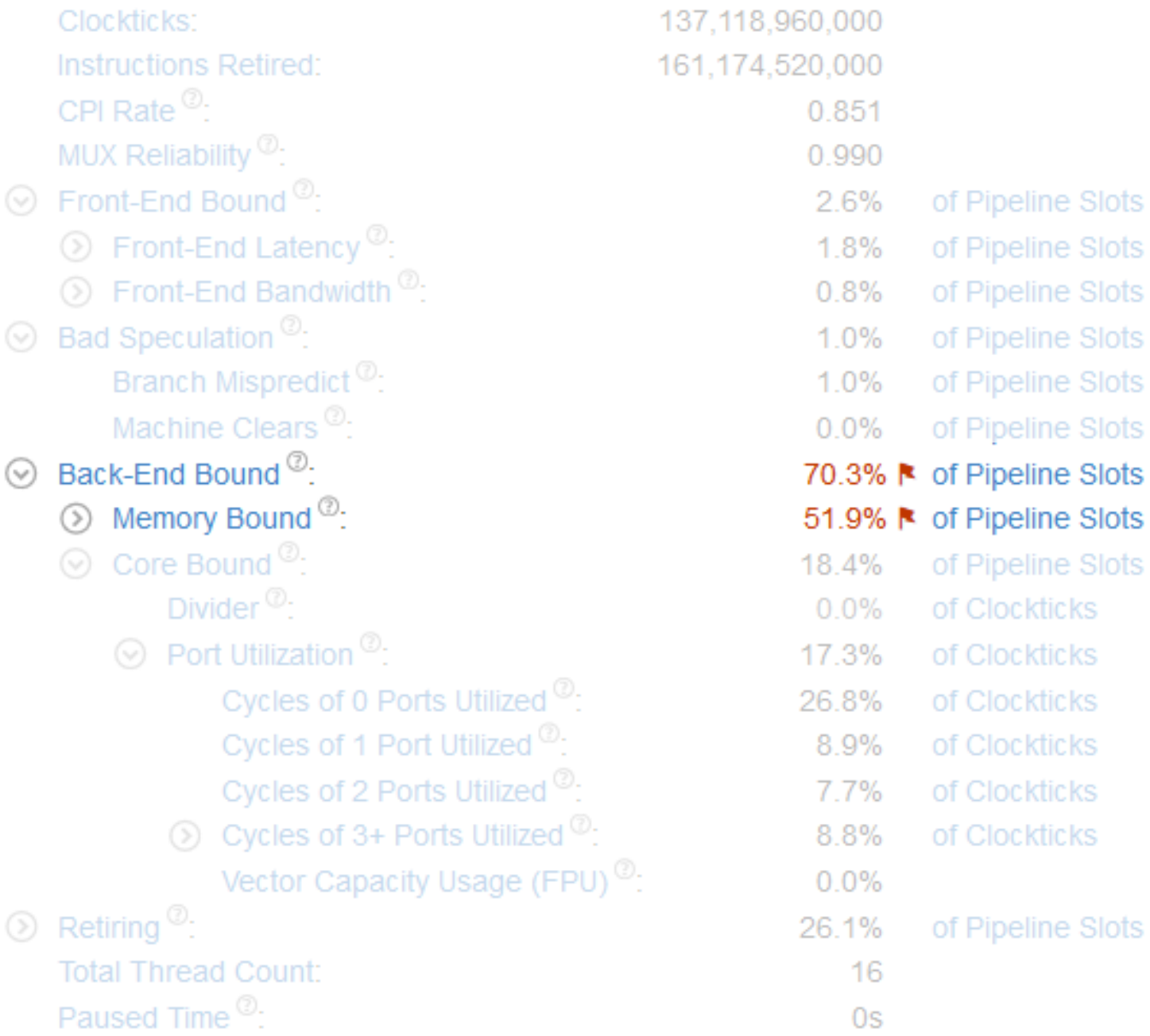
Clockticks:	137,118,960,000	
Instructions Retired:	161,174,520,000	
CPI Rate:	0.851	
MUX Reliability:	0.990	
Front-End Bound:	2.6%	of Pipeline Slots
Front-End Latency:	1.8%	of Pipeline Slots
Front-End Bandwidth:	0.8%	of Pipeline Slots
Bad Speculation:	1.0%	of Pipeline Slots
Branch Mispredict:	1.0%	of Pipeline Slots
Machine Clears:	0.0%	of Pipeline Slots
Back-End Bound:	70.3%	of Pipeline Slots
Memory Bound:	51.9%	of Pipeline Slots
Core Bound:	18.4%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	17.3%	of Clockticks
Cycles of 0 Ports Utilized:	26.8%	of Clockticks
Cycles of 1 Port Utilized:	8.9%	of Clockticks
Cycles of 2 Ports Utilized:	7.7%	of Clockticks
Cycles of 3+ Ports Utilized:	8.8%	of Clockticks
Vector Capacity Usage (FPU):	0.0%	
Retiring:	26.1%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

CPI Rate (was): 0.309
Still HPC worth, but we regressed

Retiring (was): 53%
We regressed HARD

Cycles of 3+ Ports Utilized (was): 29.9%
Did I mention we also regressed?



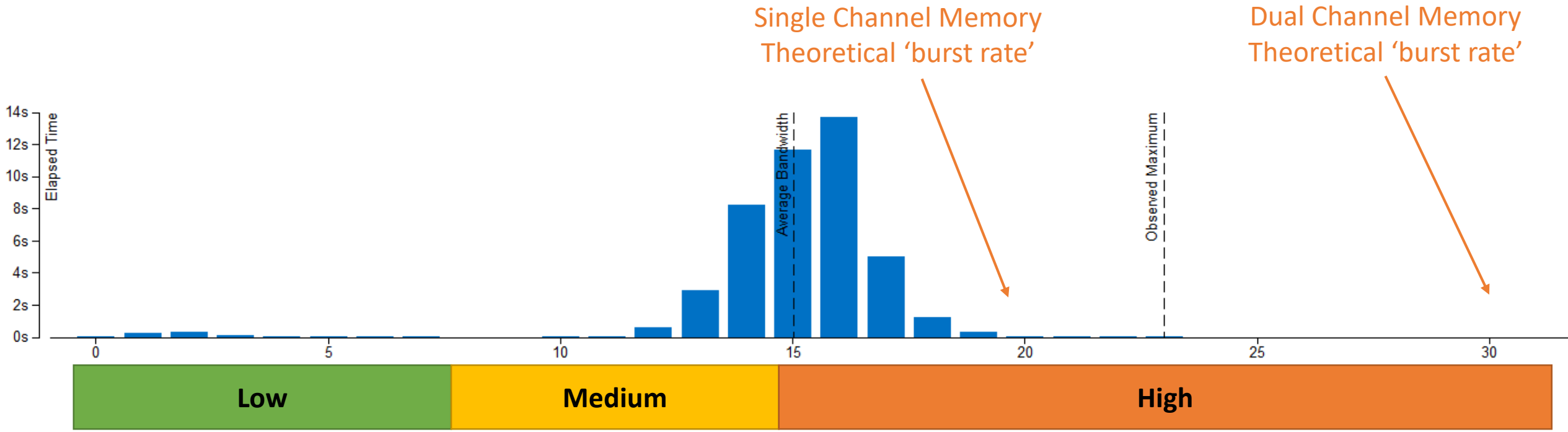
HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

CPI Rate (was): 0.309
Still HPC worth, but we regressed

Retiring (was): 53%
We regressed HARD

Cycles of 3+ Ports Utilized (was): 29.9%
Did I mention we also regressed?

