CS486: Principles of Distributed Computing

# Introduction to Concurrent Programming

Nikolaos D. Kallimanis (Author)
nkallima@ics.forth.gr

Presentation updated on November 2020 by
**Eleftherios Kosmas (Presenter)**
**ekosmas@csd.uoc.gr**

# Outline

1. Basics on Multiprocessor Architecture

2. Primitives

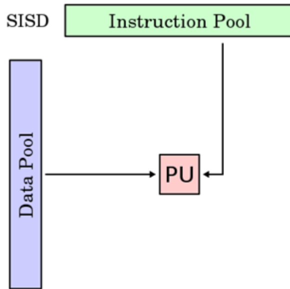
3. Code Optimization

4. Algorithmic Optimizations

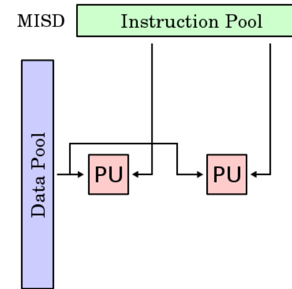
# Computer Architectures

## Flynn's taxonomy, 1966

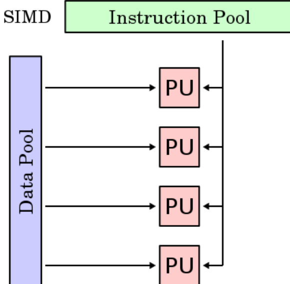
- Single instruction stream single data stream (**SISD**)



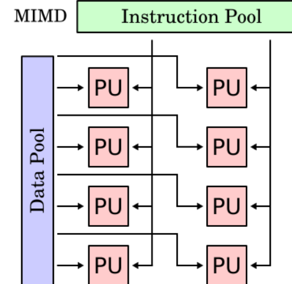

- Multiple instruction streams, single data stream (**MISD**)




- Single instruction stream, multiple data streams (**SIMD**)



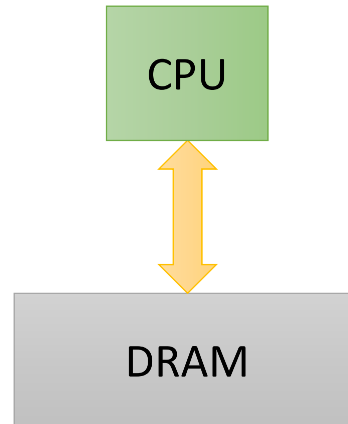

- Multiple instruction streams, multiple data streams (**MIMD**)




source: wikipedia

➢ We focus on MIMD machines

# Single Processor Architecture



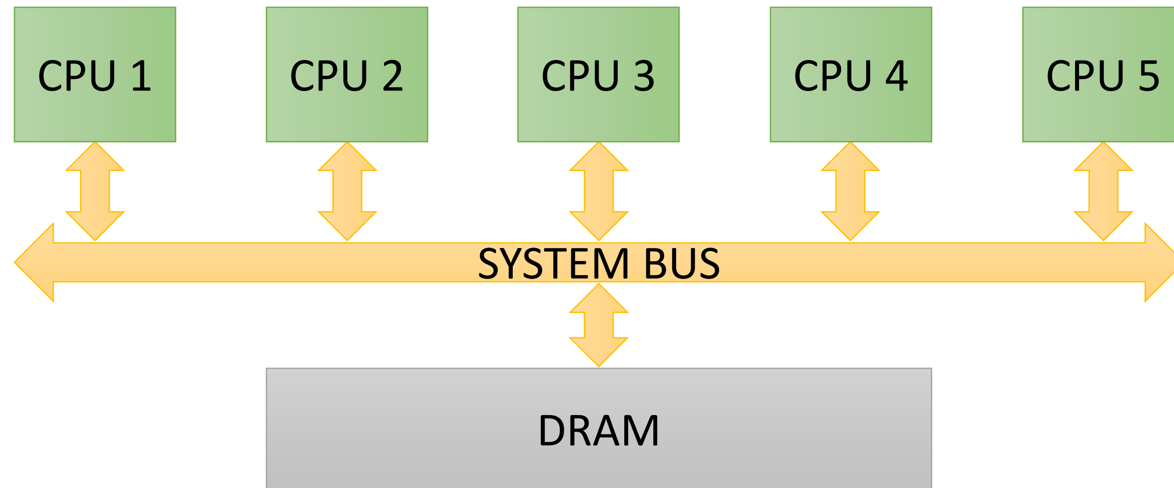

There is a **single** processing core

- accesses the main memory via a **bus**
- it is usually equipped with a **cache** for accelerating main memory accesses
- its performance is **linear** to processor's clock

➢ **Physical limits** on processors clocks

➢ **No parallelism**

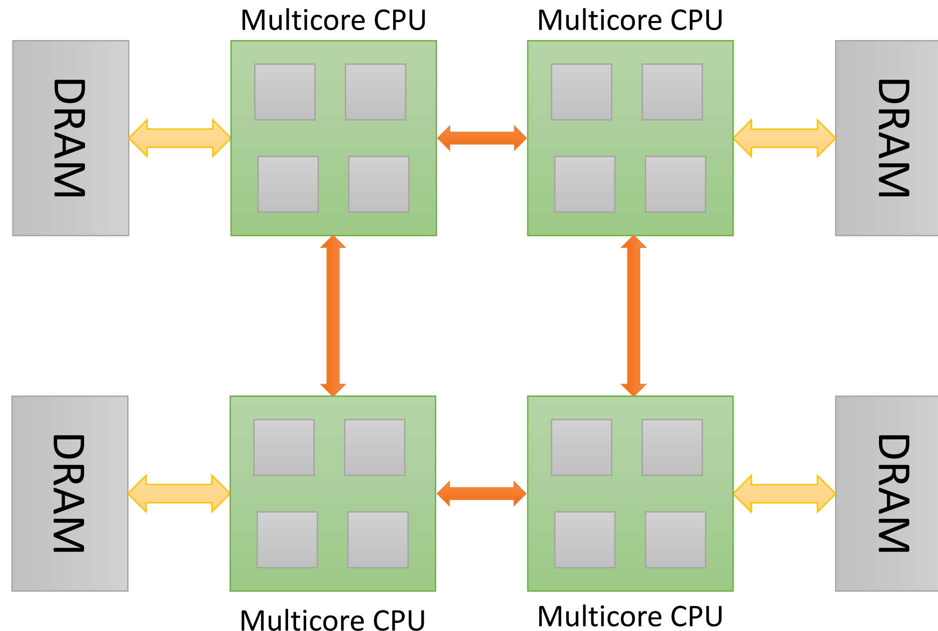
# Multiprocessor Architecture – SMP




- SMP machines are equipped with **two** or **more** processing cores
- Processing cores communicate via a **centralized** main memory
- Each processor may be equipped with a private high-speed **cache**
- **Uniform** access on the main memory

➢ Main performance bottleneck: the **centralized** nature of the **main memory**

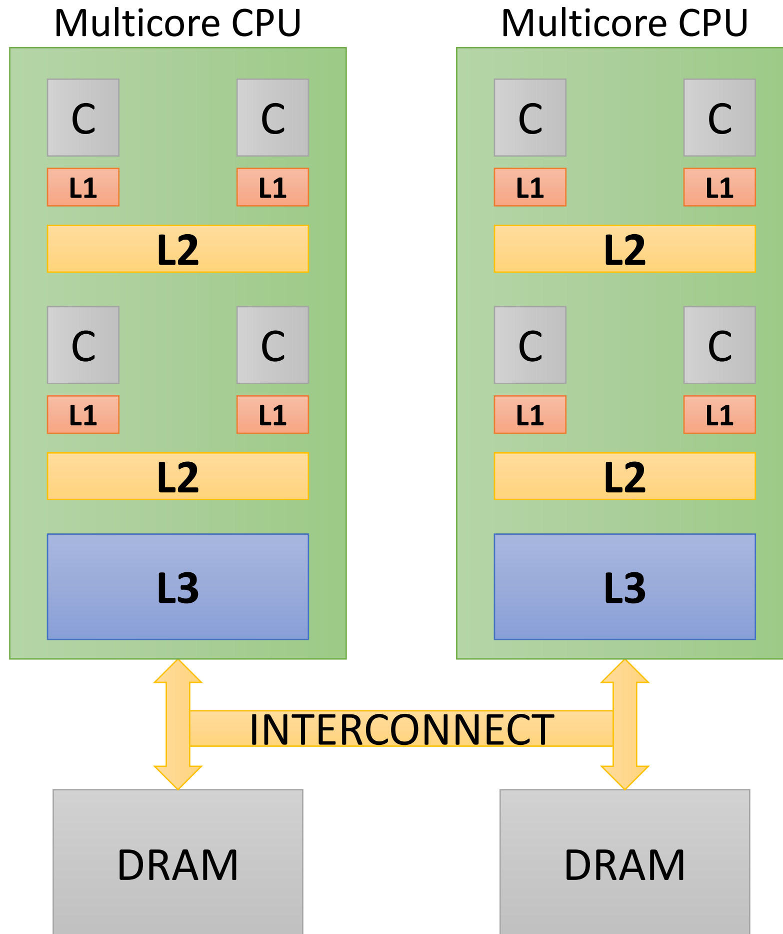
# Multiprocessor Architecture - NUMA




- NUMA machines are equipped with **two** or **more** processing cores
- **Sophisticated Interconnect** for communication among processing cores and memory
- Main memory is **distributed** among clusters of cores → **Non-Uniform** access
- Each processor may be equipped with a private high-speed **cache**

