# Designing for Efficient Cache Usage

Scott McMillan (@ScottMcMillan0)

Pacific++
October 19th 2018

# Introduction

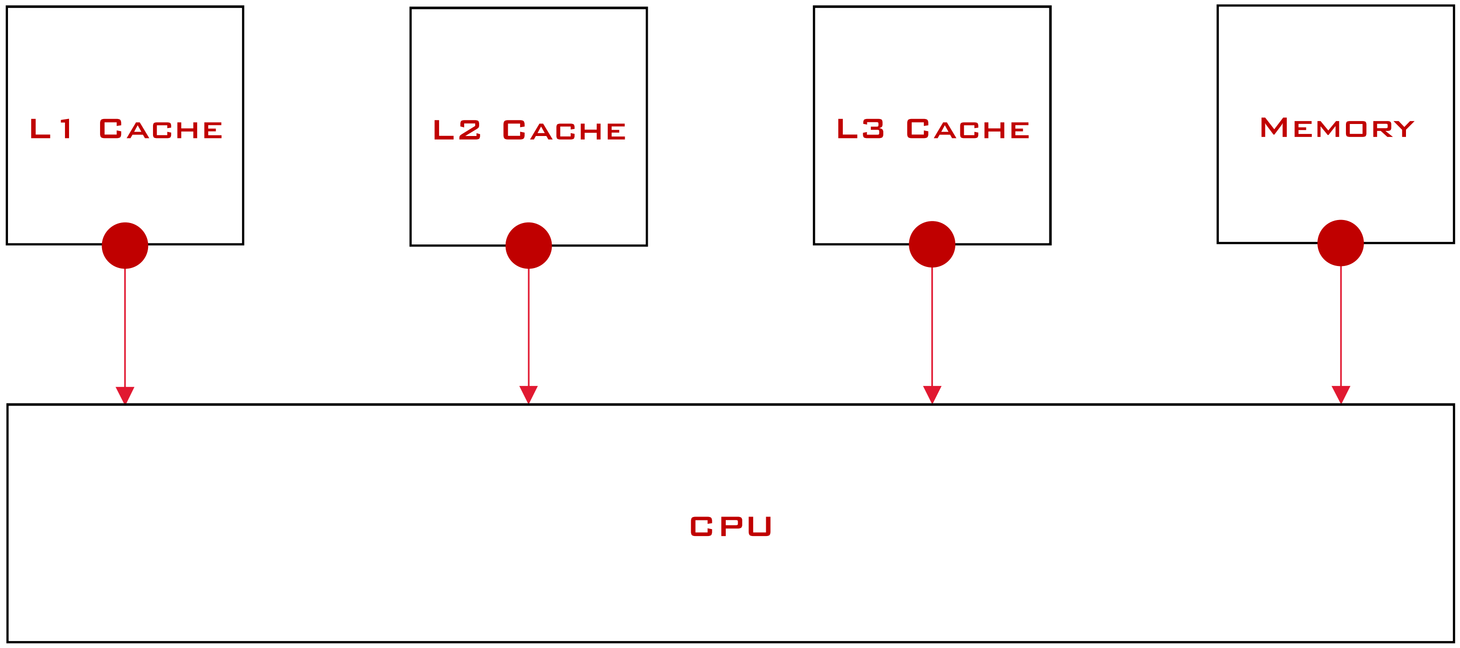
## How much time do games spend waiting for memory?

| | |
|---|---|
| 30.1% | Dota 2 |
| 20.5% | Civilisation VI |
| 32.3% | Fortnite |
| 28.1% | World of Warships |
| 28.8% | World of Tanks |
| 19.0% | Doom 2016 |
| 20.8% | Hitman |
| 24.8% | Crysis |

Optimized Games!

# Introduction

## Cache and memory access latency (based on an Intel i7 8700)



# Abstract

1. Wargaming Sydney Context
2. Sections:
    1. Cache Lines
    2. Hardware Prefetch
    3. Access Locality
    4. Multiple CPU Core Considerations
    5. Write Combined Memory
    6. Address Translation

## Wargaming Sydney Context

- On my particular team:
    - Multiple game titles.
    - Multiple platforms.
    - C++
    - Graphics
    - Optimization

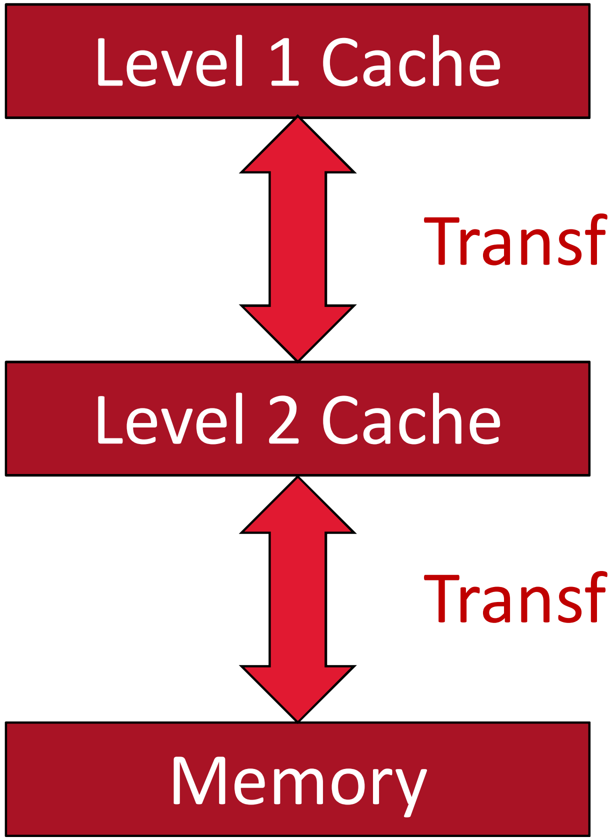
# Wargaming Sydney Context

> *"I think I could cry with this update. I keep seeing people improving from 50fps and up, but this patch has brought me from the pits of 20-30fps all the way up to 60fps minimum."*

- Reddit post after World of Tanks 9.15 release.

# Wargaming Sydney Context

# (1) Cache Lines

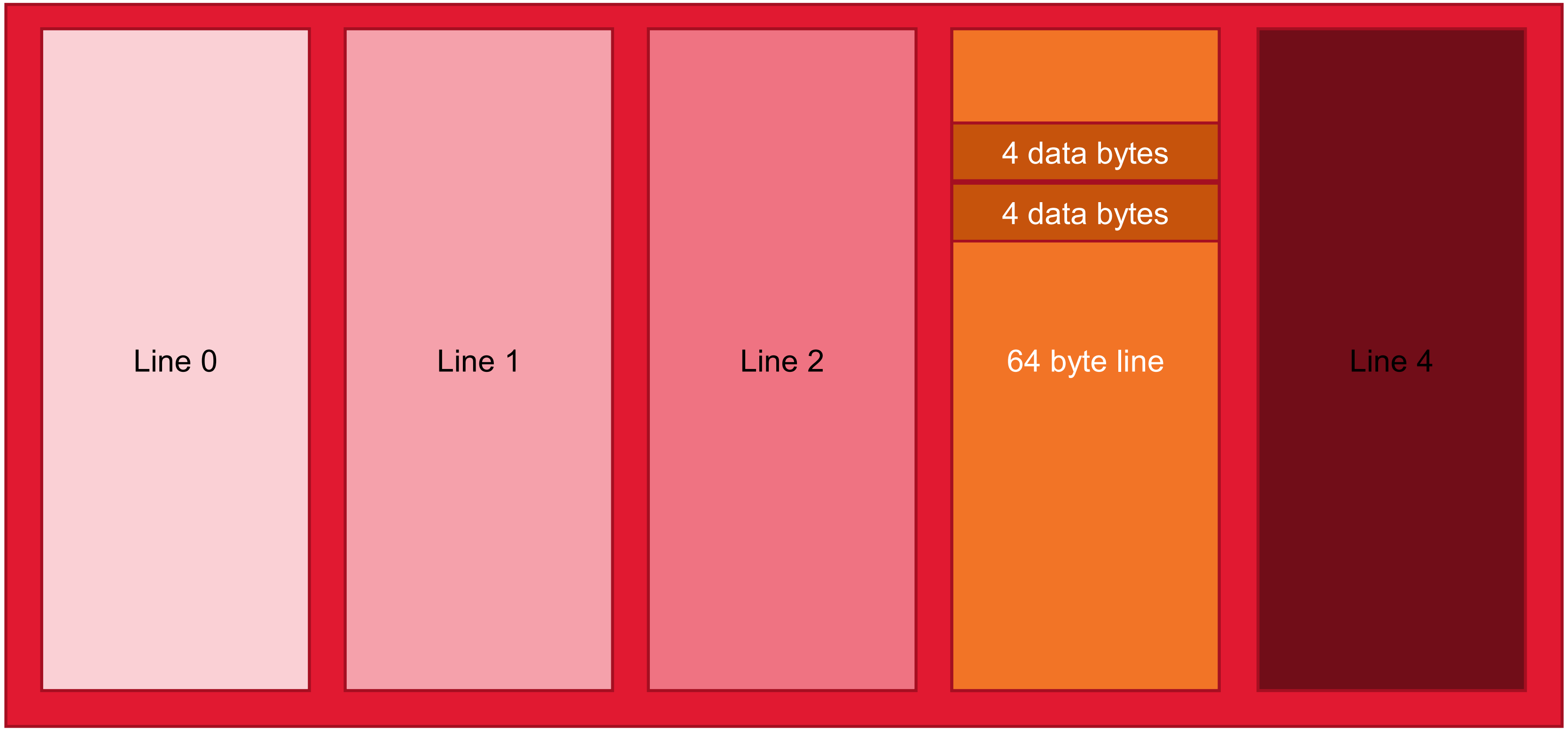


Transfers occur as cache lines.

Transfers occur as cache lines.