Backend Bound (was): 40.3%
Memory Bound: 51.9%
That's new!!! Intrigued?



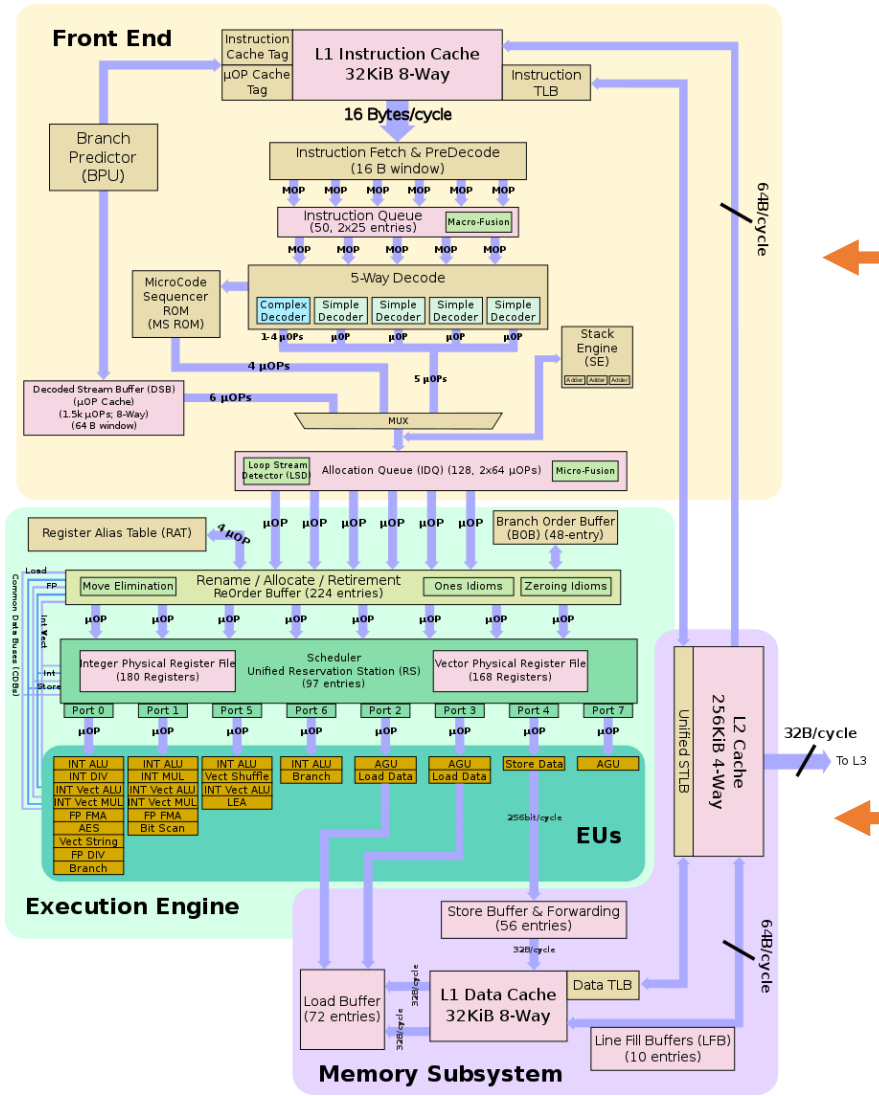
Bandwidth Utilization of (better) Naïve AVX2

A zoo of memory effects

- Memory is a strange beast
  - Many levels that would influence performance
  - It is slooooooooooow.
    - And not getting much faster
  - Difficult to measure & isolate effects
    - Latency effects would skew distribution
    - When approaching max bandwidth all hell breaks loose

Know our tool!

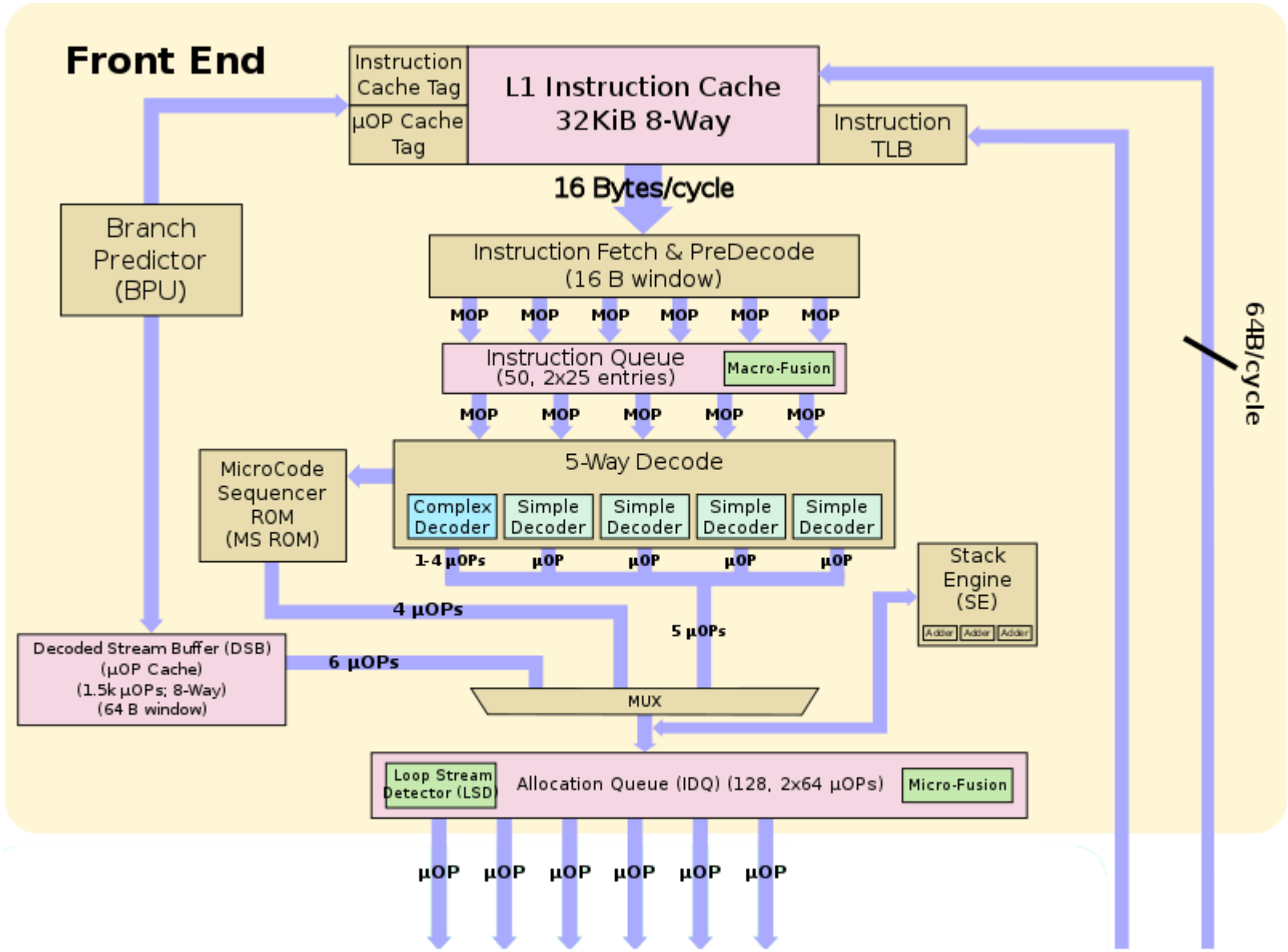
HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE



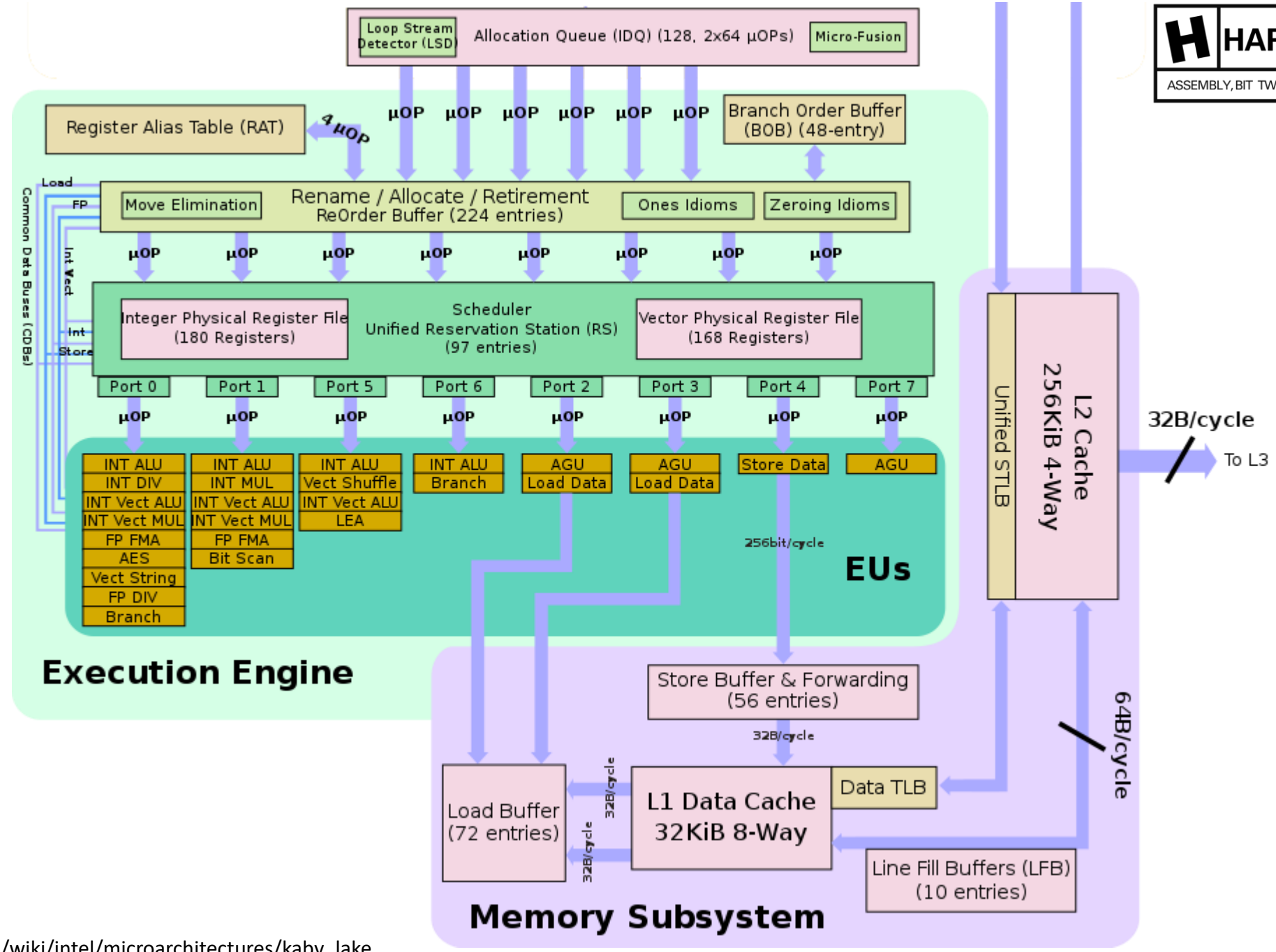
The Frontend (where preparation works happen)
In charge of figure out what we will execute, what data we will need and route it to the proper execution units.

The Backend (where work actually happens)
In charge of execute our instructions and write to memory.
This is the place where magic happens.

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



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



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

Prefetching


```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 32, originalPtr += 32, modifiedPtr += 32)
{
    Sse.Prefetch0(originalPtr + Offset);
    Sse.Prefetch0(modifiedPtr + Offset);

    Vector256<byte> m = Avx.LoadVector256(modifiedPtr);
    Vector256<byte> o = Avx.LoadVector256(originalPtr);

    o = Avx2.Xor(o, m);
    if (!Avx.TestZ(o, o))
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        Avx.Store((writePtr + writePtrOffset), m);
        writePtrOffset++;
    }
    // […] – Removed for presentation purposes
}
```