➢ **Difficult programming** for achieving high performance

# Caches in Multiprocessors

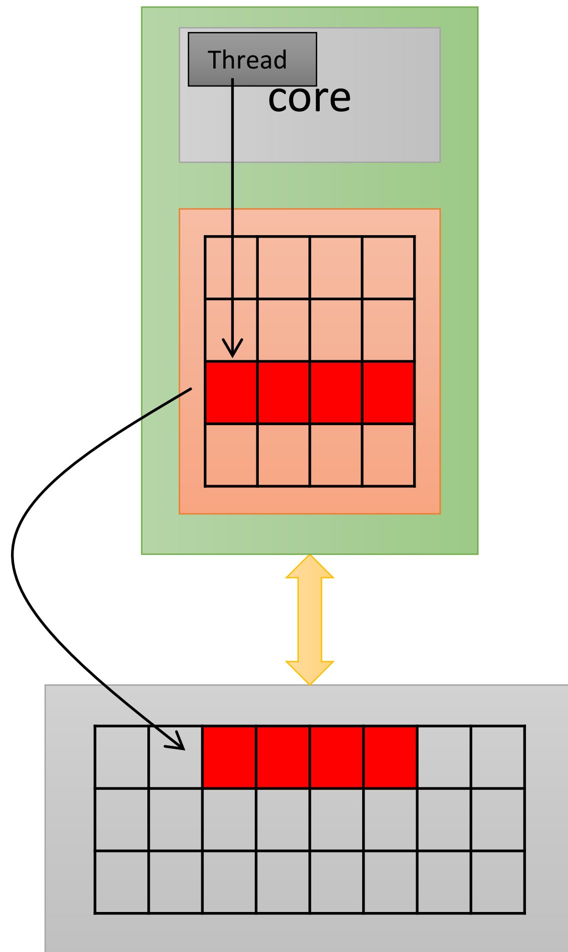



- Caches play a very **important** role in performance
- Modern multiprocessors are equipped with **more than one level** of caches

- **Private** caches: accessible only by a single core
  - usually, the L1 cache of a CPU
- **Shared** caches
  - L3 cache on a modern CPU

- A single memory location may be cached in **more than one** caches

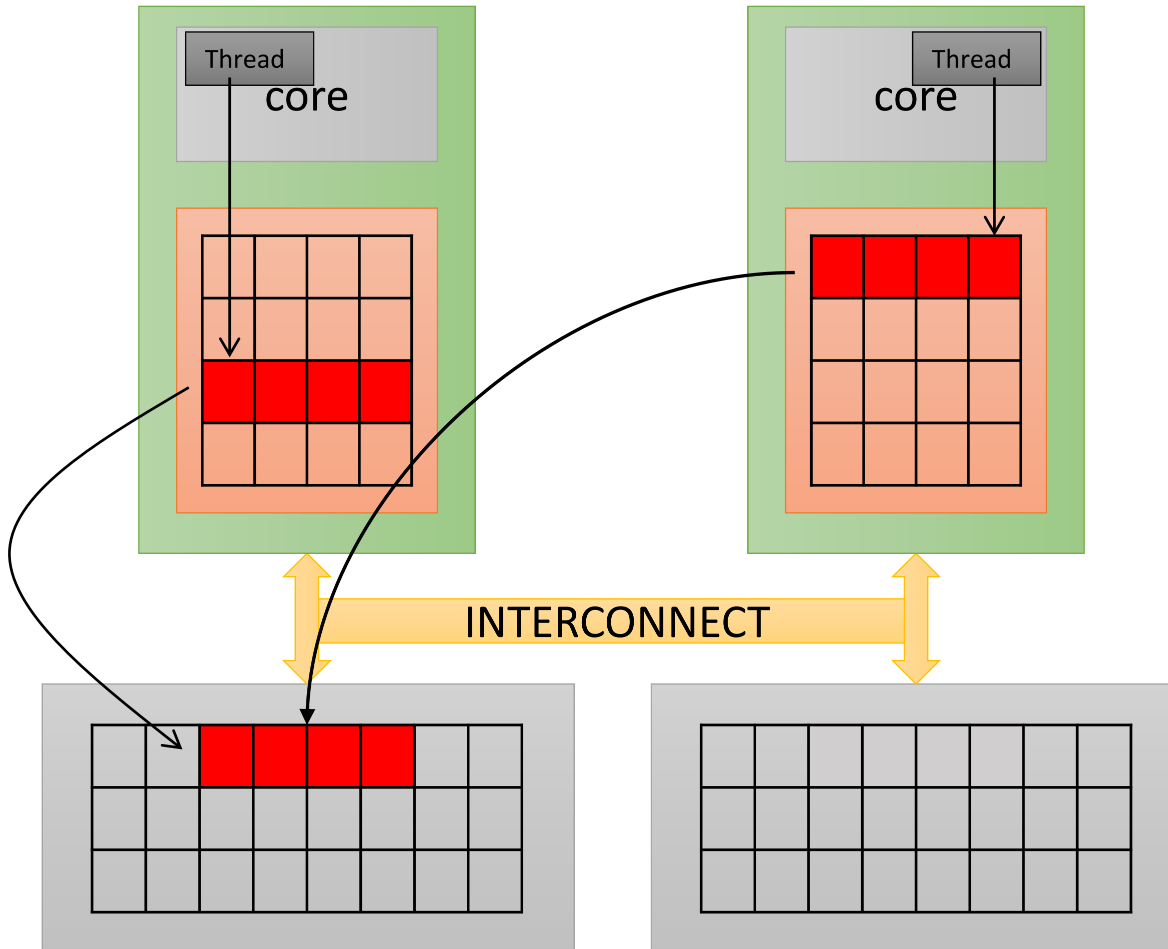
# Caches in Multiprocessors (II)




- Cache is divided into cache-lines
- A cache-line **depicts** the data stored on a small memory chunk.
- The **size** of cache-lines is usually 64 bytes
  - in some rare cases it is either 32 or 128 bytes
- Each cache level may have its **own** cache-line size, e.g.
  - 32 bytes for L1 and
  - 64 bytes for L2 and L3

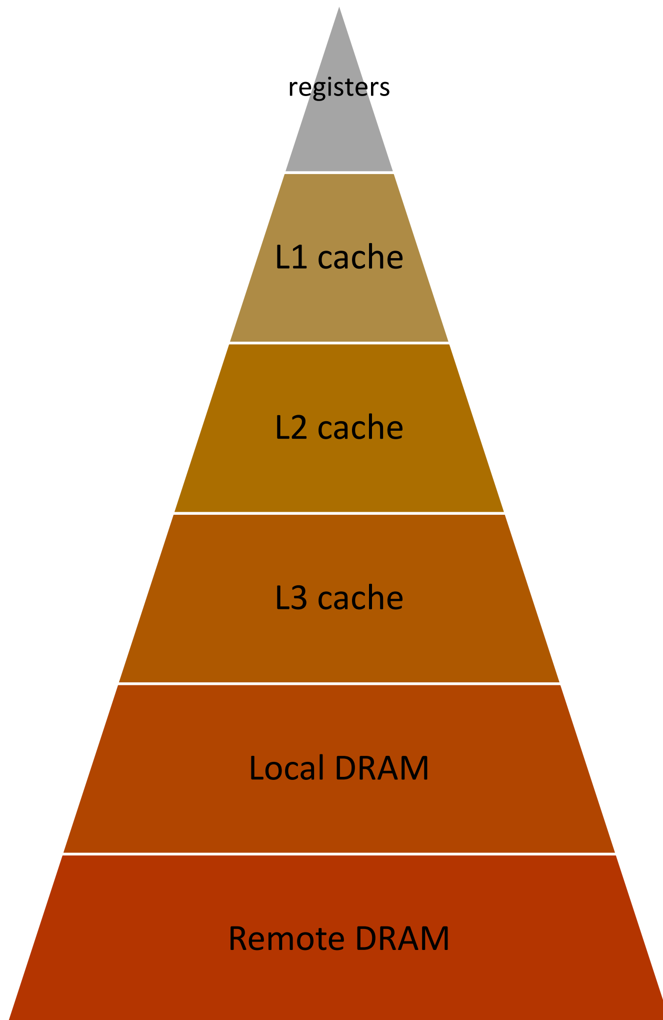
# Caches in Multiprocessors

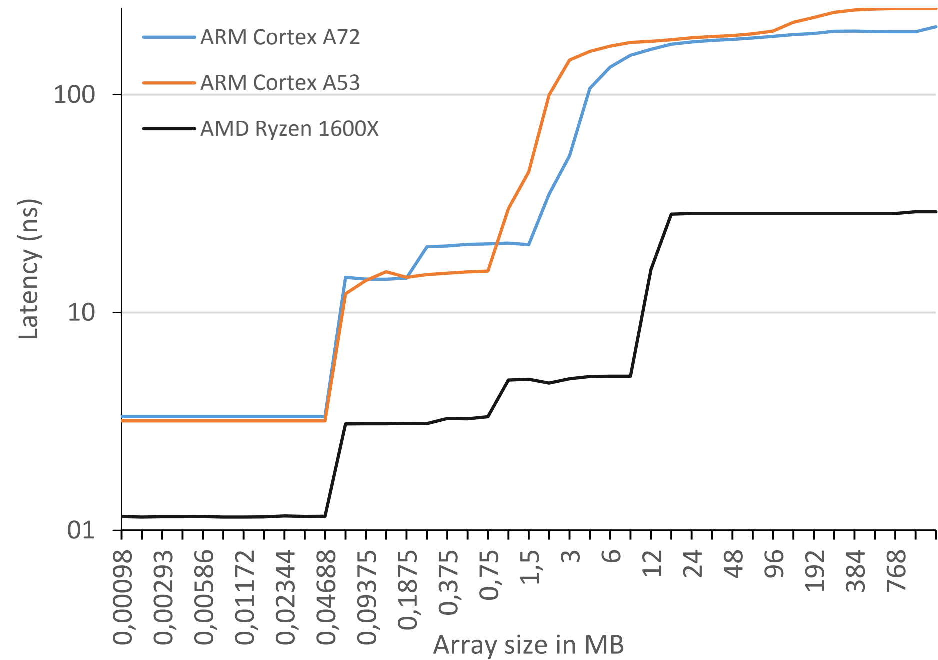
## Coherence Protocols




- Two or more **cores** may have a **copy** of a single memory area in their caches
- **Coherency protocols** are responsible for maintaining coherency among caches
  - e.g. **MESI** and **MOESI**
- **Updating** a value to a local cache may result on **invalidations** of data on other caches
  - **increases** the traffic on interconnect
  - **slows down** remote processors