# (1) Cache Lines

320 byte cache: 5x 64 byte ‘cache lines’



# (1) Cache Lines

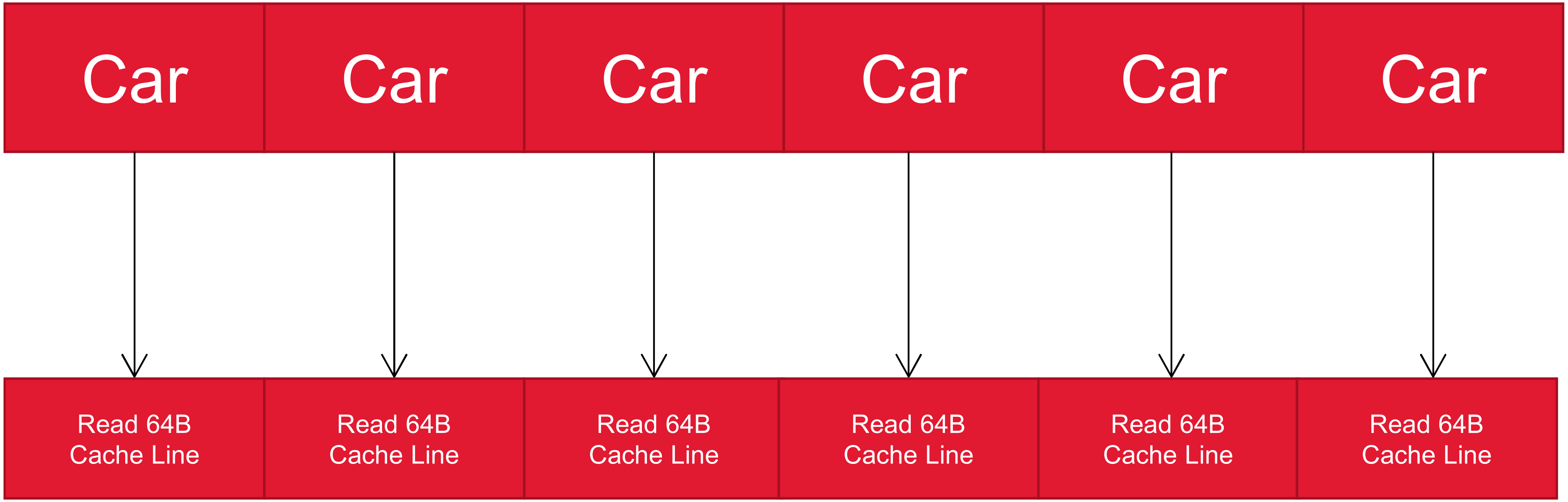
```
struct Car
{
    ....
    bool isTurboBoosting;
    ....
};


std::vector<Car> cars;
```

# (1) Cache Lines

**Iterating to read a Boolean flag from each Car …**

Array of Car objects (each Car is at least the size of a cache line)



1 bit used from each 512 bits fetched from memory …

# (1) Cache Lines

Array of Structures → Structure of Arrays

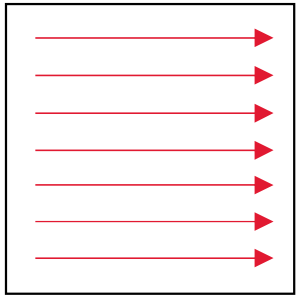
```
struct Car
{
    ….
    bool isTurboBoosting;
    ….
};

std::vector<Car> cars;
```

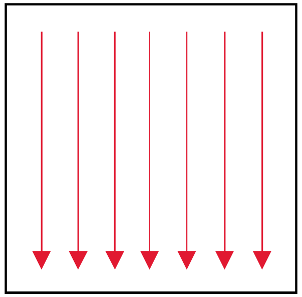
```
struct Cars
{
    … // arrays of other properties
    std::vector<bool> isTurboBoosting;
    … // arrays of other properties
};
```

# (1) Cache Lines

### Horizontal Pass



### Vertical Pass

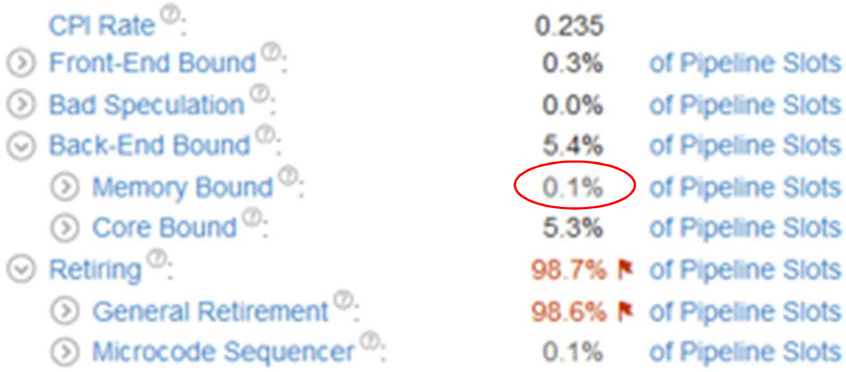


Takes **9.0x** longer
(i7 8700)

# (1) Cache Lines

## Horizontal Pass

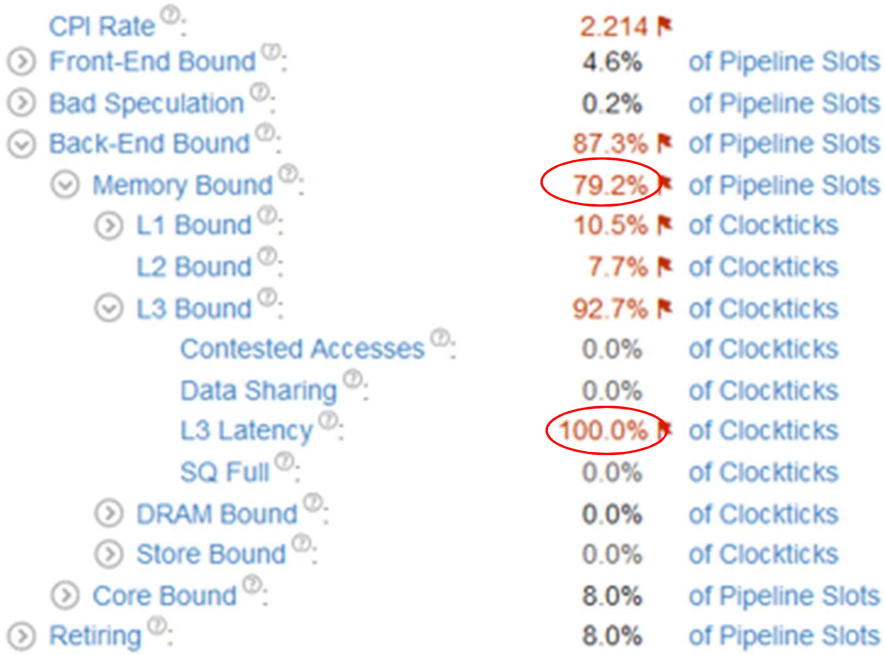
☺



| | | |
|---|---|---|
| CPI Rate: | 0.235 | |
| Front-End Bound: | 0.3% | of Pipeline Slots |
| Bad Speculation: | 0.0% | of Pipeline Slots |
| Back-End Bound: | 5.4% | of Pipeline Slots |
| Memory Bound: | 0.1% | of Pipeline Slots |
| Core Bound: | 5.3% | of Pipeline Slots |
| Retiring: | 98.7% ⚑ | of Pipeline Slots |
| General Retirement: | 98.6% ⚑ | of Pipeline Slots |
| Microcode Sequencer: | 0.1% | of Pipeline Slots |

## Vertical Pass

☹



| | | |
|---|---|---|
| CPI Rate: | 2.214 ⚑ | |
| Front-End Bound: | 4.6% | of Pipeline Slots |
| Bad Speculation: | 0.2% | of Pipeline Slots |
| Back-End Bound: | 87.3% ⚑ | of Pipeline Slots |
| Memory Bound: | 79.2% ⚑ | of Pipeline Slots |
| L1 Bound: | 10.5% ⚑ | of Clockticks |
| L2 Bound: | 7.7% ⚑ | of Clockticks |
| L3 Bound: | 92.7% ⚑ | of Clockticks |
| Contested Accesses: | 0.0% | of Clockticks |
| Data Sharing: | 0.0% | of Clockticks |
| L3 Latency: | 100.0% ⚑ | of Clockticks |
| SQ Full: | 0.0% | of Clockticks |
| DRAM Bound: | 0.0% | of Clockticks |
| Store Bound: | 0.0% | of Clockticks |
| Core Bound: | 8.0% | of Pipeline Slots |
| Retiring: | 8.0% | of Pipeline Slots |

WARGAMING SYDNEY

# (1) Cache Lines

Data Index → Pack Frequently Accessed Data into Handle

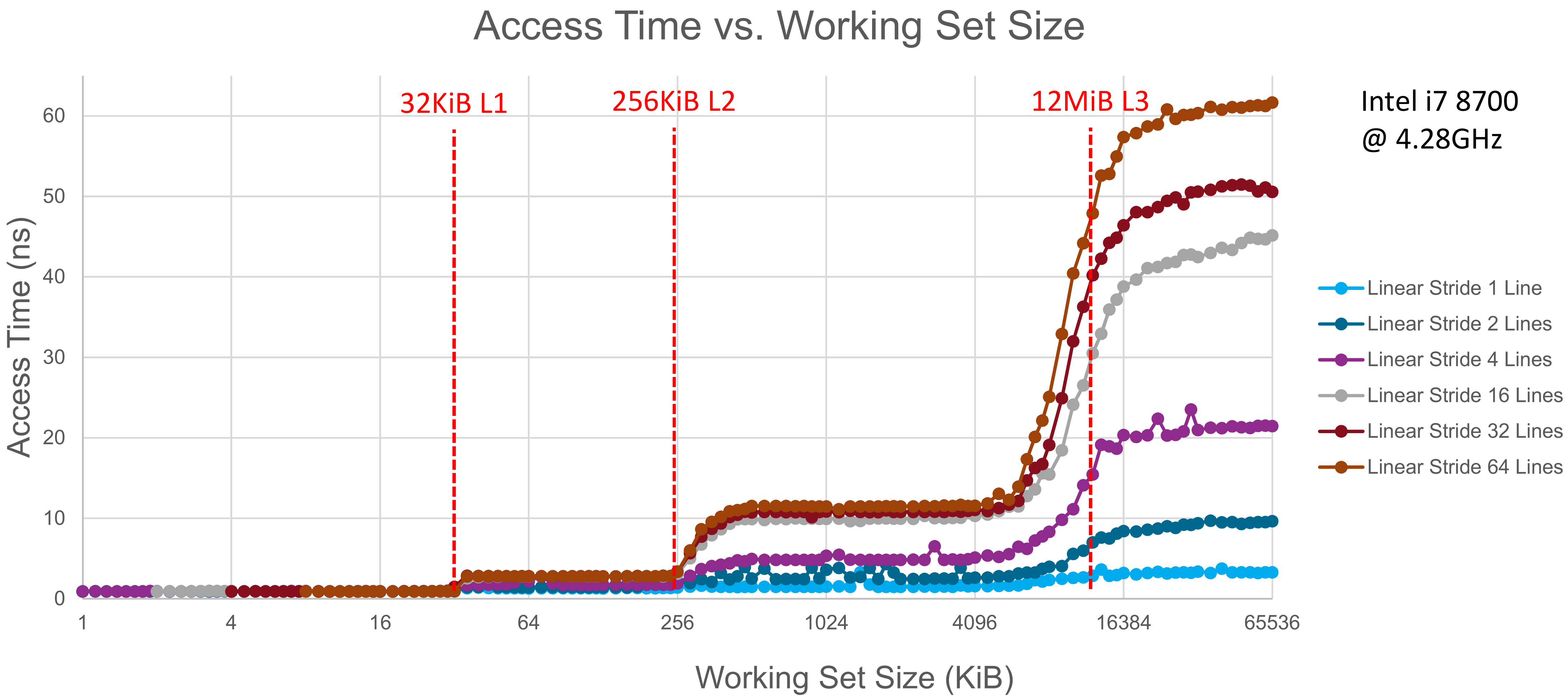
```
struct ParamHandle
{
    int32_t index;
};
```

```
struct ParamHandle
{
    uint16_t typeAndVarIndex;
    uint8_t verificationBits;
    uint8_t resourceIndex;
    uint16_t shaderParamWordOffset;
    uint16_t shaderParamWordSize;
};
```