Prefetching

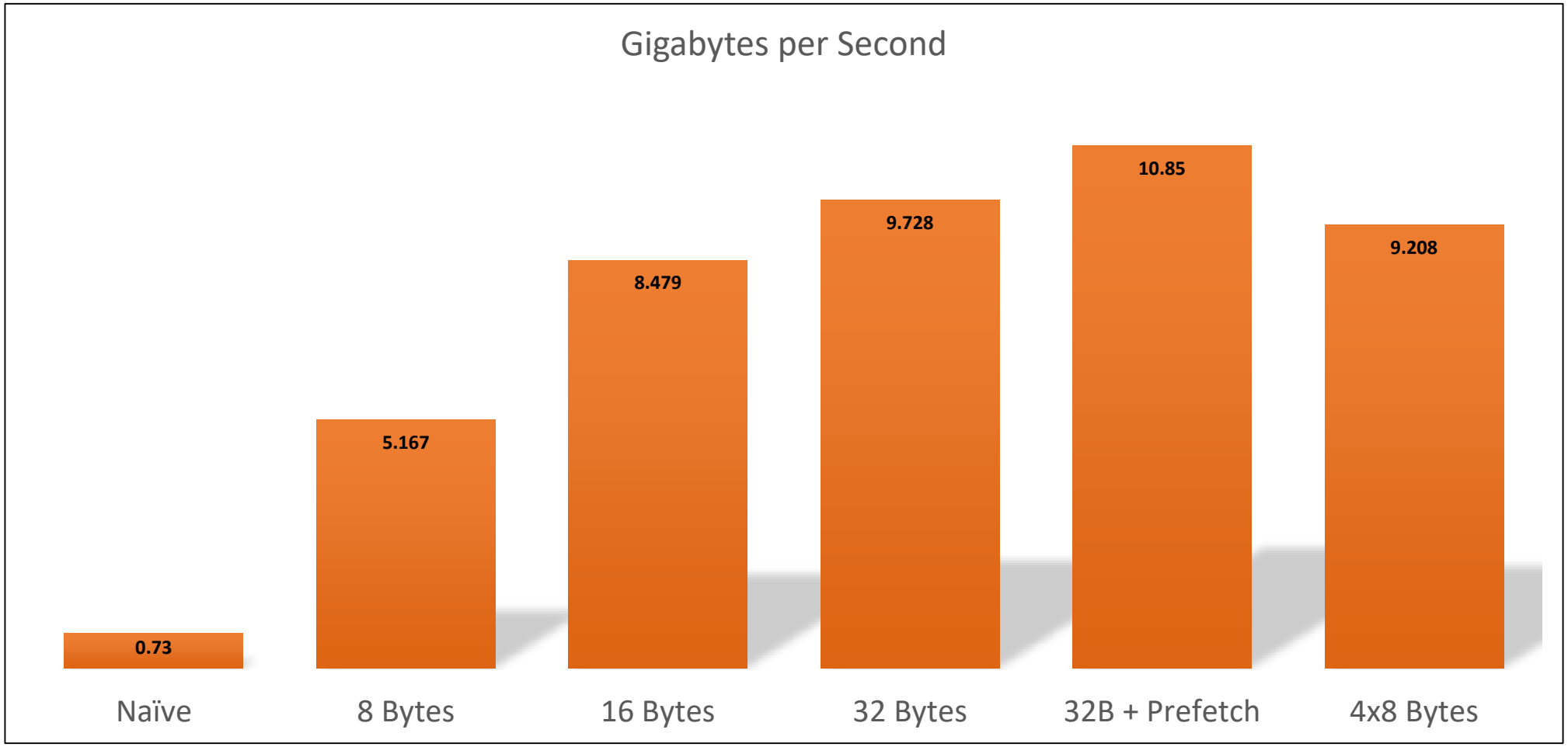
	L1	L2	L3	Non Temporal
64	1.02	0.98	1	1.02
128	1	0.99	1.01	1.07
256	0.93	0.98	1.02	1.07
512	0.91	0.97	0.97	1.03
1024	0.89	0.94	0.95	0.97
2048	0.88	0.91	0.92	1.02
4096	0.89	0.91	0.92	0.98
8192	0.89	0.92	0.91	1.04
16384	0.92	0.91	0.92	1.24

Before Prefetching

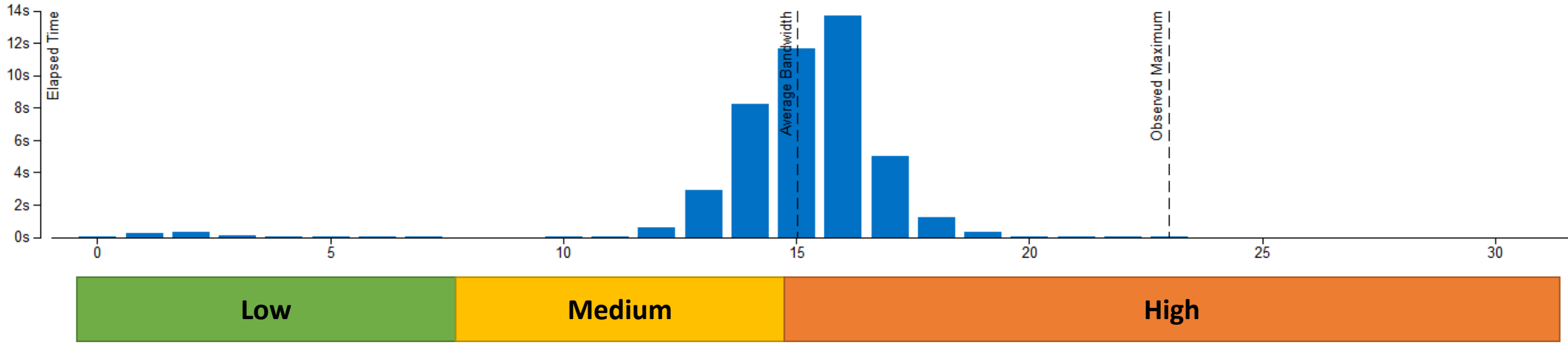
CPU Time:	33.550s	
Memory Bound:	**51.9%**	**of Pipeline Slots**
L1 Bound:	1.2%	of Clockticks
L2 Bound:	12.6%	of Clockticks
L3 Bound:	7.7%	of Clockticks
DRAM Bound:	**27.4%**	**of Clockticks**
DRAM Bandwidth Bound:	89.7%	of Elapsed Time
Stores:	648,000,000	
LLC Miss Count:	901,680,000	
Total Thread Count:	16	
Paused Time:	0s	

Optimal Prefetching

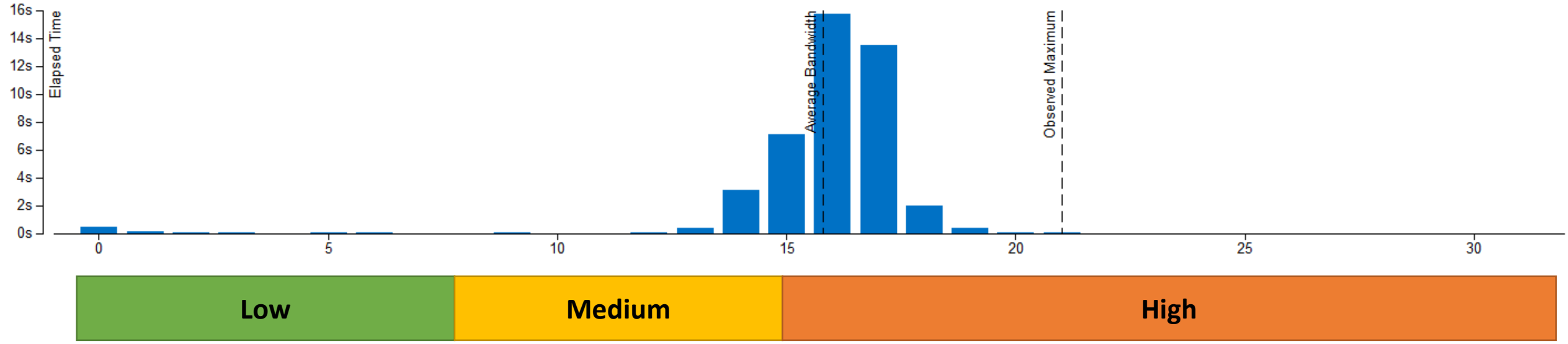
CPU Time:	31.695s	
Memory Bound:	**24.8%**	**of Pipeline Slots**
L1 Bound:	3.6%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	2.4%	of Clockticks
DRAM Bound:	**4.0%**	**of Clockticks**
DRAM Bandwidth Bound:	96.8%	of Elapsed Time
Stores:	374,400,000	
LLC Miss Count:	1,200,000	
Total Thread Count:	16	
Paused Time:	0s	


Gigabytes per Second
0.73
5.167
8.479
9.728
10.85
9.208
Naïve
8 Bytes
16 Bytes
32 Bytes
32B + Prefetch
4x8 Bytes

Bandwidth Utilization of (better) Naïve AVX



Bandwidth Utilization of Optimal Prefetch AVX



Can we do better than this?