# Performance and Cache Hierarchy




| | Cortex-A72 | Cortex-A53 | Ryzen 1600 |
|---|---|---|---|
| Frequency | 1.2 GHz | 950 MHz | 3.6 GHz |
| L1 Data Cache | 32 KB | 32 KB | 32 KB |
| L2 Cache | 2 MB | 1MB | 512 KB |
| L3 Cache | - | - | 8 MB |
| Memory | DDR3L | | DDR4 |

# Primitives

# Shared Memory – pthreads

- `int pthread_create(pthread_t * thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`
- `pthread_join(pthread_t thread, void **retval);`
- `pthread_exit(void *retval);`
- `pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

# Shared Memory – pthreads example

```
#include <stdio.h>
#include <pthread.h>

int main(void) {
   pthread_t thread_a, thread_b;

   pthread_create(&thread_a, NULL, threadAfunc, NULL);
   pthread_create(&thread_b, NULL, threadBfunc, NULL);
   pthread_join(thread_a, NULL);
   pthread_join(thread_b, NULL);

   exit(0);
}
```

```
void *threadAfunc(void *arg) {
   printf("Thread A\n");
   return NULL;
}
```

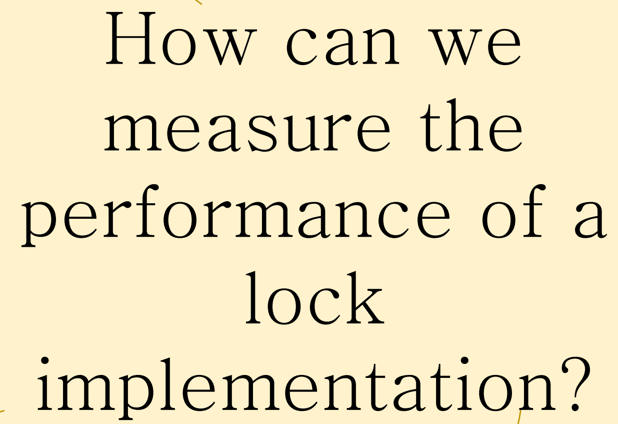
```
void *threadBfunc(void *arg) {
   printf("Thread B\n");
   return NULL;
}
```

# Shared Memory – pthreads mutex example

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main(void) {
   pthread_t thread_a, thread_b;

   pthread_create(&thread_a, NULL, threadAfunc, NULL);
   pthread_create(&thread_b, NULL, threadBfunc, NULL);
   pthread_join(thread_a, NULL);
   pthread_join(thread_b, NULL);

   exit(0);
}
```

```
void *threadAfunc(void *arg) {
  pthread_mutex_lock(&mutex);
  printf("Thread A\n");
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

```
void *threadBfunc(void *arg) {
  pthread_mutex_lock(&mutex);
  printf("Thread B\n");
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```


How can we measure the performance of a lock implementation?

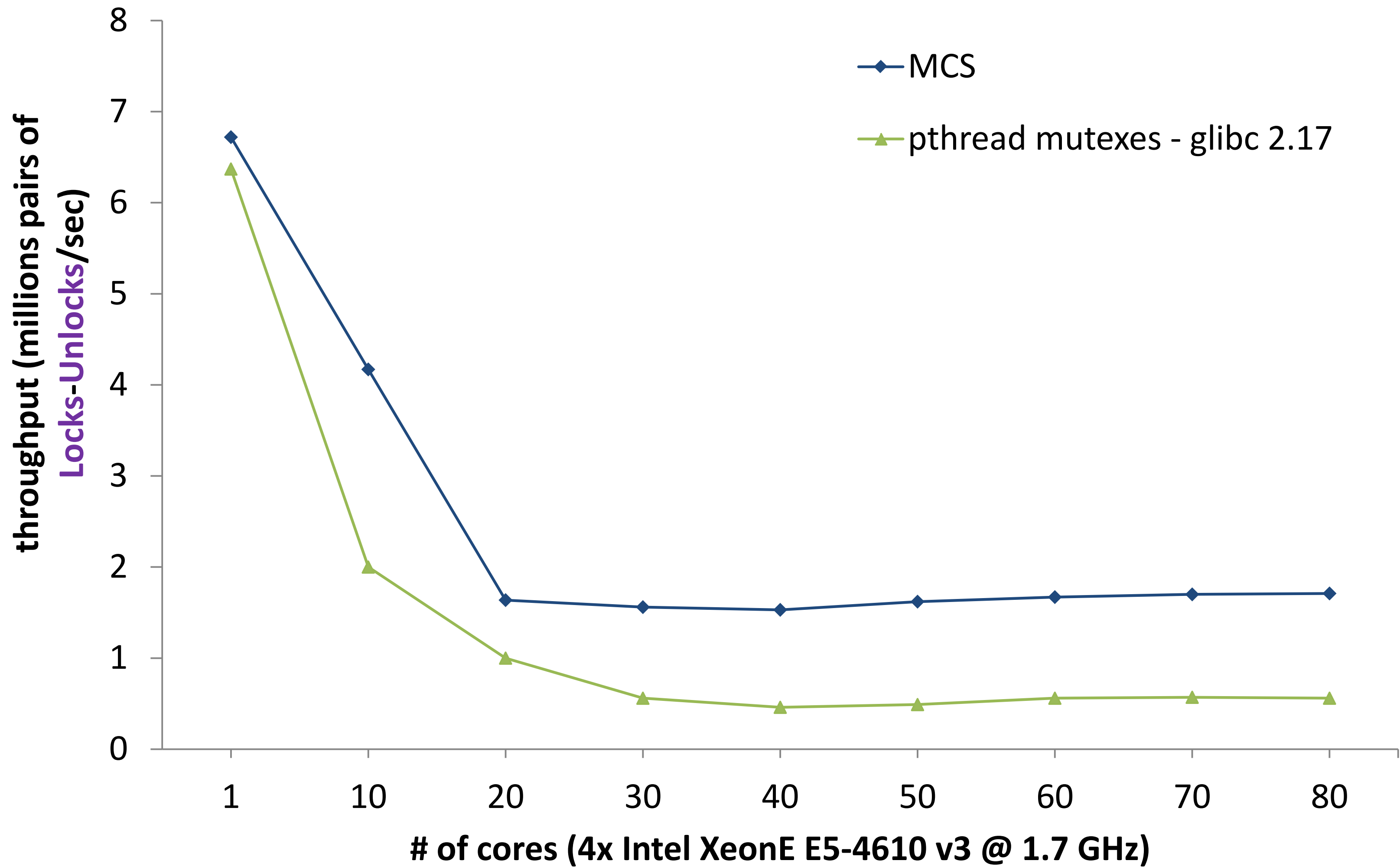
# Measuring Performance of Lock Implementations

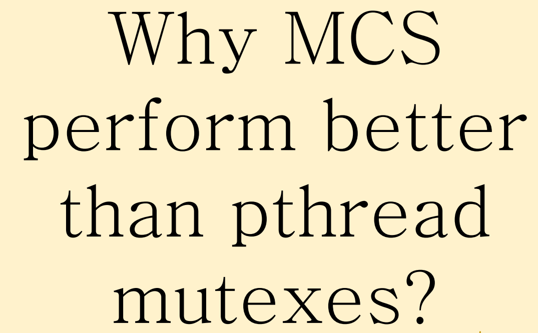
- **Spawn** a number of threads that is **equal** to the amount of cores
- Each thread executes a **big** amount of **Lock**/**Unlock** operations in a while loop manner
  - e.g. measuring the performance of pthread mutexes:

```
1. void * threadMeasuresLockFunc(void * arg) {
2.    for (int counter=0; counter < 10^7; counter++) {
3.         pthread_mutex_lock(&mutex);
4.         DummyCriticalSection();
5.         pthread_mutex_unlock(&mutex);
6.    }
7. }
```

  - the critical section should be dummy
  - remainder section should be minimal
- Calculate the **average** number of **Lock**/**Unlock** operations per second
- Repeat the experiment for **different** amounts of cores

# Pthread Mutexes VS MCS




Why MCS
perform better
than pthread
mutexes?

# Another Multi-Threaded Application Example

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;

int main(void) {
   pthread_t thread_a, thread_b;

   pthread_create(&thread_a, NULL, threadAfunc, NULL);
   pthread_create(&thread_b, NULL, threadBfunc, NULL);
   pthread_join(thread_a, NULL);
   pthread_join(thread_b, NULL);
   printf("Success\n");
   exit(0);
}
```

```
void *threadAfunc(void *arg) {
   while (counter == 0)
      ;
   return NULL;
}
```

```
void *threadBfunc(void *arg) {
   counter = 1;
   return NULL;
}
```

Compile with:

Case A) `gcc simple_example_app.c -lpthread` → Application Terminates.

Case B) `gcc -O3 simple_example_app.c -lpthread` → Application does not Terminate!

Why doesn't the simple example terminate in case 2?