# (1) Cache Lines

- Key Takeaway:
    - Transfers occur as cache lines.

# (2) Hardware Prefetching



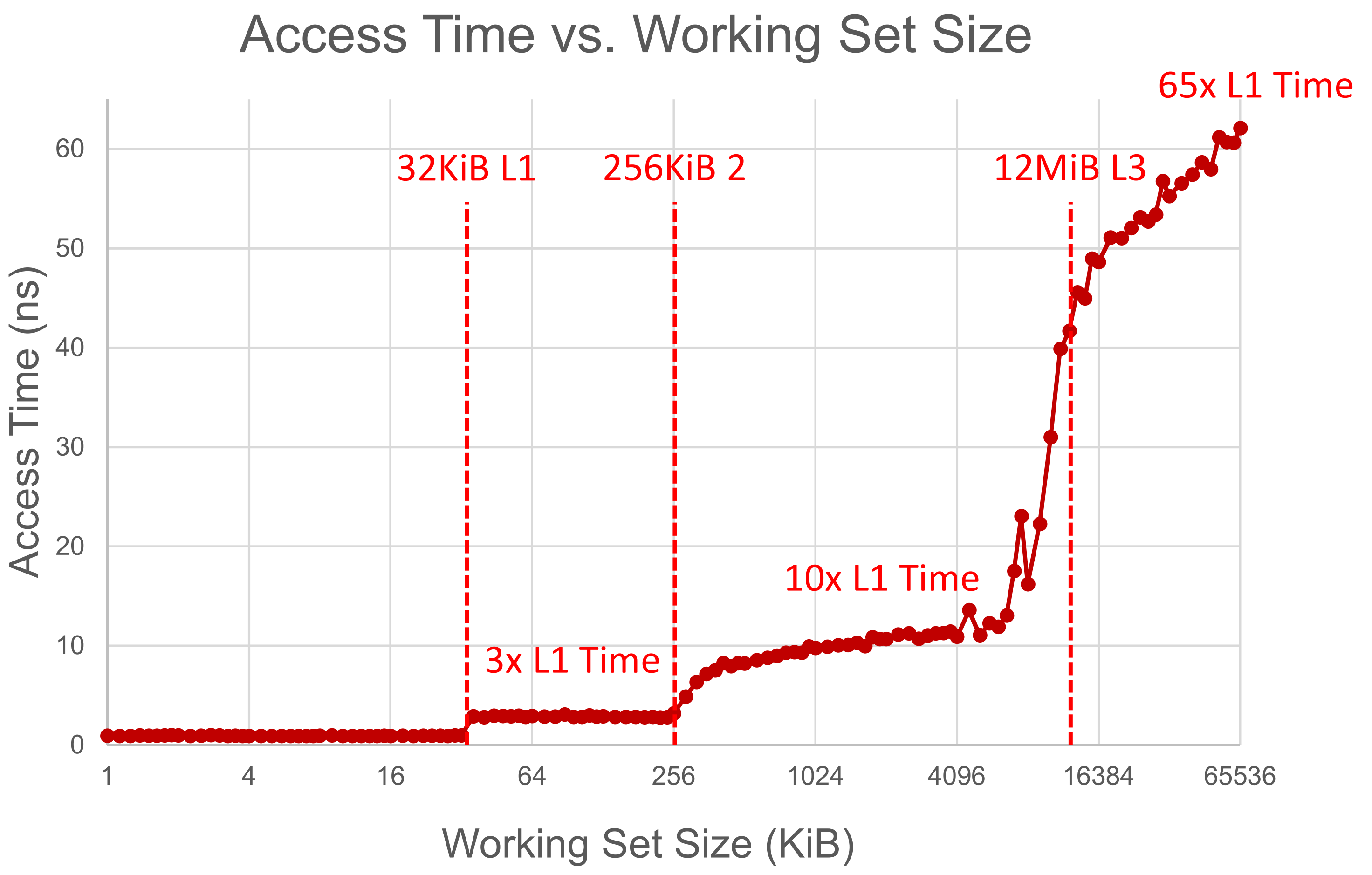
# (2) Hardware Prefetching

- Key Takeaway:
  - Predictable access patterns are faster.
  - We want sequential locality.

# (3) Access Locality

- Cache locality:
  - Spatial
  - Temporal

# (3) Access Locality



Intel i7 8700
@ 4.28GHz

# (3) Access Locality

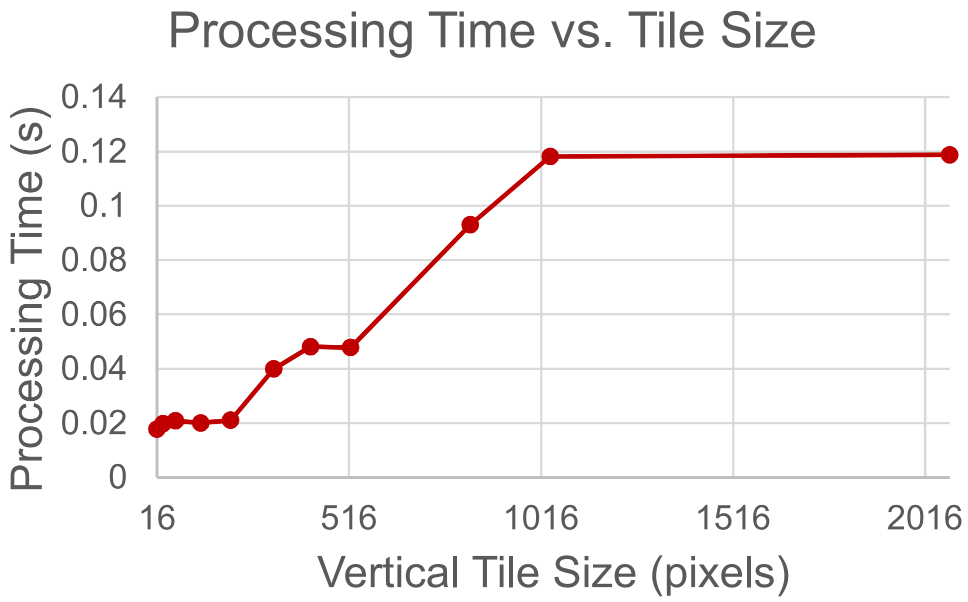
```
for every x
{
    for every y
    {
        …
    }
}
```

→

```
for each x tile
{
    for each y tile
    {
        for every x in tile
        {
            for every y in tile
            {
                …
            }
        }
    }
}
```

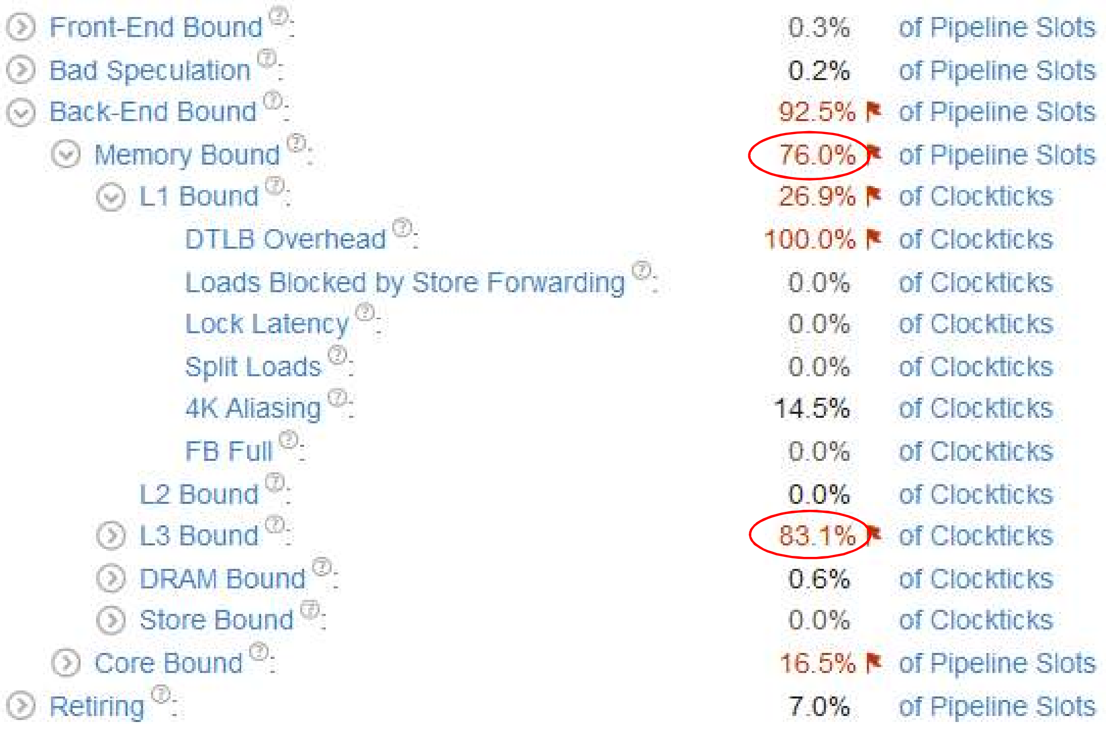
# (3) Access Locality

- Tiling the vertical pass:
  - 64x2080: no improvement
  - 64x416: 2.5x performance
  - 64x16: 6.7x performance