Optimizing the code layout

- We have to focus on how instructions are arranged
  - Is our instruction sequence predictable?
  - Are we using properly the instruction cache?
- Frontend decoding
  - Are our instructions ‘easy’ to decode?
  - Are our loops LSD optimized?
- Type of instructions we use.
  - Instruction Latency
  - Instruction Reciprocal Throughput
  - Physical Port
- Data dependencies

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 32, originalPtr += 32, modifiedPtr += 32)
{
    Sse.Prefetch0(originalPtr + 1024);
    Sse.Prefetch0(modifiedPtr + 1024);

    Vector256<byte> m = Avx.LoadVector256(modifiedPtr);
    Vector256<byte> o = Avx.LoadVector256(originalPtr);

    o = Avx2.Xor(o, m);
    if (!Avx.TestZ(o, o))
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        Avx.Store((writePtr + writePtrOffset), m);
        writePtrOffset++;
    }
    […] Code not interesting yet!!!
}
```

Strides over the words

Load and Compare

Write the modified data on the buffer

The actual code that gets generated


```
:loop
lea rbx,[r11+800h]
prefetcht0 [rbx] # Prefetch the original
lea rbx,[r8+800h]
prefetcht0 [rbx] # Prefetch the modified

vmovdqu ymm0,ymmword ptr [r8] # r8 is the modified
vmovdqu ymm1,ymmword ptr [r11] # r11 is the original
vpxor ymm1,ymm1,ymm0
xor ebx,ebx # zero out ebx
vptest ymm1,ymm1
sete bl # set the lowest byte if Equal flag is set
test ebx,ebx # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbx,[rax+r10]
vmovdqu ymmword ptr [rbx],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi

:loopend
# The loop header will increment both pointers
add rdi,20h
add r11,20h
add r8,20h
cmp rdi,r9
jl :loop
```

The actual code that gets generated


```
:loop
lea rbx,[r11+800h]
prefetcht0 [rbx] # Prefetch the original
lea rbx,[r8+800h]
prefetcht0 [rbx] # Prefetch the modified

vmovdqu ymm0,ymmword ptr [r8] # r8 is the modified
vmovdqu ymm1,ymmword ptr [r11] # r11 is the original
vpxor ymm1,ymm1,ymm0
xor ebx,ebx # zero out ebx
vptest ymm1,ymm1
sete bl # set the lowest byte if Equal flag is set
test ebx,ebx # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbx,[rax+r10]
vmovdqu ymmword ptr [rbx],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi

:loopend
# The loop header will increment both pointers
add rdi,20h
add r11,20h
add r8,20h
cmp rdi,r9
jl :loop
```

The actual code that gets generated


```
:loop
lea rbx,[r11+800h]
prefetcht0 [rbx] # Prefetch the original
lea rbx,[r8+800h]
prefetcht0 [rbx] # Prefetch the modified

vmovdqu ymm0,ymmword ptr [r8] # r8 is the modified
vmovdqu ymm1,ymmword ptr [r11] # r11 is the original
vpxor ymm1,ymm1,ymm0
xor ebx,ebx # zero out ebx
vptest ymm1,ymm1
sete bl # set the lowest byte if Equal flag is set
test ebx,ebx # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbx,[rax+r10]
vmovdqu ymmword ptr [rbx],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi

:loopend
# The loop header will increment both pointers
add rdi,20h
add r11,20h
add r8,20h
cmp rdi,r9
jl :loop
```

The actual code that gets generated


```
:loop
lea rbx,[r11+800h]
prefetcht0 [rbx] # Prefetch the original
lea rbx,[r8+800h]
prefetcht0 [rbx] # Prefetch the modified

vmovdqu ymm0,ymmword ptr [r8] # r8 is the modified
vmovdqu ymm1,ymmword ptr [r11] # r11 is the original
vpxor ymm1,ymm1,ymm0
xor ebx,ebx # zero out ebx
vptest ymm1,ymm1
sete bl # set the lowest byte if Equal flag is set
test ebx,ebx # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbx,[rax+r10]
vmovdqu ymmword ptr [rbx],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi

:loopend
# The loop header will increment both pointers
add rdi,20h
add r11,20h
add r8,20h
cmp rdi,r9
jl :loop
```

The actual code that gets generated


```
:loop
lea rbx,[r11+800h]
prefetcht0 [rbx] # Prefetch the original
lea rbx,[r8+800h]
prefetcht0 [rbx] # Prefetch the modified

vmovdqu ymm0,ymmword ptr [r8] # r8 is the modified
vmovdqu ymm1,ymmword ptr [r11] # r11 is the original
vpxor ymm1,ymm1,ymm0
xor ebx,ebx # zero out ebx
vptest ymm1,ymm1
sete bl # set the lowest byte if Equal flag is set
test ebx,ebx # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbx,[rax+r10]
vmovdqu ymmword ptr [rbx],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi

:loopend
# The loop header will increment both pointers
add rdi,20h
add r11,20h
add r8,20h
cmp rdi,r9
jl :loop
```

The actual code that gets generated


```
:loop
lea rbx,[r11+800h]
prefetcht0 [rbx] # Prefetch the original
lea rbx,[r8+800h]
prefetcht0 [rbx] # Prefetch the modified

vmovdqu ymm0,ymmword ptr [r8] # r8 is the modified
vmovdqu ymm1,ymmword ptr [r11] # r11 is the original
vpxor ymm1,ymm1,ymm0
xor ebx,ebx # zero out ebx
vptest ymm1,ymm1
sete bl # set the lowest byte if Equal flag is set
test ebx,ebx # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbx,[rax+r10]
vmovdqu ymmword ptr [rbx],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi

:loopend
# The loop header will increment both pointers
add rdi,20h
add r11,20h
add r8,20h
cmp rdi,r9
jl :loop
```

Main loop when Unchanged

HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

```
:loop
lea rbx,[r11+800h]
prefetcht0 [rbx] # Prefetch the original
lea rbx,[r8+800h]
prefetcht0 [rbx] # Prefetch the modified

vmovdqu ymm0,ymmword ptr [r8] # r8 is the modified
vmovdqu ymm1,ymmword ptr [r11] # r11 is the original
vpxor ymm1,ymm1,ymm0
xor ebx,ebx # zero out ebx
vptest ymm1,ymm1
sete bl # set the lowest byte if Equal flag is set
test ebx,ebx # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbx,[rax+r10]
vmovdqu ymmword ptr [rbx],ymm0
add r10,20h
jmp :loopend
```

```
:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi
```

```
:loopend
# The loop header will increment both pointers
add rdi,20h
add r11,20h
add r8,20h
cmp rdi,r9
jl :loop
```

Main loop when Unchanged

HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

```
:loop
lea rbx,[r11+800h]
prefetcht0 [rbx] # Prefetch the original
lea rbx,[r8+800h]
prefetcht0 [rbx] # Prefetch the modified

vmovdqu ymm0,ymmword ptr [r8] # r8 is the modified
vmovdqu ymm1,ymmword ptr [r11] # r11 is the original
vpxor ymm1,ymm1,ymm0
xor ebx,ebx # zero out ebx
vptest ymm1,ymm1
sete bl # set the lowest byte if Equal flag is set
test ebx,ebx # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbx,[rax+r10]
vmovdqu ymmword ptr [rbx],ymm0
add r10,20h
jmp :loopend
```