# A Simple Multi-Threaded Application Example

**threadAfunc: GCC –O3 output**

```
threadAfunc:
.LFB39:
        .cfi_startproc
        movl    counter(%rip), %eax
        testl   %eax, %eax
        jne     .L2
.L3:
        jmp     .L3
        .p2align 4,,10
        .p2align 3
.L2:
        xorl    %eax, %eax
        ret
        .cfi_endproc
.LFE39:
```

**threadAfunc: GCC –O0 output**

```
threadAfunc:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -8(%rbp)
        nop
.L2:
        movl    counter(%rip), %eax
        testl   %eax, %eax
        je      .L2
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
```

# Volatiles

- Declaring a volatile variable:

```
volatile int x;
int volatile y;
```

- Declaring a non-volatile pointer to volatile data

```
volatile int *p1;
```

- Declaring a volatile pointer to non-volatile data

```
int * volatile p2;
```

- Declaring a volatile pointer to volatile data

```
volatile int * volatile p3;
```

- There is also the `register` keyword
  - `register int z;`
  - `volatile int * register w;`

- `volatile` keyword indicates that a value may change between different accesses, even if it does not appear to be modified
- `register` keyword suggests that the compiler store a declared variable in a CPU register (the location of a variable declared with `register` cannot be accessed)

# Another Multi-Threaded Application Example

That terminates with both gcc and gcc -O3

```
#include <stdio.h>
#include <pthread.h>

volatile int counter = 0;

int main(void) {
   pthread_t thread_a, thread_b;

   pthread_create(&thread_a, NULL, threadAfunc, NULL);
   pthread_create(&thread_b, NULL, threadBfunc, NULL);
   pthread_join(thread_a, NULL);
   pthread_join(thread_b, NULL);
   printf("Success\n");
   exit(0);
}
```

```
void *threadAfunc(void *arg) {
   while (counter == 0)
      ;
   return NULL;
}
```

```
void *threadBfunc(void *arg) {
   counter = 1;
   return NULL;
}
```

# Atomic Instructions

- GCC atomic builtins

```
type __sync_fetch_and_add(type *ptr, type value)
bool __sync_bool_compare_and_swap(type *ptr, type oldval, type newval)
type __sync_val_compare_and_swap(type *ptr, type oldval, type newval)
type __sync_lock_test_and_set(type *ptr, type value)
type __sync_synchronize()
```

- GCC atomic builtins compatible with C++11 (newer versions of GCC)

```
type __atomic_load_n(type *ptr, int memorder)
void __atomic_store(type *ptr, type *val, int memorder)
void __atomic_exchange(type *ptr, type *val, type *ret, int memorder)
type __atomic_add_fetch(type *ptr, type val, int memorder)
```

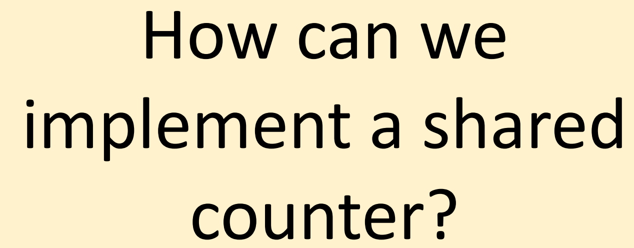
  - `memorder` could be any of:

```
__ATOMIC_SEQ_CST, __ATOMIC_ACQ_REL, __ATOMIC_RELEASE,
__ATOMIC_ACQUIRE, __ATOMIC_CONSUME, __ATOMIC_RELAXED
```

Some of the primitives provided by GCC are provided by the hardware (i.e. native), while some others are not

# Native Atomic Instructions Support

| | x86 | X86_64 | Sparc | ARMv8 |
|---|---|---|---|---|
| Fetch&Add | ✓ | ✓ | ✗ | ✗ |
| Atomic Get&Set | ✓ | ✓ | ✓(deprecated) | ✓(deprecated) |
| weak LL/SC | ✓ | ✓ | ✗ | ✓ |
| Compare&Swap | ✓ | ✓ | ✓ | ✗ |
| Fetch&XOR | ✓ | ✓ | ✗ | ✗ |
| Fetch&AND | ✓ | ✓ | ✗ | ✗ |


How can we
implement a shared
counter?

# Atomically Incrementing a Shared Counter

```
volatile long counter = 0;        // shared variable to be atomically incremented
```

## MCS

```
void MCS_lock() {
     MyPred = Get&Set(&Tail, MyNode);
     if (MyPred != NULL) {
         MyNode->locked = TRUE;
         MyPred->next = MyNode;
         while (MyNode->locked) noop;
     }
}

void MCS_unlock() {
     if (MyNode->next == NULL) {
         if (CAS(&Tail,MyNode,NULL)==TRUE)
             return;
         while(MyNode->next==NULL)
             noop;
     }
     MyNode->next->locked = FALSE;
     MyNode->next = null;
}
```

```
1.   MCS_lock();
2.   counter = counter + 1;
3.   MCS_unlock();
```

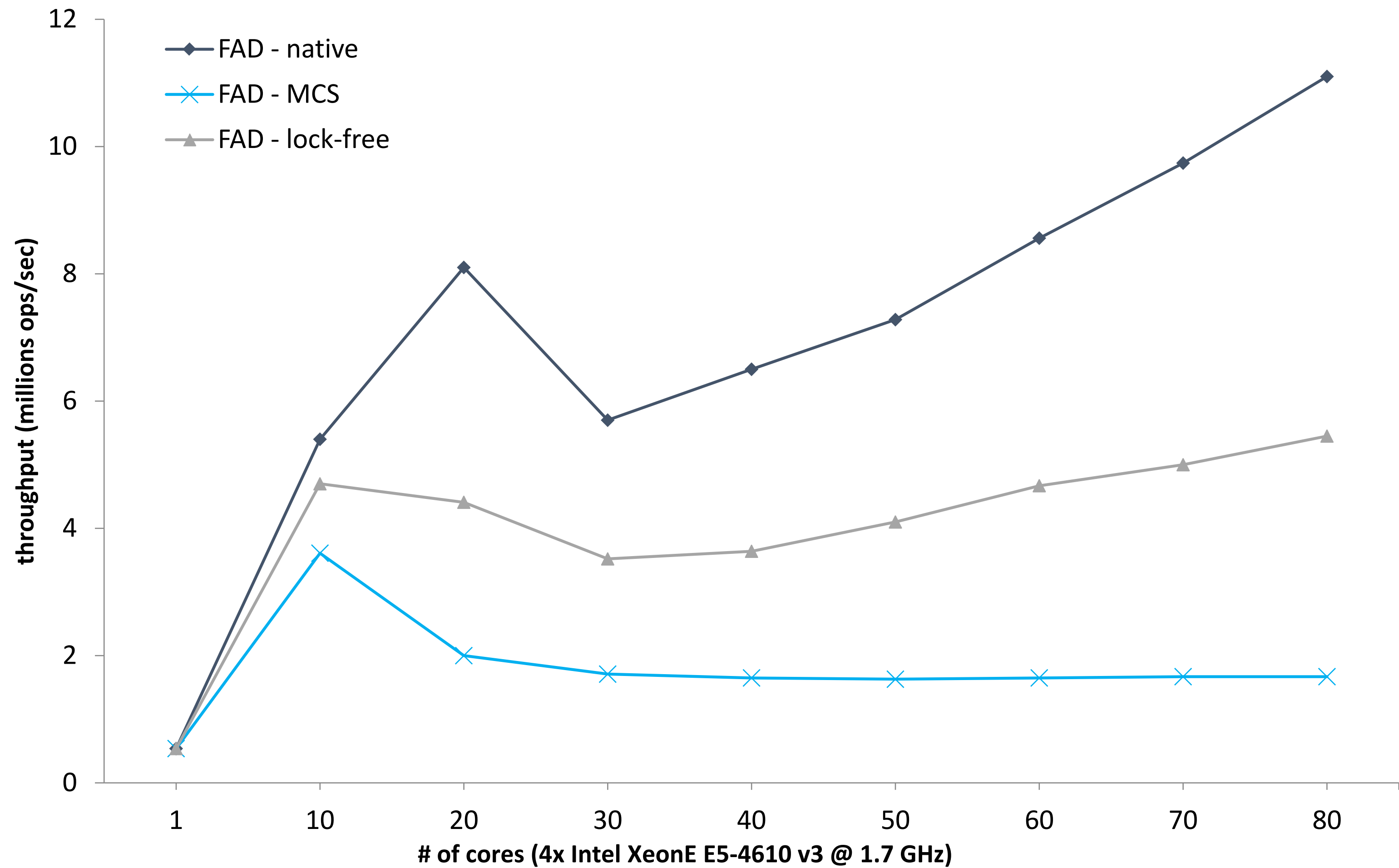
## Lock-Free

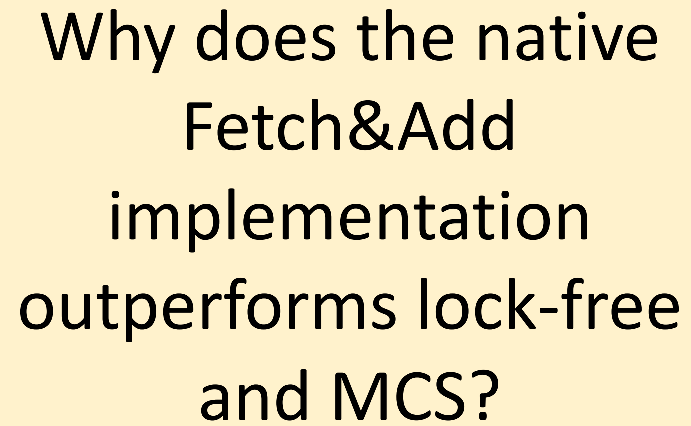
```
1.   long cur;
2.   do {
3.         cur = counter;
4.   } while (CAS(&counter,cur,cur+1)==FALSE)
```

## Fetch&Add

```
1. __sync_fetch_and_add(&counter, 1);
```

# Locks vs Lock-Free vs Atomic Instructions




Why does the native Fetch&Add implementation outperforms lock-free and MCS?

# Locks vs Lock-Free vs Atomic Instructions

```
volatile long counter = 0;     // shared variable to be atomically incremented
```

## MCS

```
void MCS_lock() {
    MyPred = Get&Set(&Tail, MyNode);
    if (MyPred != NULL) {
        MyNode->locked = TRUE;
        MyPred->next = MyNode;
        while (MyNode->locked) noop;
    }
}

void MCS_unlock() {
    if (MyNode->next == NULL) {
        if (CAS(&Tail,MyNode,NULL)==TRUE)
            return;
        while(MyNode->next==NULL)
            noop;
    }
    MyNode->next->locked = FALSE;
    MyNode->next = null;
}
```

```
1.  MCS_lock();
2.  counter = counter + 1;
3.  MCS_unlock();
```

## Lock-Free

```
1.  long cur;
2.
3.  do {
4.      cur = counter;
5.  } while (CAS(&counter,cur,cur+1)==FALSE)
```

## Fetch&Add

```
1. __sync_fetch_and_add(&counter, 1);
```

Cache **invalidations** play important role to performance!