# (3) Access Locality

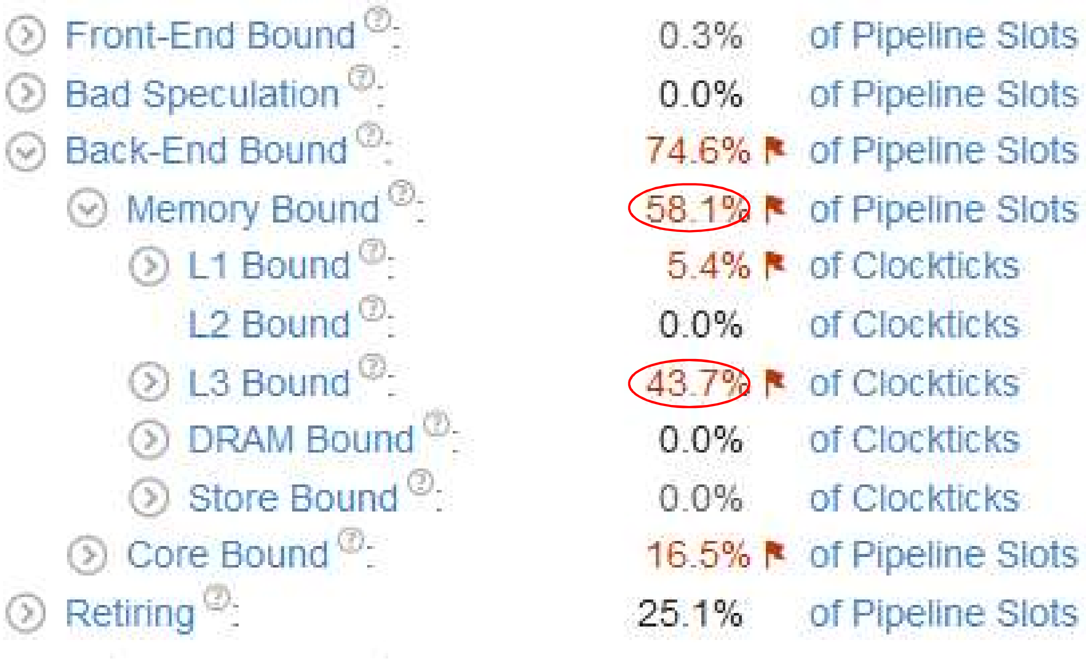
64x2080 tiles. Cycles per instruction: 2.213



| | | |
|---|---|---|
| Front-End Bound: | 0.3% | of Pipeline Slots |
| Bad Speculation: | 0.2% | of Pipeline Slots |
| Back-End Bound: | 92.5% ⚑ | of Pipeline Slots |
| Memory Bound: | 76.0% ⚑ | of Pipeline Slots |
| L1 Bound: | 26.9% ⚑ | of Clockticks |
| DTLB Overhead: | 100.0% ⚑ | of Clockticks |
| Loads Blocked by Store Forwarding: | 0.0% | of Clockticks |
| Lock Latency: | 0.0% | of Clockticks |
| Split Loads: | 0.0% | of Clockticks |
| 4K Aliasing: | 14.5% | of Clockticks |
| FB Full: | 0.0% | of Clockticks |
| L2 Bound: | 0.0% | of Clockticks |
| L3 Bound: | 83.1% ⚑ | of Clockticks |
| DRAM Bound: | 0.6% | of Clockticks |
| Store Bound: | 0.0% | of Clockticks |
| Core Bound: | 16.5% ⚑ | of Pipeline Slots |
| Retiring: | 7.0% | of Pipeline Slots |

# (3) Access Locality

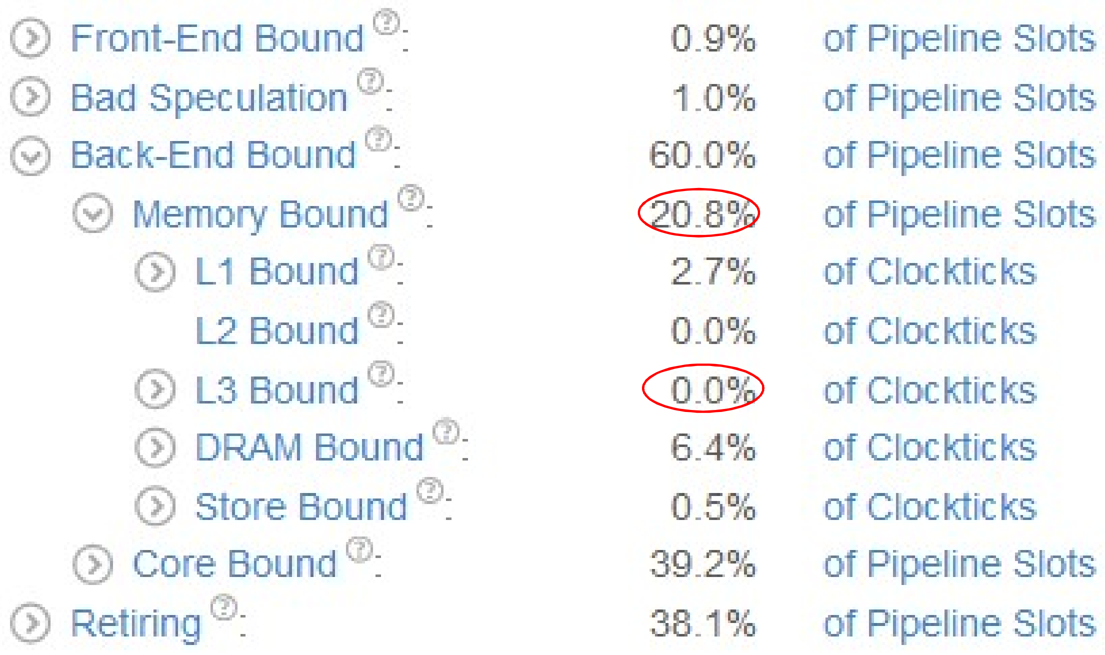
64x416 tiles. Cycles per instruction: 0.874



```
Front-End Bound:           0.3%     of Pipeline Slots
Bad Speculation:           0.0%     of Pipeline Slots
Back-End Bound:           74.6% ⚑  of Pipeline Slots
    Memory Bound:         58.1% ⚑  of Pipeline Slots
        L1 Bound:          5.4% ⚑  of Clockticks
        L2 Bound:          0.0%     of Clockticks
        L3 Bound:         43.7% ⚑  of Clockticks
        DRAM Bound:        0.0%     of Clockticks
        Store Bound:       0.0%     of Clockticks
    Core Bound:           16.5% ⚑  of Pipeline Slots
Retiring:                 25.1%     of Pipeline Slots
```

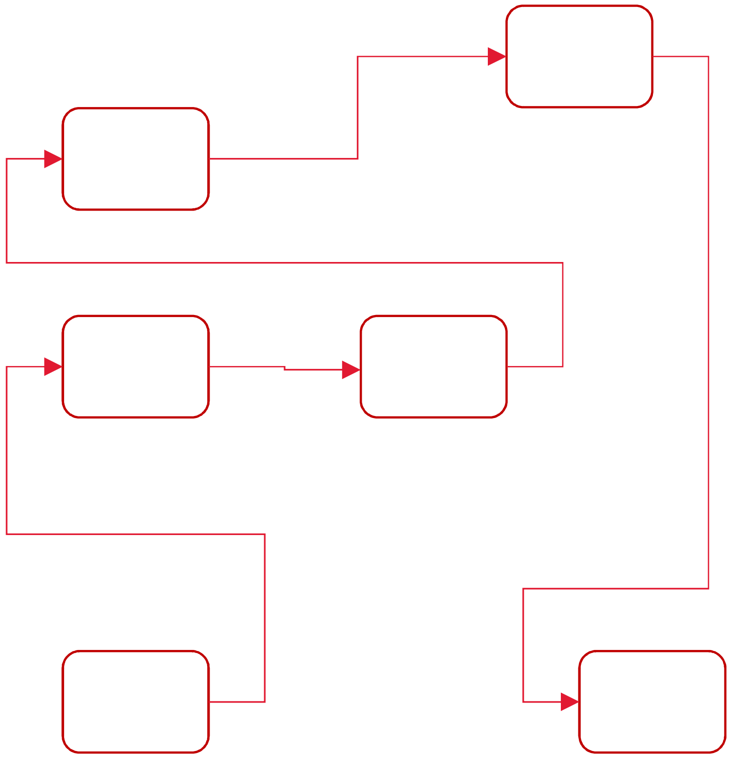
# (3) Access Locality

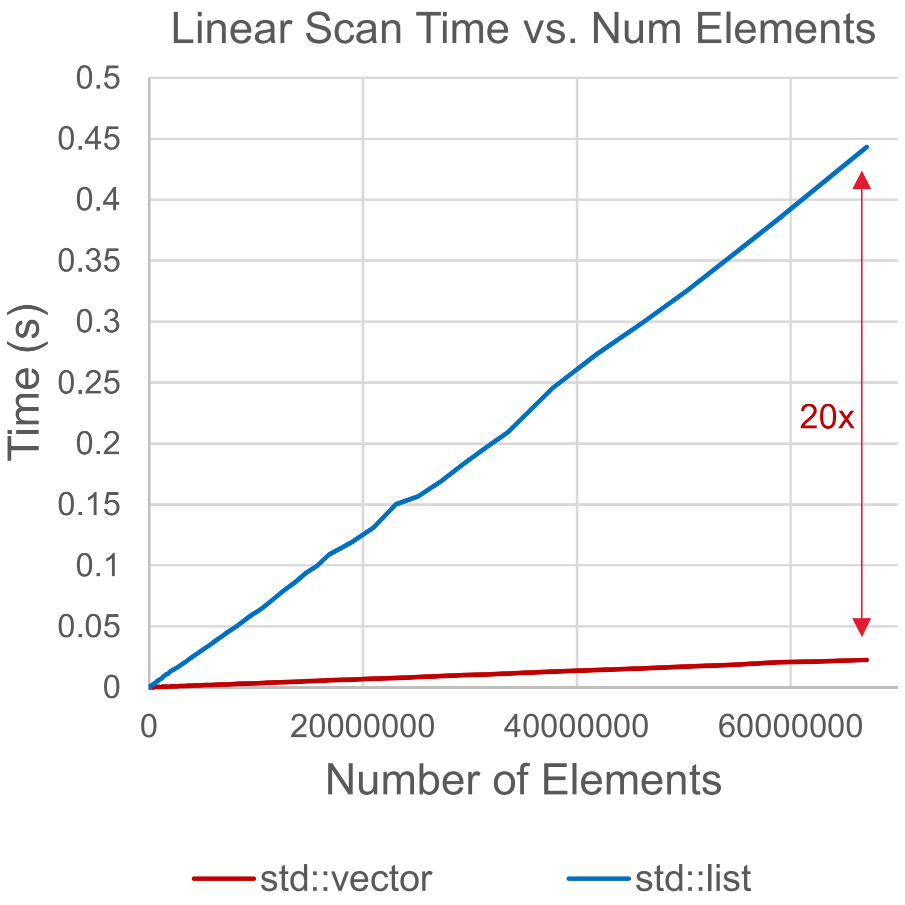
64x16 tiles. Cycles per instruction: 0.293