```
:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi
```

```
:loopend
# The loop header will increment both pointers
add rdi,20h
add r11,20h
add r8,20h
cmp rdi,r9
jl :loop
```

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 32, originalPtr += 32, modifiedPtr += 32)
{
    Sse.Prefetch0(originalPtr + 1024);
    Sse.Prefetch0(modifiedPtr + 1024);

    Vector256<byte> m = Avx.LoadVector256(modifiedPtr);
    Vector256<byte> o = Avx.LoadVector256(originalPtr);

    o = Avx2.Xor(o, m);
    if (!Avx.TestZ(o, o))
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        Avx.Store((writePtr + writePtrOffset), m);
        writePtrOffset++;
    }
    […] – Removed for presentation purposes
}
```

Strides over the words

Load and Compare

```
byte* ptr = (byte*)originalBuffer; // These are void* pointers
long offset = (byte*)modifiedBuffer - (byte*)originalBuffer;

bool started = false;
for (byte* end = ptr + size; ptr < end; ptr += 32)
{
    Sse.Prefetch0(ptr + 1024);
    Sse.Prefetch0(ptr + offset + 1024);

    Vector256<byte> m = Avx.LoadVector256(ptr + offset);
    Vector256<byte> o = Avx.LoadVector256(ptr);

    o = Avx2.Xor(o, m);
    if (!Avx.TestZ(o, o))
    {
        if (started == false)
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        Avx.Store((writePtr + writePtrOffset), m);
        writePtrOffset++;
    }
    […] – Removed for presentation purposes
}
```

Strides over the words

Load and Compare

This trick works better with scalar MOV code though ☺

And we got, minus 1 arithmetic instruction


```
:loop
lea rbp,[r11+400h]
prefetcht0 [rbp] # Prefetch ptr
add rbx,400h
prefetcht0 [rbx] # Prefetch ptr+offset

lea rbx,[r11+r8] # rbx will store the ptr + offset
vmovdqu ymm0,ymmword ptr [rbx]
vmovdqu ymm1,ymmword ptr [r11] # r11 is the ptr
vpxor ymm1,ymm1,ymm0
xor ebp,ebp # zero out ebp
vptest ymm1,ymm1
sete bpl # set the lowest byte if Equal flag is set
test ebp,ebp # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbp,[rax+r10]
vmovdqu ymmword ptr [rbp],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi

:loopend
# The loop header will increment the ptr and counter
add r11,20h
add rdi,20h
add r8,20h
cmp r11,r9 # r9 gets assigned outside the loop.
jb :loop
```

And we got, minus 1 arithmetic instruction


```
:loop
lea rbp,[r11+400h]
prefetcht0 [rbp] # Prefetch ptr
add rbx,400h
prefetcht0 [rbx] # Prefetch ptr+offset

lea rbx,[r11+r8] # rbx will store the ptr + offset
vmovdqu ymm0,ymmword ptr [rbx]
vmovdqu ymm1,ymmword ptr [r11] # r11 is the ptr
vpxor ymm1,ymm1,ymm0
xor ebp,ebp # zero out ebp
vptest ymm1,ymm1
sete bpl # set the lowest byte if Equal flag is set
test ebp,ebp # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbp,[rax+r10]
vmovdqu ymmword ptr [rbp],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi

:loopend
# The loop header will increment the ptr and counter
add r11,20h
add rdi,20h
add r8,20h
cmp r11,r9 # r9 gets assigned outside the loop.
jb :loop
```

Main loop when Unchanged

HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

```
:loop
lea rbp,[r11+400h]
prefetcht0 [rbp] # Prefetch ptr
add rbx,400h
prefetcht0 [rbx] # Prefetch ptr+offset

lea rbx,[r11+r8] # rbx will store the ptr + offset
vmovdqu ymm0,ymmword ptr [rbx]
vmovdqu ymm1,ymmword ptr [r11] # r11 is the ptr
vpxor ymm1,ymm1,ymm0
xor ebp,ebp # zero out ebp
vptest ymm1,ymm1
sete bpl # set the lowest byte if Equal flag is set
test ebp,ebp # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword

# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbp,[rax+r10]
vmovdqu ymmword ptr [rbp],ymm0
add r10,20h
jmp :loopend
```

```
:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi
```

```
:loopend
# The loop header will increment the ptr and counter
add r11,20h
cmp r11,r9 # r9 gets assigned outside the loop.
jb :loop
```

Main loop when Changed


```
:loop
lea rbp,[r11+400h]
prefetcht0 [rbp] # Prefetch ptr
add rbx,400h
prefetcht0 [rbx] # Prefetch ptr+offset

lea rbx,[r11+r8] # rbx will store the ptr + offset
vmovdqu ymm0,ymmword ptr [rbx]
vmovdqu ymm1,ymmword ptr [r11] # r11 is the ptr
vpxor ymm1,ymm1,ymm0
xor ebp,ebp # zero out ebp
vptest ymm1,ymm1
sete bpl # set the lowest byte if Equal flag is set
test ebp,ebp # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword
```

```
# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbp,[rax+r10]
vmovdqu ymmword ptr [rbp],ymm0
add r10,20h
jmp :loopend
```

```
:closeblock
test esi,esi
je :loopend

lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi
```

```
:loopend
# The loop header will increment the ptr and counter
add r11,20h
cmp r11,r9 # r9 gets assigned outside the loop.
jb :loop
```

Can we do better than this?

Hart: Seriously Funny (2010)

Where in the world is the hot code?

HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

```
:loop
lea rbp,[r11+400h]
prefetcht0 [rbp] # Prefetch ptr
add rbx,400h
prefetcht0 [rbx] # Prefetch ptr+offset

lea rbx,[r11+r8] # rbx will store the ptr + offset
vmovdqu ymm0,ymmword ptr [rbx]
vmovdqu ymm1,ymmword ptr [r11] # r11 is the ptr
vpxor ymm1,ymm1,ymm0
xor ebp,ebp # zero out ebp
vptest ymm1,ymm1
sete bpl # set the lowest byte if Equal flag is set
test ebp,ebp # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword
```

```
# We are starting the block
mov rsi,r11
sub rsi,rdx
mov qword ptr [rax],rsi # write start index in buffer
mov esi,1 # started = true
```

```
:storeword
# We will store back to memory
lea rbp,[rax+r10]
vmovdqu ymmword ptr [rbp],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend
```

```
lea rsi,[r10-10h] # We are closing the block
mov qword ptr [rax+8],rsi
add rax,r10
mov r10d,10h
xor esi,esi
```

```
:loopend
# The loop header will increment the ptr and counter
add r11,20h
cmp r11,r9 # r9 gets assigned outside the loop.
jb :loop
```

We want it to look nice …


```
:loop
lea rbp,[r11+400h]
prefetcht0 [rbp] # Prefetch ptr
add rbx,400h
prefetcht0 [rbx] # Prefetch ptr+offset

lea rbx,[r11+r8] # rbx will store the ptr + offset
vmovdqu ymm0,ymmword ptr [rbx]
vmovdqu ymm1,ymmword ptr [r11] # r11 is the ptr
vpxor ymm1,ymm1,ymm0
xor ebp,ebp # zero out ebp
vptest ymm1,ymm1
sete bpl # set the lowest byte if Equal flag is set
test ebp,ebp # check for zero and set ZF
jne :closeblock # will jump if ZF = 1

test esi,esi
jne :storeword
```

```
:storeword
# We will store back to memory
lea rbp,[rax+r10]
vmovdqu ymmword ptr [rbp],ymm0
add r10,20h
jmp :loopend

:closeblock
test esi,esi
je :loopend
```

```
:loopend
# The loop header will increment the ptr and counter
add r11,20h
cmp r11,r9 # r9 gets assigned outside the loop.
jb :loop
```

… like this!!!

HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

Address	Sour... Line	Assembly	Clockticks	Instructions Retired	CPI Rate	Locators																
						Front-End B...		Ba. Sp.	Back-End Bound													
									Memory Bound													
									L1 Bound						L2 Bound	L3 Bound				DRAM Bound		Store Bound
						Fro... Late...	Fr. Ba.		DTLB Ove...	Loa... Bloc...	Lock Late...	Split Loads	4K Alia...	FB Full		Con... Acc...	Data Shar...	L3 Late...	SQ Full	Memory Bandwi...	Mem... Latency	
0x7fff1bd4b29d		lea rdi, ptr [r11+0x400]	22,133,160,000	40,227,840,000	0.550	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.1%	1.0%	4.7%	0.7%	0.0%
0x7fff1bd4b2a4		prefetcht0 zmmword ptr [rdi]	95,400,000	10,440,000	9.138	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2a7		lea rdi, ptr [r8+0x400]	116,595,000,0...	167,352,120, ...	0.697	0.4%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.3%	5.2%	28.8%	1.8%	0.0%
0x7fff1bd4b2ae		prefetcht0 zmmword ptr [rdi]	398,160,000	32,760,000	12.154	0.0%	0.0%	0.0%	0.9%	0.0%	0.0%	0.0%	0.0%	2.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2b1		vmovdqu ymm0, ymmword ptr [r8]	133,090,200,0...	184,901,400, ...	0.720	0.3%	0.1%	0.1%	0.2%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.9%	5.8%	31.9%	1.7%	0.0%
0x7fff1bd4b2b6		vmovdqu ymm1, ymmword ptr [r11]	7,777,440,000	4,228,560,000	1.839	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	10.8%	0.0%	0.0%	0.0%	0.3%	0.1%	1.6%	0.2%	0.0%
0x7fff1bd4b2bb		vpxor ymm1, ymm1, ymm0	16,869,600,000	27,939,600,000	0.604	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.1%	0.7%	3.7%	0.3%	0.0%
0x7fff1bd4b2c0		xor edi, edi	1,512,360,000	2,319,480,000	0.652	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.3%	0.1%	0.0%
0x7fff1bd4b2c2		vptest ymm1, ymm1	21,826,800,000	47,770,560,000	0.457	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.1%	1.0%	4.8%	0.5%	0.0%
0x7fff1bd4b2c7		setz dil	11,338,200,000	24,103,080,000	0.470	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.1%	0.6%	2.3%	0.4%	0.0%
0x7fff1bd4b2cb		test edi, edi	1,227,960,000	2,140,920,000	0.574	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.2%	0.1%	0.0%
0x7fff1bd4b2cd		jnz 0x7fff1bd4b2e2 <Block 5>	0	0	0.000	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2cf		**Block 3:**																				
0x7fff1bd4b2cf		lea rdi, ptr [rax+r10*1]	93,240,000	197,280,000	0.473	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2d3		vmovdqu ymmword ptr [rdi], ymm0	2,160,000	2,880,000	0.750	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2d8		add r10, 0x20	230,040,000	179,280,000	1.283	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%
0x7fff1bd4b2dc		test esi, esi	10,080,000	13,320,000	0.757	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2de		jnz 0x7fff1bd4b2e6 <Block 6>	0	0	0.000	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2e0		**Block 4:**																				
0x7fff1bd4b2e0		jmp 0x7fff1bd4b30a <Block 9>	360,000	360,000	1.000	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2e2		**Block 5:**																				
0x7fff1bd4b2e2		test esi, esi	22,034,880,000	44,189,280,000	0.499	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	1.0%	4.7%	0.7%	0.0%
0x7fff1bd4b2e4		jnz 0x7fff1bd4b2f5 <Block 8>	0	0	0.000	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2e6		**Block 6:**																				
0x7fff1bd4b2e6		add r11, 0x20	201,240,000	180,360,000	1.116	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2ea		add r8, 0x20	10,839,240,000	21,555,360,000	0.503	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.0%	0.5%	2.4%	0.3%	0.0%
0x7fff1bd4b2ee		cmp r11, r9	58,320,000	27,360,000	2.132	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2f1		jb 0x7fff1bd4b29d <Block 2>	360,000	0		0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

… like this!!!

HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

Address	Sour... Line	Assembly	Clockticks	Instructions Retired	CPI Rate	Locators																
						Front-End B...		Ba. Sp.	Back-End Bound													
									Memory Bound													
									L1 Bound						L2 Bound	L3 Bound				DRAM Bound		Store Bound
						Fro... Late...	Fr. Ba.		DTLB Ove...	Loa... Bloc...	Lock Late...	Split Loads	4K Alia...	FB Full		Con... Acc...	Data Shar...	L3 Late...	SQ Full	Memory Bandwi...	Mem... Latency	
0x7fff1bd4b29d		lea rdi, ptr [r11+0x400]	22,133,160,000	40,227,840,000	0.550	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.1%	1.0%	4.7%	0.7%	0.0%
0x7fff1bd4b2a4		prefetcht0 zmmword ptr [rdi]	95,400,000	10,440,000	9.138	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2a7		lea rdi, ptr [r8+0x400]	116,595,000,0...	167,352,120, ...	0.697	0.4%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.3%	5.2%	28.8%	1.8%	0.0%
0x7fff1bd4b2ae		prefetcht0 zmmword ptr [rdi]	398,160,000	32,760,000	12.154	0.0%	0.0%	0.0%	0.9%	0.0%	0.0%	0.0%	0.0%	2.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2b1		vmovdqu ymm0, ymmword ptr [r8]	133,090,200,0...	184,901,400, ...	0.720	0.3%	0.1%	0.1%	0.2%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.9%	5.8%	31.9%	1.7%	0.0%
0x7fff1bd4b2b6		vmovdqu ymm1, ymmword ptr [r11]	7,777,440,000	4,228,560,000	1.839	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	10.8%	0.0%	0.0%	0.0%	0.3%	0.1%	1.6%	0.2%	0.0%
0x7fff1bd4b2bb		vpxor ymm1, ymm1, ymm0	16,869,600,000	27,939,600,000	0.604	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.1%	0.7%	3.7%	0.3%	0.0%

```
vmovdqu ymm0,ymmword ptr [rbx]
```

L1 Bound. FB Queue Full: 100%
DRAM Bound
- Bandwidth: 31.9%
- Latency: 1.7%

… like this!!!

HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

Address	Sour... Line	Assembly	Clockticks	Instructions Retired	CPI Rate	Locators																
						Front-End B...		Back-End Bound														
									Memory Bound													
									L1 Bound						L2 Bound	L3 Bound				DRAM Bound		
						Fro... Late...	Fr. Ba.	Ba. Sp.	DTLB Ove...	Loa... Bloc...	Lock Late...	Split Loads	4K Alia...	FB Full		Con... Acc...	Data Shar...	L3 Late...	SQ Full	Memory Bandwi...	Mem... Latency	Store Bound
0x7fff1bd4b29d		lea rdi, ptr [r11+0x400]	22,133,160,000	40,227,840,000	0.550	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.1%	1.0%	4.7%	0.7%	0.0%
0x7fff1bd4b2a4		prefetcht0 zmmword ptr [rdi]	95,400,000	10,440,000	9.138	0.0%	0.0%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2a7		lea rdi, ptr [r8+0x400]	116,595,000,0...	167,352,120, ...	0.697	0.4%	0.0%	0.3%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.3%	5.2%	28.8%	1.8%	0.0%
0x7fff1bd4b2ae		prefetcht0 zmmword ptr [rdi]	398,160,000	32,760,000	12.154	0.0%	0.0%	0.0%	0.9%	0.0%	0.0%	0.0%	0.0%	2.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
0x7fff1bd4b2b1		vmovdqu ymm0, ymmword ptr [r8]	133,090,200,0...	184,901,400, ...	0.720	0.3%	0.1%	0.1%	0.2%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.9%	5.8%	31.9%	1.7%	0.0%
0x7fff1bd4b2b6		vmovdqu ymm1, ymmword ptr [r11]	7,777,440,000	4,228,560,000	1.839	0.0%	0.0%	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	10.8%	0.0%	0.0%	0.0%	0.3%	0.1%	1.6%	0.2%	0.0%
0x7fff1bd4b2bb		vpxor ymm1, ymm1, ymm0	16,869,600,000	27,939,600,000	0.604	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%	0.0%	0.1%	0.7%	3.7%	0.3%	0.0%

This metric does a rough estimation of how often L1D Fill Buffer unavailability limited additional L1D miss memory access requests to proceed. The higher the metric value, the deeper the memory hierarchy level the misses are satisfied from. Often it hints on approaching bandwidth limits (to L2 cache, L3 cache or external memory). Avoid adding software prefetches if indeed memory BW limited.

And the code?

```
for (byte* end = originalPtr + size; originalPtr < end; originalPtr += 32, modifiedPtr += 32)
{
    Top:
    // […] Removed for presentation purposes...
    Vector256<byte> m = Avx.LoadVector256(modifiedPtr);
    Vector256<byte> o = Avx.LoadVector256(originalPtr);

    o = Avx2.Xor(o, m);
    if (!Avx.TestZ(o, o))
    {
        Avx.Store((writePtr + writePtrOffset), m);
        writePtrOffset += 32;

        if (!started) goto StartBlock;
    }
    else if (started) goto CloseBlock;

    originalPtr += 32; modifiedPtr += 32;
    if (originalPtr >= end)
        break; // Early break.
    goto Top;

    CloseBlock:
    // […] Removed for presentation purposes...
    continue;

    StartBlock:
    // […] Removed for presentation purposes...
}
```

```
for (byte* end = originalPtr + size; originalPtr < end; originalPtr += 32, modifiedPtr += 32)
{
    Top:
    // […] Removed for presentation purposes...
    Vector256<byte> m = Avx.LoadVector256(modifiedPtr);
    Vector256<byte> o = Avx.LoadVector256(originalPtr);

    o = Avx2.Xor(o, m);
    if (!Avx.TestZ(o, o))
    {
        Avx.Store((writePtr + writePtrOffset), m);
        writePtrOffset += 32;

        if (!started) goto StartBlock;
    }
    else if (started) goto CloseBlock;

    originalPtr += 32; modifiedPtr += 32;
    if (originalPtr >= end)
        break; // Early break.
    goto Top;

    CloseBlock:
    // […] Removed for presentation purposes...
    continue;

    StartBlock:
    // […] Removed for presentation purposes...
}
```

RyuJIT: Allow developers to provide Branch Prediction Information #6024

Open **redknightlois** opened this issue on Jun 28, 2016 · 21 comments

redknightlois commented on Jun 28, 2016

The `__builtin_expect` function provided by GCC comes to my mind here. While I would root for a great profile based collection system, this could be very helpful as an stopgap into that direction.

http://stackoverflow.com/questions/7346929/why-do-we-use-builtin-expect-when-a-straightforward-way-is-to-use-if-else

myungjoo commented on Jun 28, 2016 Contributor

@lemmaa @seanshpark @parjong @chunseoklee We might need to consider this option (something like "likely" and "unlikely" macros in Linux kernel) for ARM optimisation.

https://github.com/dotnet/coreclr/issues/6024

```
byte* originalPtr = (byte*)originalBuffer; // These are void* pointers
byte* modifiedPtr = (byte*)modifiedBuffer;

bool started = false;
for (long i = 0; i < size; i += 32, originalPtr += 32, modifiedPtr += 32)
{
    Vector256<byte> m = Avx.LoadVector256(modifiedPtr);
    Vector256<byte> o = Avx.LoadVector256(originalPtr);

    o = Avx2.Xor(o, m);
    if (!Avx.TestZ(o, o))
    {
        if (Jit.Unlikely(started == false))
        {
            *(long*)(writePtr + 0) = originalPtr - (byte*)originalBuffer;
            started = true;
        }

        Avx.Store((writePtr + writePtrOffset), m);
        writePtrOffset++;
    }
    else if (Jit.Unlikely(started)) // our byte is untouched here.
    {
        *(long*)(writePtr + 8) = writePtrOffset - 16;
        writePtr += writePtrOffset;
        writePtrOffset = 16;

        started = false;
    }
}
```

One final thought…

If someone sais you CAN’T write fast code in C#

You know the answer ☺


H
HARDCORE
ASSEMBLY, BIT TWIDDLING, MICROARCHITECTURE

Want to know more?

Start here!!!

RavenDB 4.0 Source Code: Grep my commits (author: redknightlois) ☺
https://github.com/ravendb/ravendb/search?o=desc&q=author%3Aredknightlois&s=author-date&type=Commits
What Every Programmer Should Know About Memory – Ulrich Drepper
https://www.akkadia.org/drepper/cpumemory.pdf
Going Nowhere Faster – Chandler Carruth [CppCon 2017]
https://www.youtube.com/watch?v=2EWejmkKlxs
Fastware – Andrei Alexandrescu [DotNext 2018]
- if you didn't attend I suggest to watch it when video is available
Meltdown & Spectre Vulnerabilities – Simply Explained
https://www.youtube.com/watch?v=bs0xswK0eZk
Grace Hopper – Nanoseconds
https://www.youtube.com/watch?v=JEpsKnWZrJ8
https://www.youtube.com/watch?v=ZR0ujwlvbkQ (whole lecture – worth it)
Oren Eini's blog – mostly database tech (I write occasionally as guest writer)
https://ayende.com/blog/
BenchmarkDotNet
https://github.com/dotnet/BenchmarkDotNet

Thank you for coming!

I will be at the discussion zone
(I promise)