# Code Optimization

# GCC optimizations flags

Main Optimization Flags:

- -O0, -O1, -O2, -O3, -Ofast
- -Os
- -flto

Optimizing for specific machine architectures:

- -march=native, -mtune=native
  - -mtune=niagara2
  - -mtune=bdver1

Automatic Vectorization:

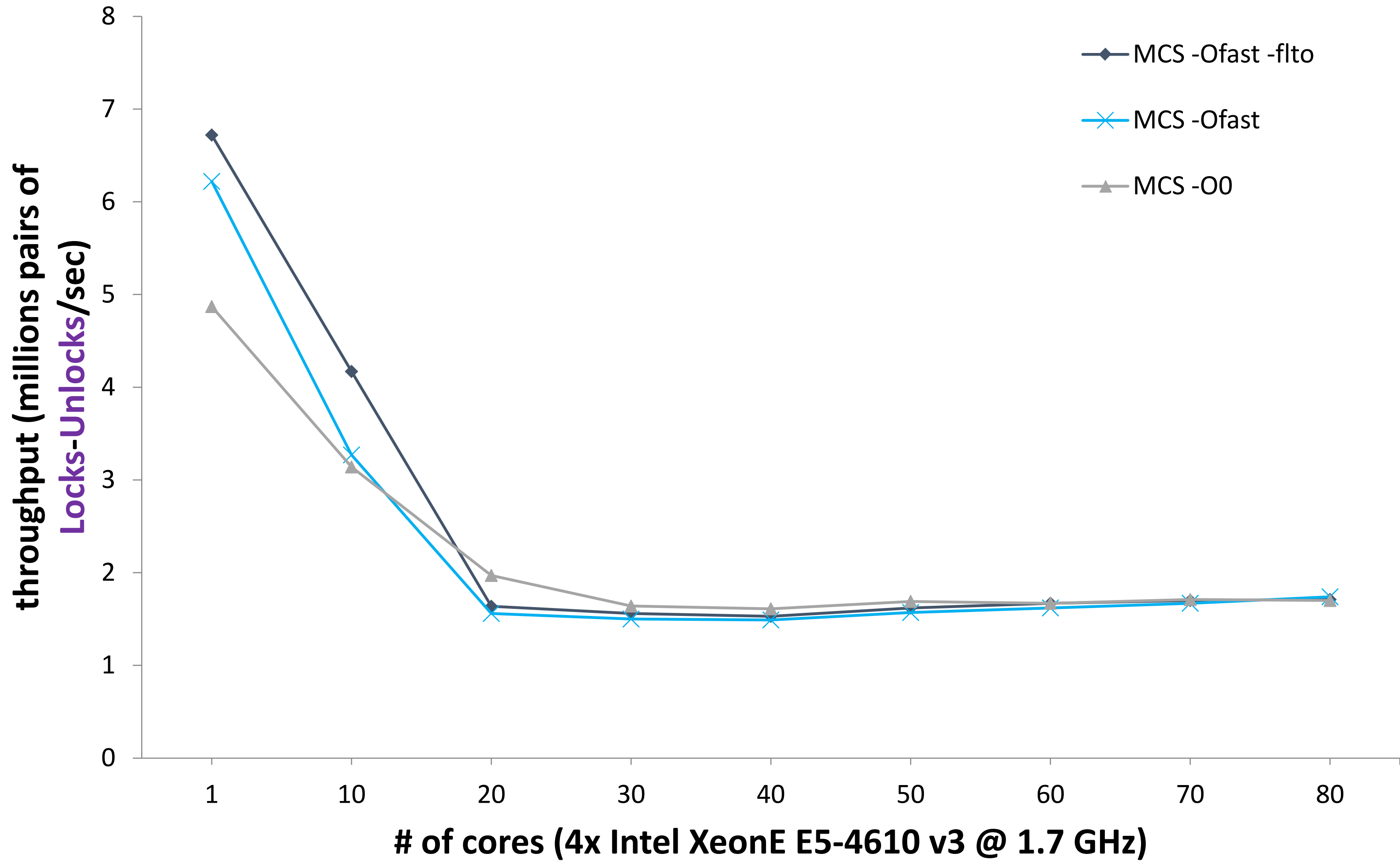
- -ftree-vectorize, -ftree-vectorizer-verbose=0
  - x86/x86_64: -msse3, -mavx, -mavx2, etc.
  - ARM: -mfpu=neon

How do GCC optimization flags affect performance?

# MCS Locks Performance vs GCC flags

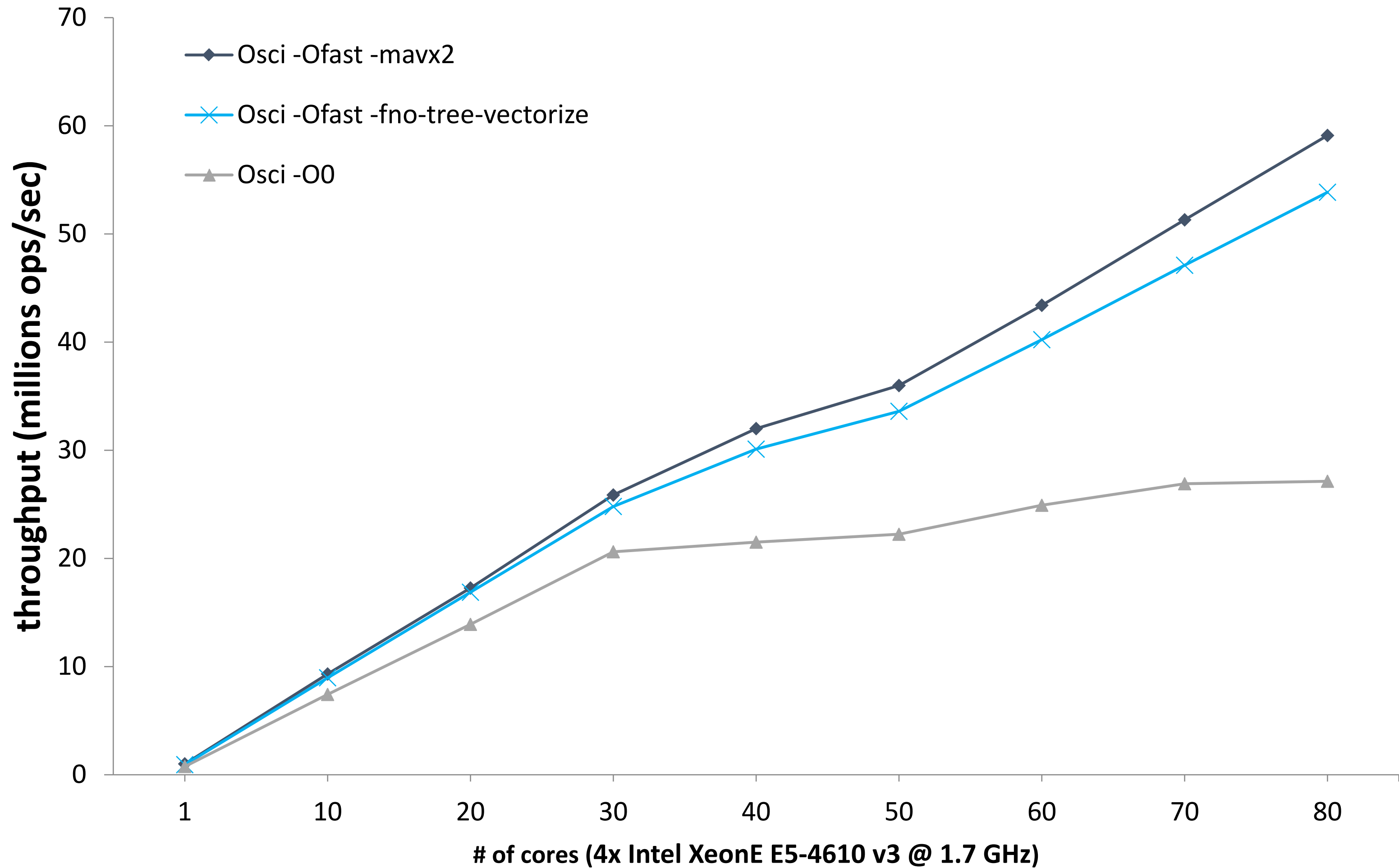
| | # cores |
|---|---|
| **Locks** & **Unlocks** millions pairs/sec | -Ofast -flto |
| | -Ofast |
| | -O0 |

# MCS* Locks Performance vs GCC flags




*https://github.com/nkallima/sim-universal-construction/tree/master/libconcurrent

# OSCI* Universal Construction vs GCC flags




*https://github.com/nkallima/sim-universal-construction/tree/master/libconcurrent

# Multiple Counters - **Application A**

```
#include <stdio.h>
#include <pthread.h>

volatile long counter[10] = {0};

int main(void) {
    pthread_t thread_a, thread_b;

    pthread_create(&thread_a, NULL, threadAfunc, NULL);
    pthread_create(&thread_b, NULL, threadBfunc, NULL);
    pthread_join(thread_a, NULL);
    pthread_join(thread_b, NULL);
}
```

```
void *threadAfunc(void *arg) {
    while (counter[0] < 10^7)
        FAD(&counter[0], 1);
    return NULL;
}
```

```
void *threadBfunc(void *arg) {
    while (counter[1] < 10^7)
        FAD(&counter[1], 1);
    return NULL;
}
```

Find the points of contention in this code!