| | | |
|---|---|---|
| Front-End Bound: | 0.9% | of Pipeline Slots |
| Bad Speculation: | 1.0% | of Pipeline Slots |
| Back-End Bound: | 60.0% | of Pipeline Slots |
| Memory Bound: | 20.8% | of Pipeline Slots |
| L1 Bound: | 2.7% | of Clockticks |
| L2 Bound: | 0.0% | of Clockticks |
| L3 Bound: | 0.0% | of Clockticks |
| DRAM Bound: | 6.4% | of Clockticks |
| Store Bound: | 0.5% | of Clockticks |
| Core Bound: | 39.2% | of Pipeline Slots |
| Retiring: | 38.1% | of Pipeline Slots |

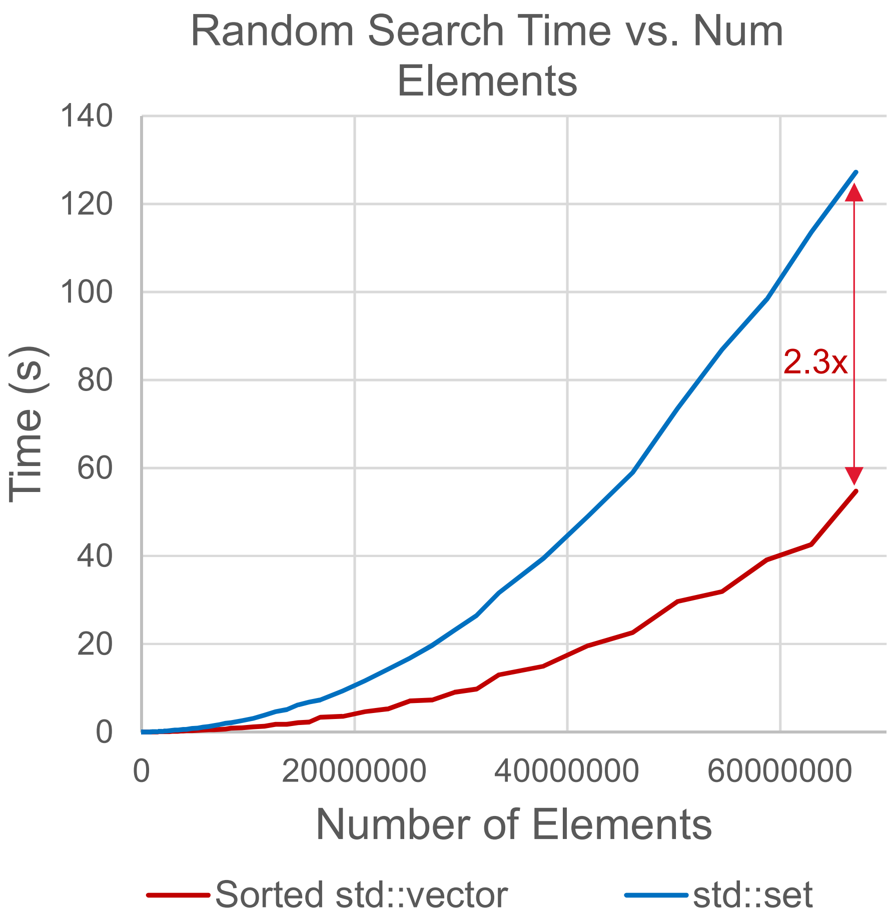
## (3) Access Locality

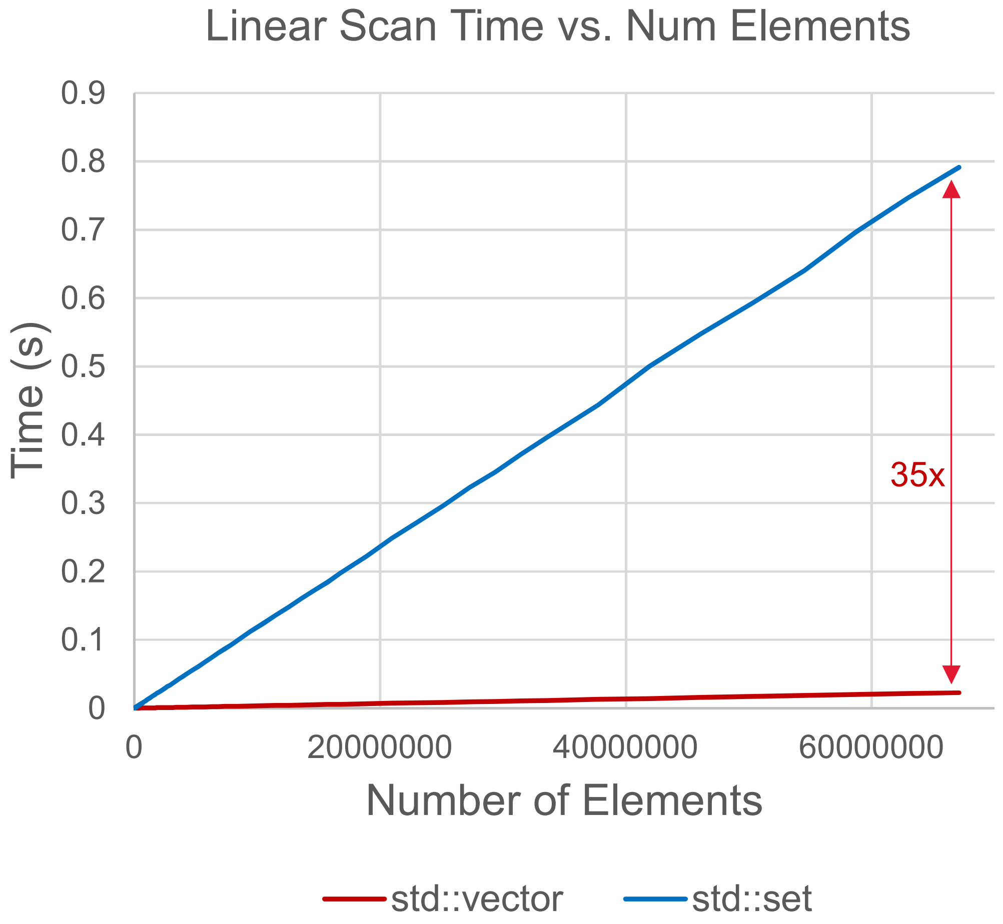




Tests run on MSVC++ 2017 64b, i7 8700

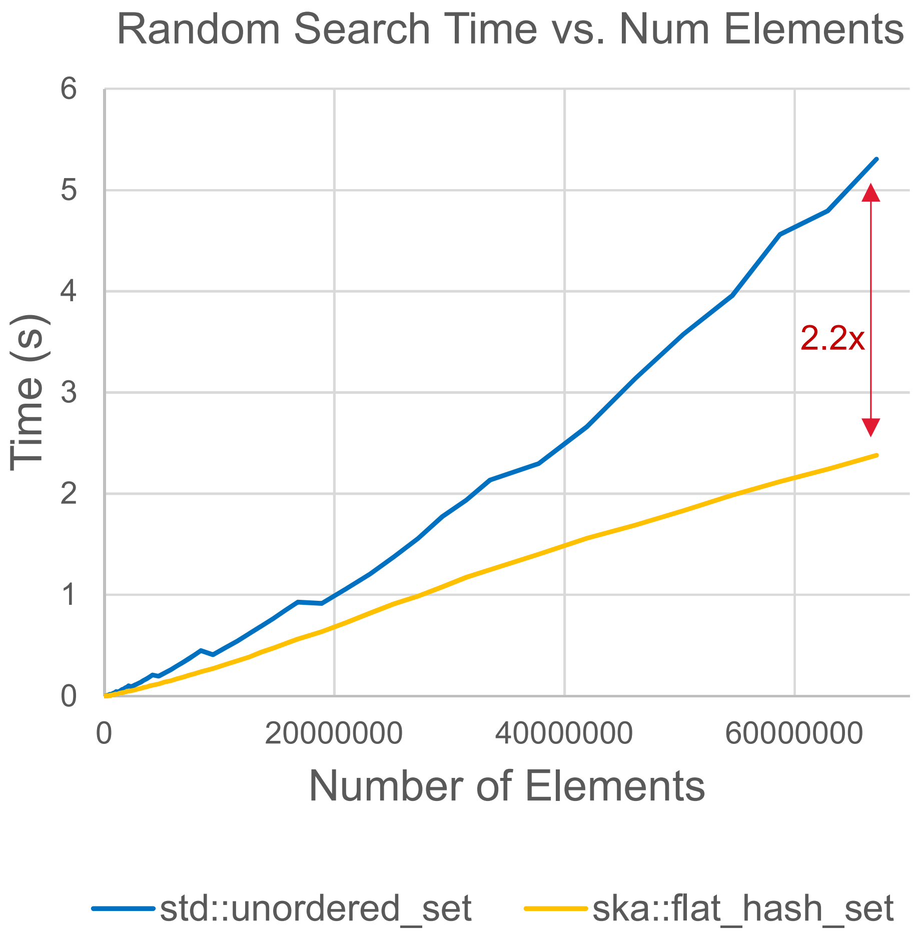
# (3) Access Locality

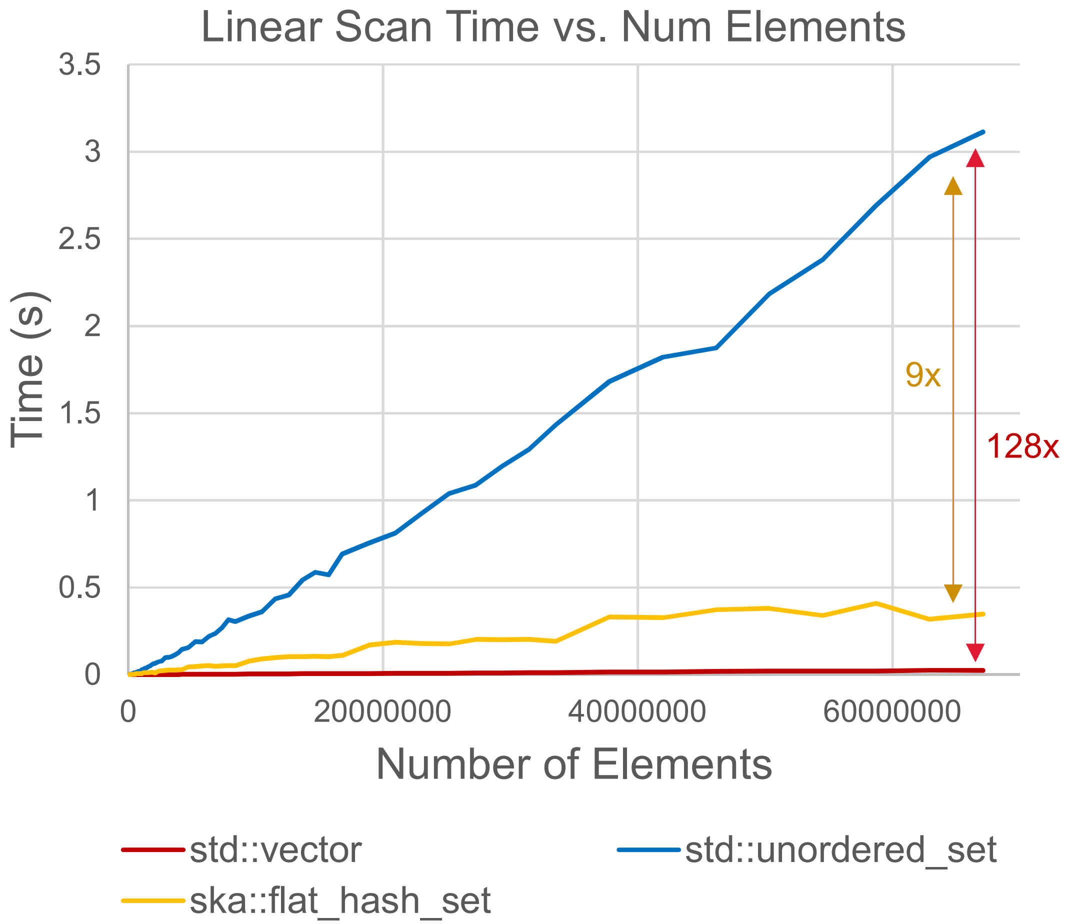




Tests run on MSVC++ 2017 64b, i7 8700

# (3) Access Locality





Tests run on MSVC++ 2017 64b, i7 8700