# Multiple Counters - **Application B**

```
#include <stdio.h>
#include <pthread.h>

volatile long counter[10] = {0};

int main(void) {
   pthread_t thread_a, thread_b;

   pthread_create(&thread_a, NULL, threadAfunc, NULL);
   pthread_create(&thread_b, NULL, threadBfunc, NULL);
   pthread_join(thread_a, NULL);
   pthread_join(thread_b, NULL);
}
```

```
void *threadAfunc(void *arg) {
   while (counter[0] < 10^7)
      FAD(&counter[0], 1);
   return NULL;
}
```

```
void *threadBfunc(void *arg) {
   while (counter[9] < 10^7)
      FAD(&counter[9], 1);
   return NULL;
}
```

Find the points of contention in this code!

# Multiple Counters - **Application A** VS **Application B**:

## Performance Study

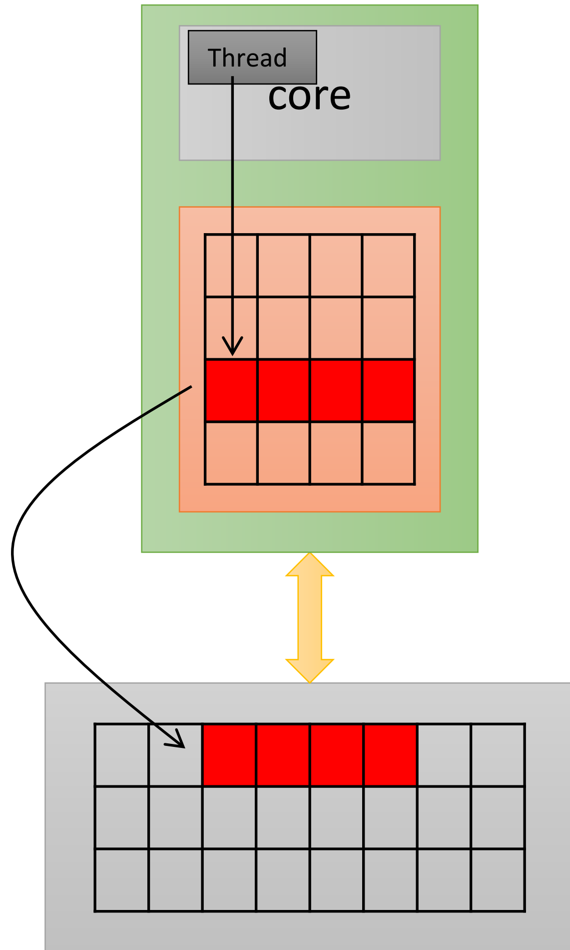
- we run both applications (A and B) in a multiprocessor with the following configuration:
  - 4x Intel Xeon CPU E5-4610 v3 @ 1.70GHz
- we measure the time consumed by each application to finish its execution
- we calculated the number of FAD operations (simulated or not) per sec

## Results

| Application | millions FAD operations / second |
|---|---|
| Application A | 8.13 |
| Application B | 14.29 |

Why is application B faster than A?

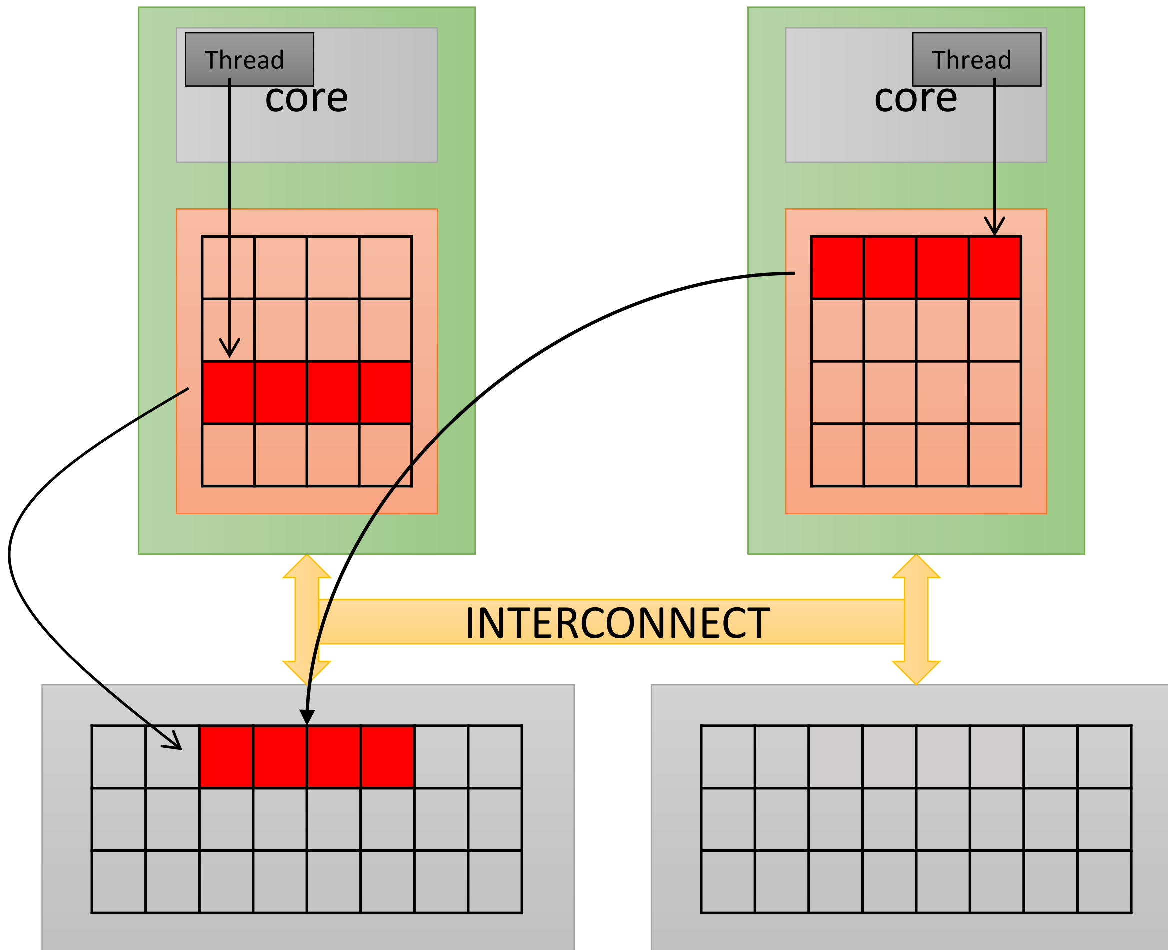
# Caches in Multiprocessors




- Cache is divided into cache-lines
- A cache-line depicts the data stored on a small memory chunk.
- The size of cache-lines is usually 64 bytes
  - in some rare cases it is either 32 or 128 bytes
- Each cache level may have its own cache-line size, e.g.
  - 32 bytes for L1 and
  - 64 bytes for L2 and L3

☞ **A cache-line may contain more than one shared objects (or variables)**

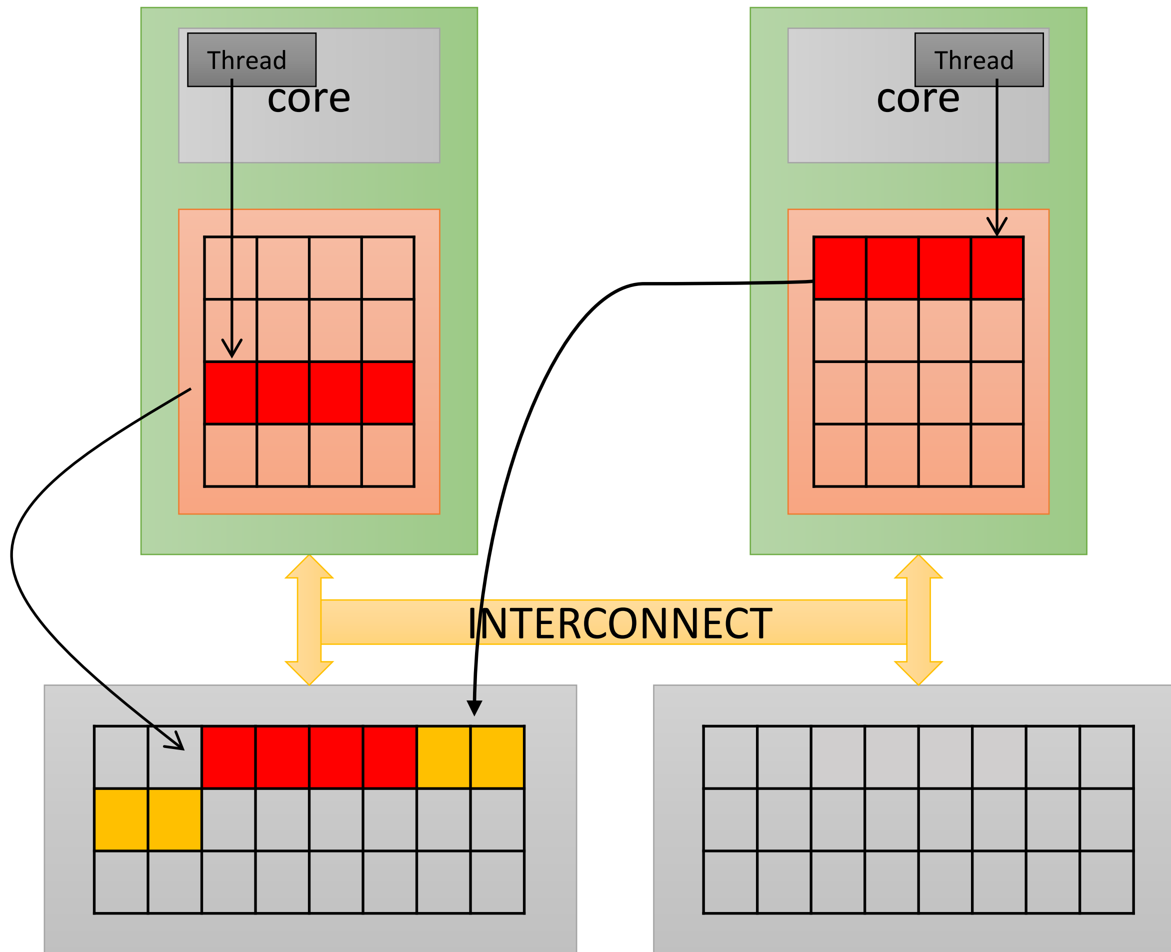
# Caches in Multiprocessors




- Two or more **cores** may have a **copy** of a single memory area in their caches
- **Coherency protocols** are responsible for maintaining coherency among caches
  - e.g. **MESI** and **MOESI**
- **Updating** a value to a local cache may result on **invalidations** of data on other caches
  - **increases** the traffic on interconnect
  - **slows down** remote processors

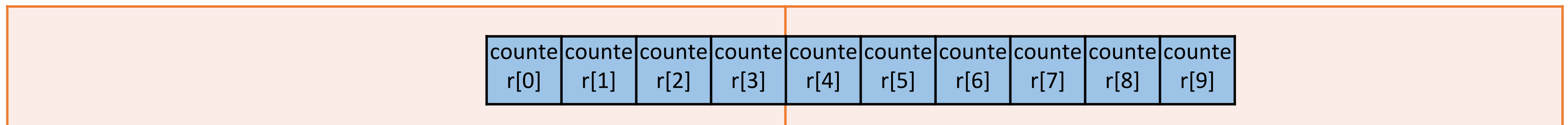
# Caches in Multiprocessors

## False Sharing



# Caches in Multiprocessors

## False Sharing - Example

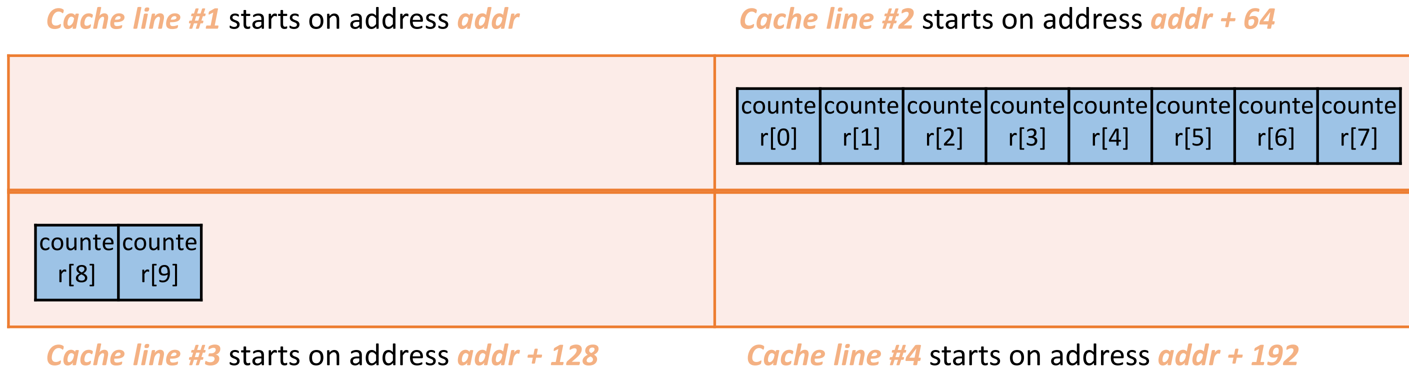


How can we eliminate false sharing?

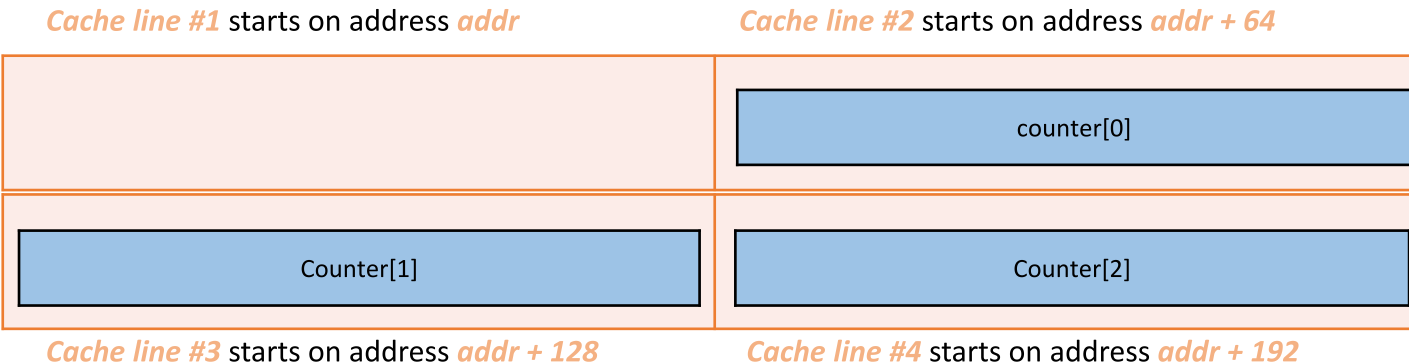
# Caches in Multiprocessors

## Eliminating False Sharing: Alignment & Padding - Example

- **Alignment**




- **Alignment & Padding**



# Caches in Multiprocessors

## Eliminating False Sharing: Alignment & Padding

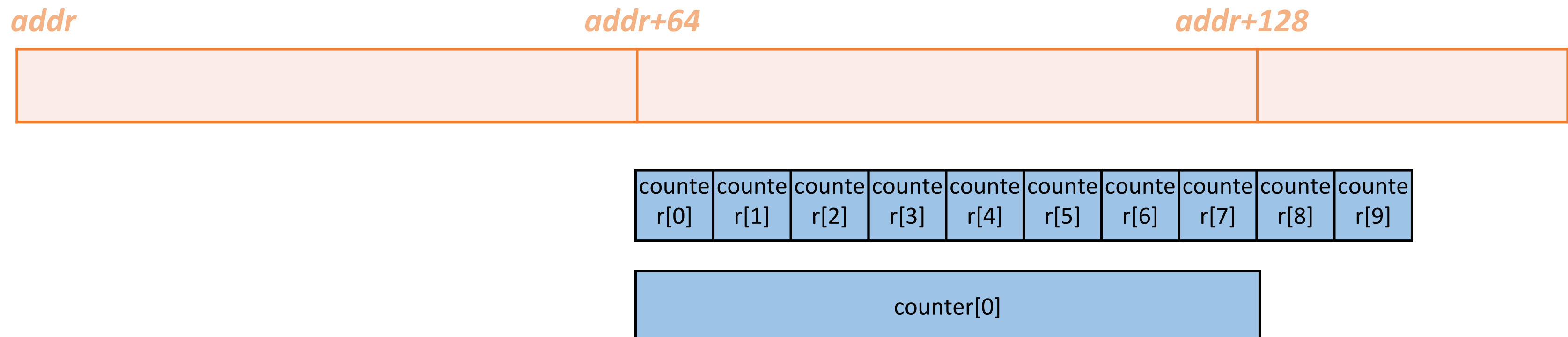
- Alignment: The GCC way:
  volatile long x __attribute__ (aligned (64)) = 0;
- Alignment: The posix way:

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc(size_t alignment, size_t size);
```

- Padding:

```
struct padded_int {
    int64_t value;
    char pad[56];
}
```

  ➢ sizeof(struct padded_int) → 64



# Multiple Counters - **Application A** - Avoid False Sharing

```
#include <stdio.h>
#include <pthread.h>

volatile struct padded_int counter[10] __attribute__ (aligned (64))={0};

int main(void) {
   pthread_t thread_a, thread_b;

   pthread_create(&thread_a, NULL, threadAfunc, NULL);
   pthread_create(&thread_b, NULL, threadBfunc, NULL);
   pthread_join(thread_a, NULL);
   pthread_join(thread_b, NULL);
}
```

```
void *threadAfunc(void *arg) {
   while (counter[0].value < 10^7)
      FAD(&counter[0].value, 1);
   return NULL;
}
```

```
void *threadBfunc(void *arg) {
   while (counter[1].value < 10^7)
      FAD(&counter[1].value, 1);
   return NULL;
}
```

Performance Restored!

# Caches in Multiprocessors

To align or not?

- There some cases that **alignment** causes performance **reduction**
- By **aligning** the data, we **increase** the memory footprint
  - the critical data may **not fit** in caches
- **No strict rule exists** that imposes when the data of an algorithm or application should be aligned

# Thread pinning/affinity

- Linux way:

```
#include <sched.h>

int TARGET_CORE_ID = 0;
int ret;
cpu_set_t mask;

CPU_ZERO(&mask);
CPU_SET(TARGET_CORE_ID, &mask);

ret = sched_setaffinity(0, sizeof(mask), &mask);
if (ret == -1)
    perror("sched_setaffinity");
```

A thread's CPU affinity mask determines the set of CPUs on which it is eligible to run

**sched_setaffinity(pid_t** *pid*, **size_t** *cpusetsize*, **const cpu_set_t** **mask***);**

- sets the CPU affinity mask of the thread whose ID is pid to the value specified by mask.
- If pid is zero, then the calling thread is used.

# Thread pinning/affinity

- Pthreads way:

```
#include <pthreads.h>

int TARGET_CORE_ID = 0;
int ret;
cpu_set_t mask;

CPU_ZERO(&mask);
CPU_SET(TARGET_CORE_ID, &mask);

ret = pthread_setaffinity_np(pthread_self(), sizeof(mask), &mask);
if (ret != 0)
   perror("pthread_setaffinity_np");
```

A thread's CPU affinity mask determines the set of CPUs on which it is eligible to run

**pthread_setaffinity_np(pthread_t** *thread*, **size_t** *cpusetsize*, **const cpu_set_t ***cpuset***);**
•sets the CPU affinity mask of the thread *thread* to the CPU set pointed to by *cpuset*.

Does thread affinity play any role in performance?

# Simple Application: No Thread Affinity