# (3) Access Locality

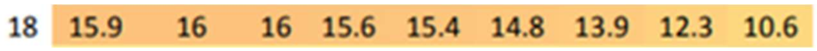
- Collection summary:
  - Look beyond just big-O notation as constant-time costs can differ significantly.
  - More tests at https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html

# (3) Access Locality

- Allocators
  - Lots of data in: On Quantifying Memory-Allocation Strategies, John Lakos et al

**Throughput ratio (local arena):(global allocator)**

Decreasing temporal locality →



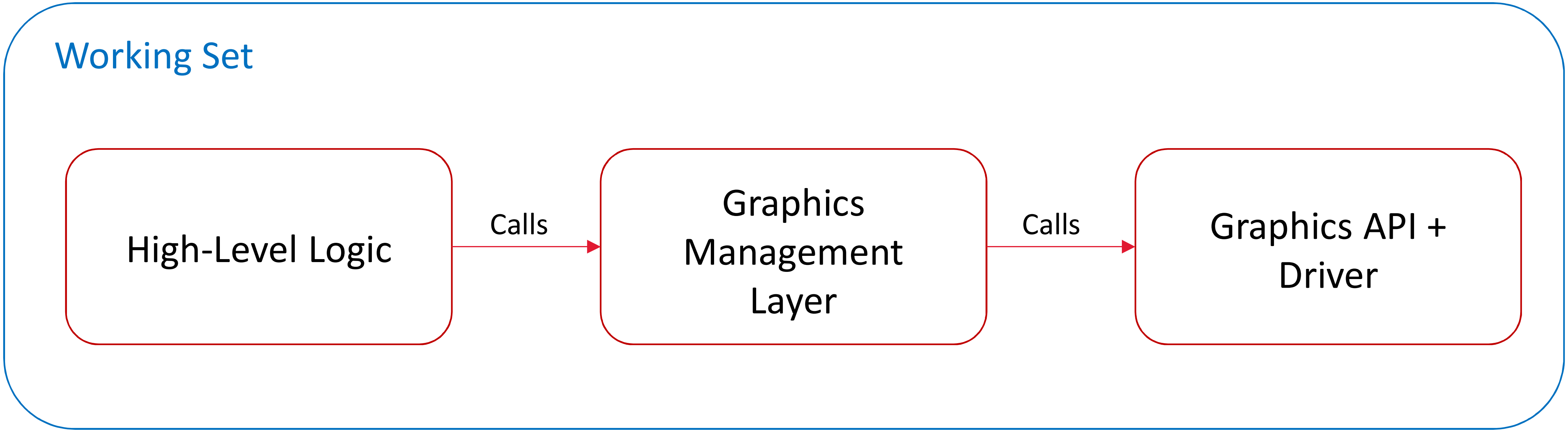
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 18 | 15.9 | 16 | 16 | 15.6 | 15.4 | 14.8 | 13.9 | 12.3 | 10.6 |

$2^{25}$ total links, $2^{18}$ std::list<int> initial size

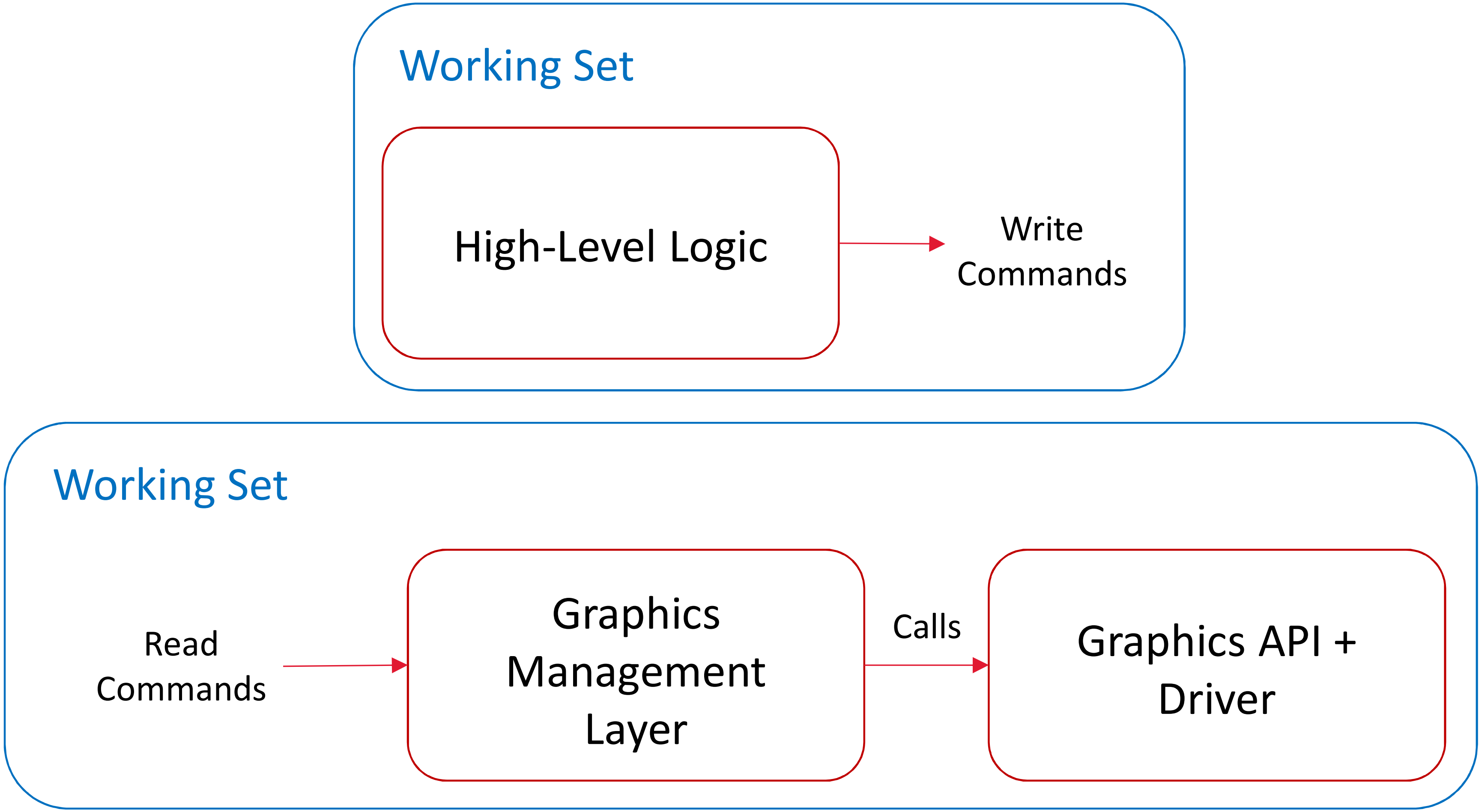
Source: On Quantifying Memory-Allocation Strategies, John Lakos et al.

As much as 16x faster with the local arena.

# (3) Access Locality



# (3) Access Locality



## (3) Access Locality

- Key Takeaway:
    - Spatial and temporal locality.
    - Large benefit in hitting faster cache levels.

# MESI States

**Modified**

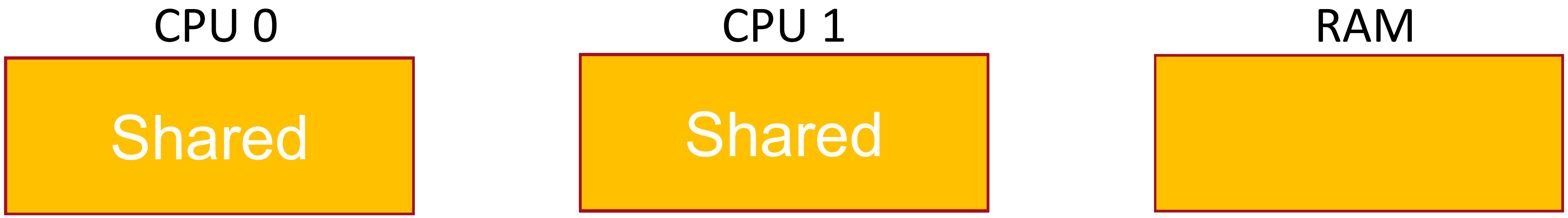
**Exclusive**

**Shared**

**Invalid**

# (4) Multiple CPU Core Considerations

## Example MESI Operation on a Single Cache Line



CPU 0

Shared

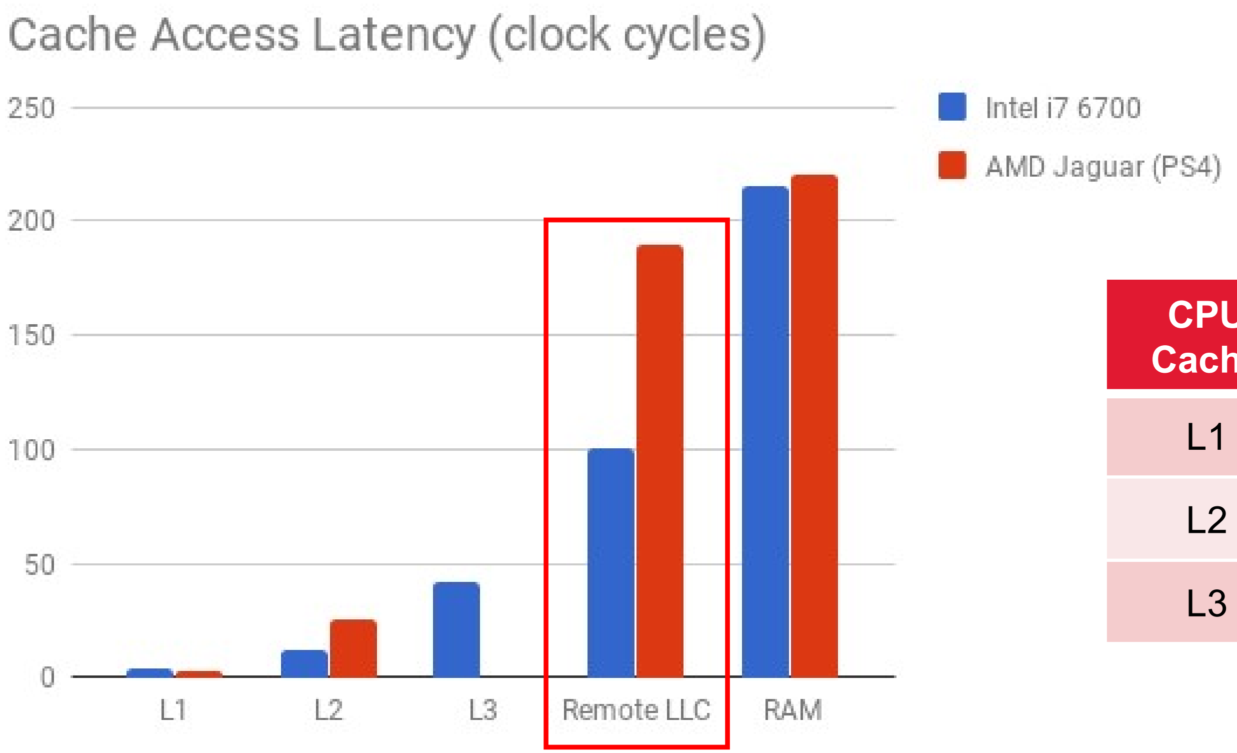
CPU 1

Shared

RAM

1. CPU 0 read -> Exclusive (only in this cache, same as in RAM)
2. CPU 1 read -> Shared (potentially in multiple caches, same as in RAM)
3. CPU 0 write -> Modified (only in this cache, different to RAM)
4. CPU 1 write -> Modified. CPU 0 previous value written to RAM.
5. CPU 0 read -> Shared. CPU 1 value written into RAM.

# (4) Multiple CPU Core Considerations



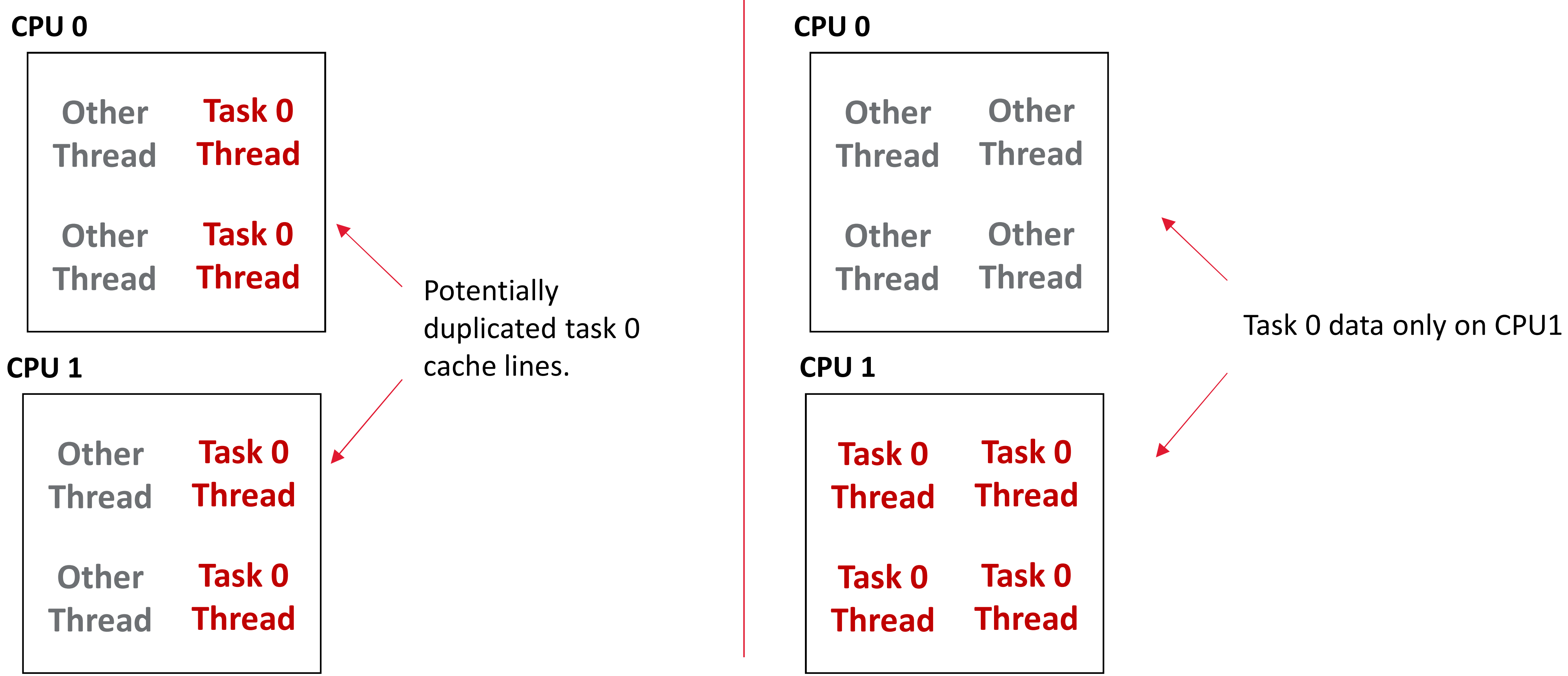
Note: Intel i7 6700 RAM latency calculated at 3.4GHz base frequency as 51ns RAM + 42 cycle L3 latency.

Intel i7 Remote LLC access figure for Xeon i7 5500 (~100-300 cycles).

PS4 figures from SINFO XXI, Jason Gregory

| CPU Cache | Intel i7 8700 | AMD Jaguar (PS4/Xbox One) |
|---|---|---|
| L1 | Private | Private |
| L2 | Private | ***Shared*** |
| L3 | ***Shared*** | N/A |

# (4) Multiple CPU Core Considerations



CPU 0

Other Thread | Task 0 Thread

Other Thread | Task 0 Thread

CPU 1

Other Thread | Task 0 Thread

Other Thread | Task 0 Thread

Potentially duplicated task 0 cache lines.

CPU 0

Other Thread | Other Thread

Other Thread | Other Thread

CPU 1

Task 0 Thread | Task 0 Thread

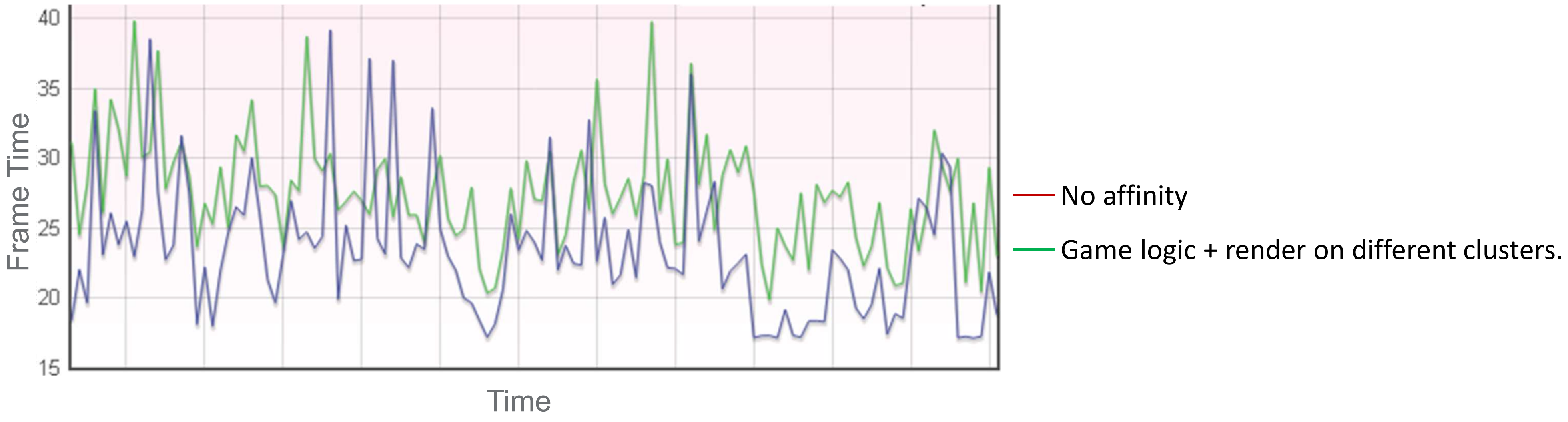
Task 0 Thread | Task 0 Thread

Task 0 data only on CPU1

# (4) Multiple CPU Core Considerations

- In-development Wargaming title:

Max Frame Times (1s periods – lower is better)



- Monolith’s Middle-Earth - Shadow of War:
    - 10% performance gain

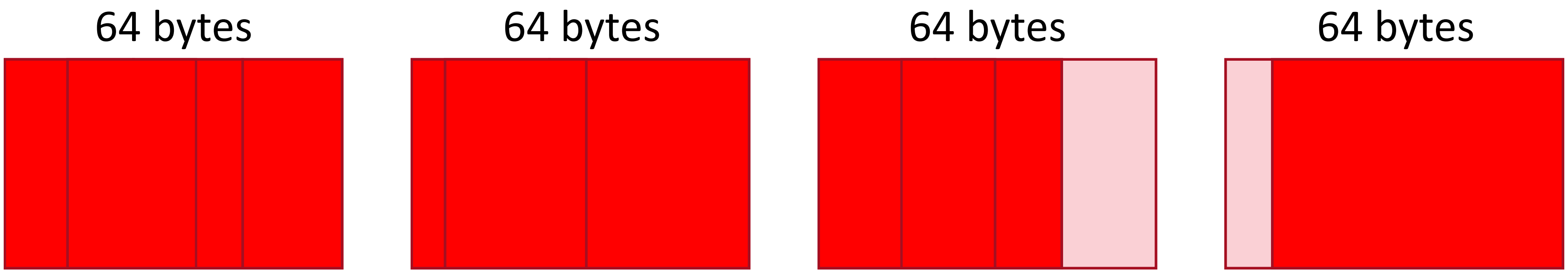
# (4) Multiple CPU Core Considerations

- Key Takeaway:
  - Take care to avoid data sharing problems.

# (5) Write Combined Memory

- Accumulate writes to flush as 64B operations

# Write Combine Buffers

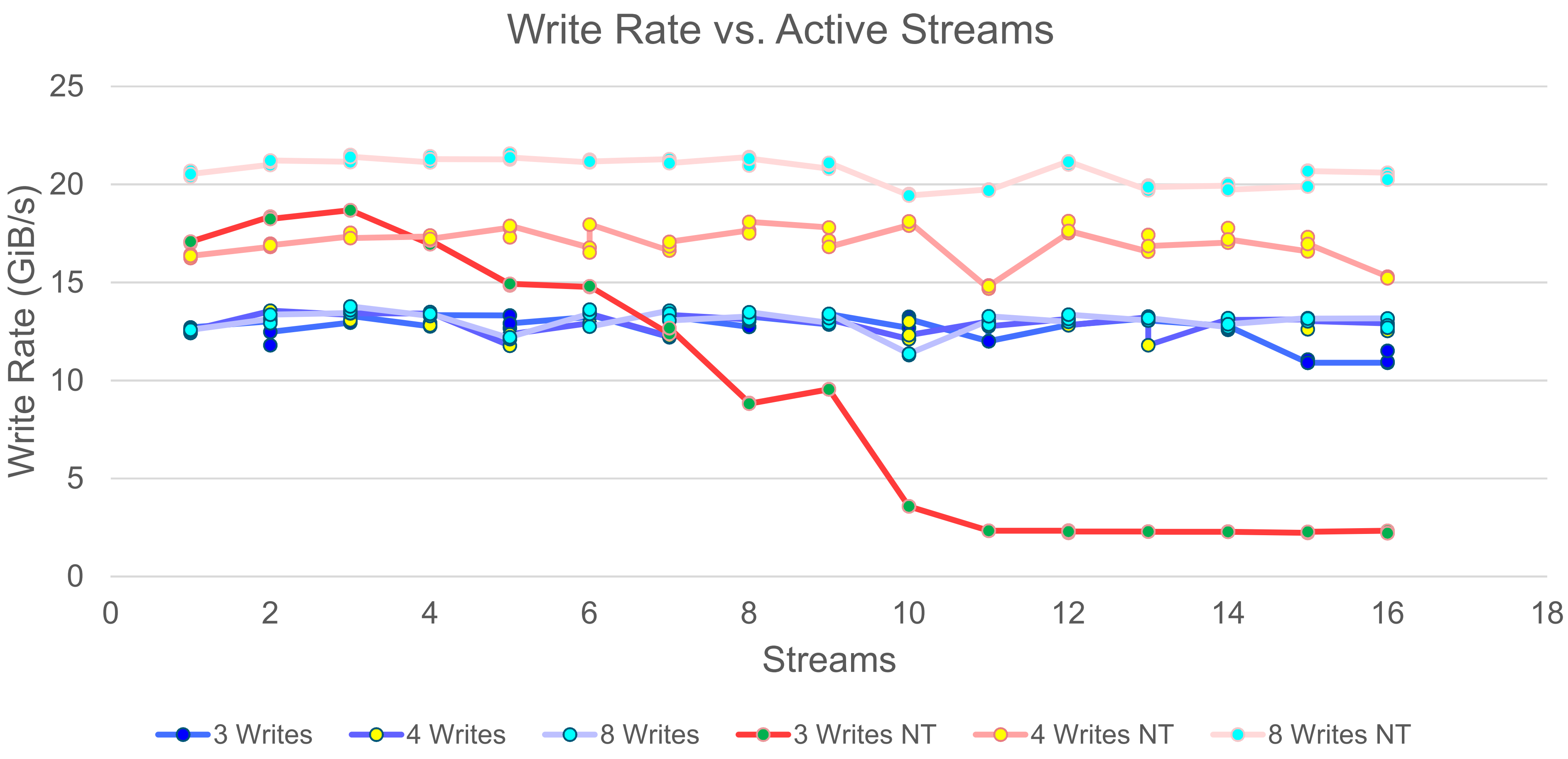


Write 64B
to RAM

Write 64B
to RAM

Partial line
to RAM

# (5) Write Combined Memory



Note:

- Writes are 16 bytes
- Intel i7 8700 system

# (5) Write Combined Memory

- WC memory read causes:
    - C++ bit fields
    - Optimizations
    - Virtually always an accident.
- Expose write-only interface.

# Non-temporal Writes on x86

Use compiler intrinsics:

- SSE2
  - _mm_stream_si32: store 4 bytes
  - _mm_stream_si128: store 16 bytes
- AVX
  - _mm256_stream_si256: store 32 bytes
- AVX-512
  - _mm512_stream_si512: store 64 bytes

# (5) Write Combined Memory

- Key Takeaway:
  - Avoids cache for performance reasons.
  - Make sure you avoid the pitfalls.

# TLB Size (4KiB pages)

Intel Skylake Series (per hardware thread) :

- L1 data: 64 entries => 256KiB addressed
- L2 shared: 1536 entries => 6144KiB addressed

AMD Jaguar Series (as in PS4/XBox One) :

- L1 data: 40 entries => 160KiB addressed
- L2 data: 512 entries => 2048KiB addressed

# TLB Size (2MiB pages)

Intel Skylake Series (per hardware thread) :

- L1 data: 32 entries => 64MiB addressed
- L2 shared: 1536 entries => 3072MiB addressed

AMD Jaguar Series (as in PS4/XBox One) :

- L1 data: 8 entries => 16MiB addressed
- L2 data: 256 entries => 512MiB addressed

# (6) Address Translation

- Platform-specific
- Not directly pageable.
- Difficult/slow to allocate.
- Windows:
  - Requires special permissions (SeLockMemoryPrivilege privilege).
  - Pass the flag MEM_LARGE_PAGES to VirtualAlloc. Use a custom allocator.
- Linux:
  - Huge TLB Page (Linux):
    - Allocate on hugetlbfs.
    - Access via mmap or shared memory.
  - Transparent Huge Pages (Linux):
    - Latency spike concerns?

# (6) Address Translation

- Key Takeaway:
    - Address translation can be a significant overhead.
    - Large pages can help.

# Conclusion

1. Cache Lines
2. Hardware Prefetch
3. Access Locality
4. Multi-core
5. Write-combined Memory
6. Address Translation

# The End

- Thanks!
- Questions?

@ScottMcMillan0

# References

- Intel Vtune Memory Bound Metric
- Latency Numbers that Every Programmer Should Know
- CPPCon 2016 High Performance Code 201: Hybrid Data Structures
- Mike Acton CPPCon 2014: Data Oriented Design
- Intel 64 and IA-32 Optimization Manual April 2018
- AMD Software Optimization Guide for AMD Family 15h Processors
- AMD Software Optimization Guide for AMD Family 17h Processors
- Baptiste-Wicht Container Comparison 1, Comparison 2
- C++Now 2018: You Can Do Better Than std::unordered_map
- On Quantifying Memory-Allocation Strategies, John Lakos et al
- CPPCon 2017: Local ('Arena') Memory Allocators
- Quantifying the Cost of a Context Switch
- Performance & Memory Post-mortem for Middle-earth: Shadow of War (GDC2018)
- Write Combining Is Not Your Friend
- Large Page Support (Windows), Remarks on MEM_LARGE_PAGES
- Linux Huge TLB Page, Transparent Huge Pages