```
#include <stdio.h>
#include <pthread.h>

volatile struct padded_int counter __attribute__ (aligned (64))={0};

int main(void) {
   pthread_t thread_a, thread_b;

   pthread_create(&thread_a, NULL, threadAfunc, NULL);
   pthread_create(&thread_b, NULL, threadBfunc, NULL);
   pthread_join(thread_a, NULL);
   pthread_join(thread_b, NULL);
}
```

```
void *threadAfunc(void *arg) {
   while (counter.value < 10^7)
      FAD(&counter.value, 1);
   return NULL;
}
```

```
void *threadBfunc(void *arg) {
   while (counter.value < 10^7)
      FAD(&counter.value, 1);
   return NULL;
}
```

# Affinity on Threads: Performance Impact

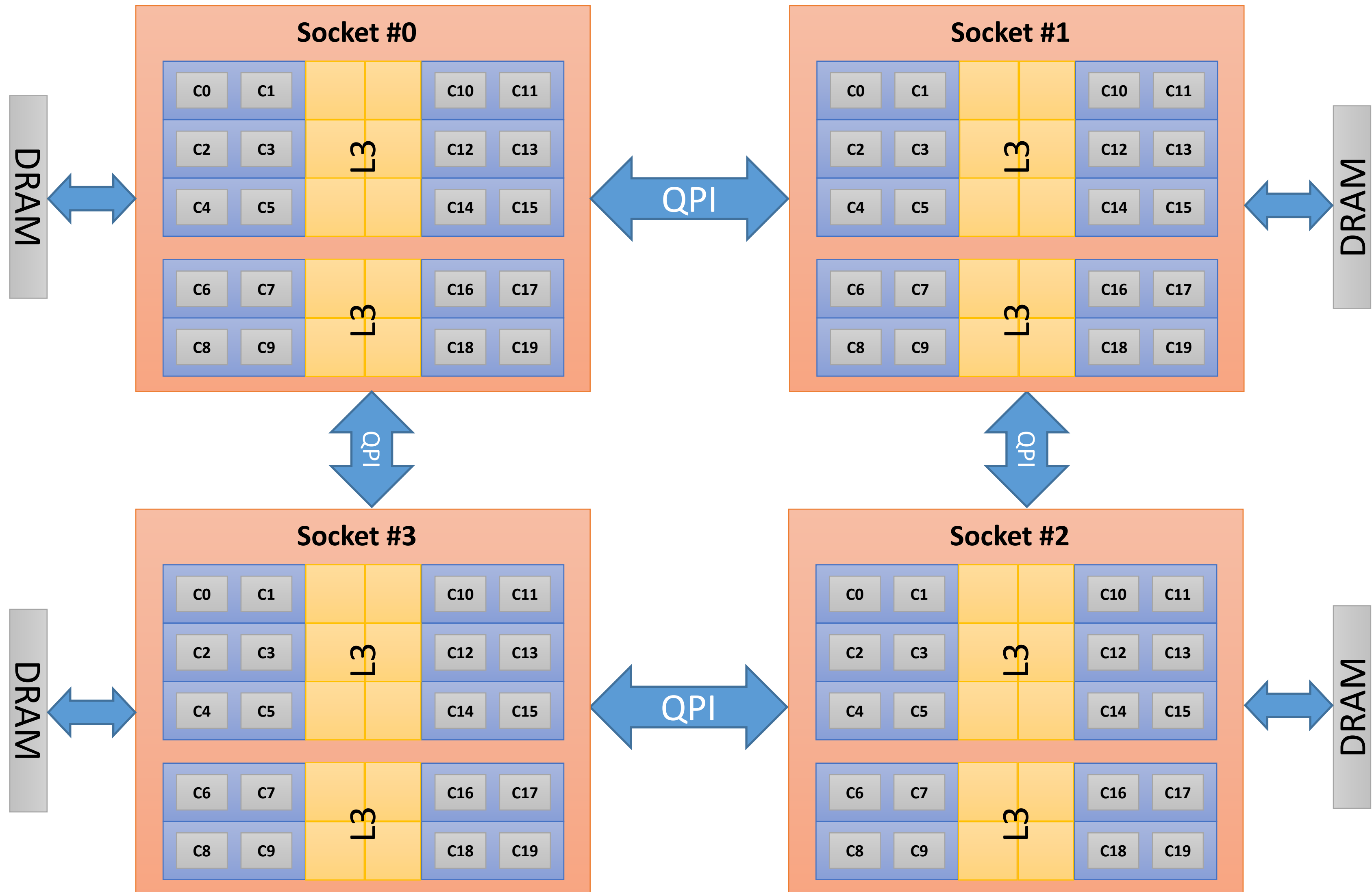
## Performance Study

- we run the previous simple application in a multiprocessor with the following configuration:
  - 4x Intel Xeon CPU E5-4610 v3 @ 1.70GHz → 80 hardware threads
- we calculated the number of FAD operations per sec

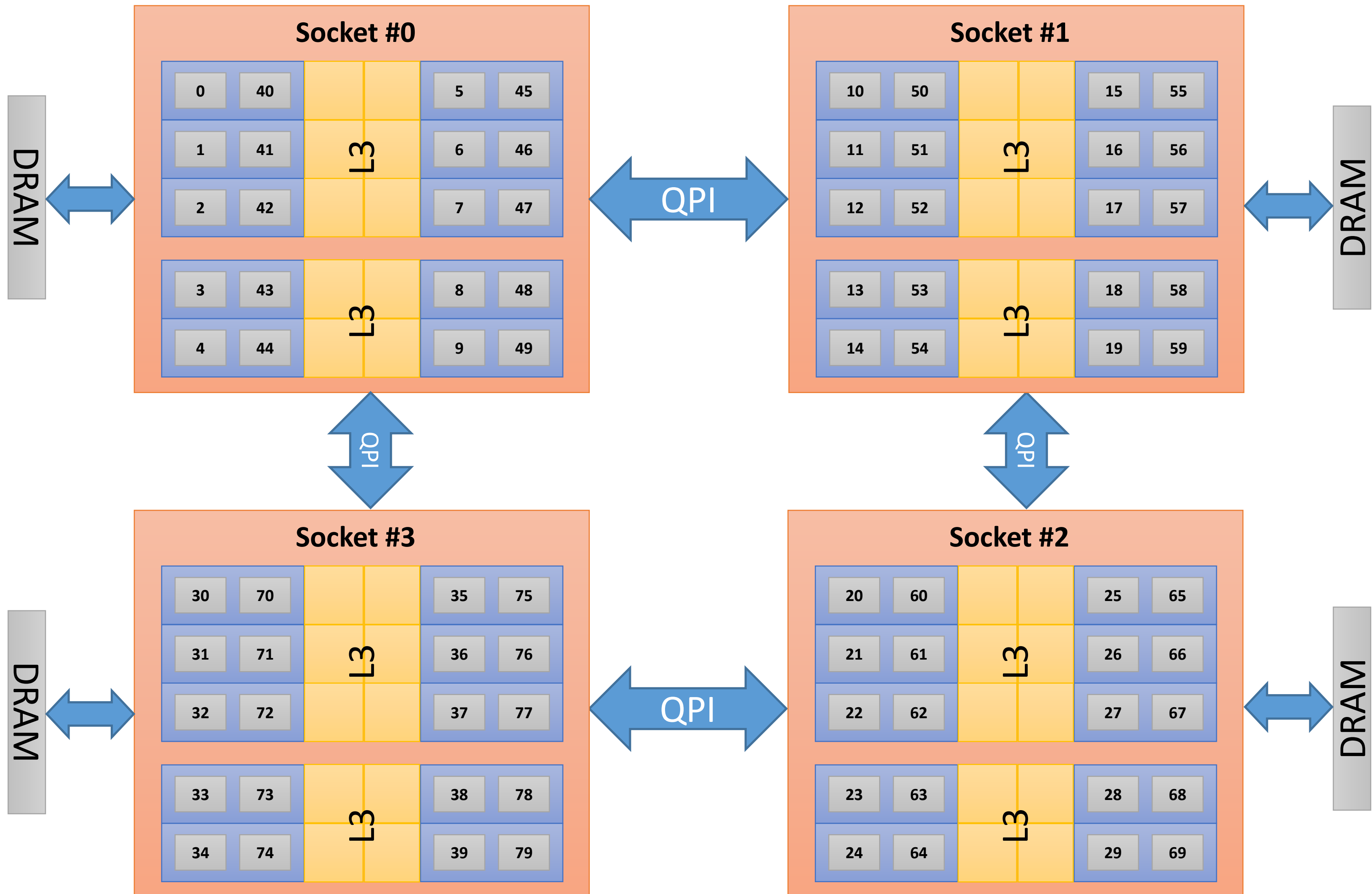
## Results

| Thread A | Thread B | millions FAD operations / second |
|---|---|---|
| Default scheduling/pinning (e.g. cores 42 & 72) | | 9.1 |

# TestBed Interconnect Topology - Actual Perspective



# TestBed Interconnect Topology - Linux Perspective



How can we achieve better performance?

# Affinity on Threads: Performance Impact (II)

## Performance Study

- we run the previous simple application in a multiprocessor with the following configuration:
  - 4x Intel Xeon CPU E5-4610 v3 @ 1.70GHz → 80 hardware threads
- we calculated the number of FAD operations per sec

## Results

| Thread A | Thread B | millions FAD operations / second |
|---|---|---|
| Default scheduling/pinning (e.g. cores 42 & 72) | | 9.1 |
| pinned on core 0 | pinned on core 1 | 13.5 |
| pinned on core 0 | pinned on core 5 | 13.5 |
| pinned on core 0 | pinned on core 8 | 13.5 |
| pinned on core 0 | pinned on core 10 | 9.1 |
| pinned on core 0 | pinned on core 20 | 7.6 |
| pinned on core 0 | pinned on core 30 | 9.1 |
| pinned on core 0 | pinned on core 40 | 12.0 |

# Atomically Incrementing a Shared Counter (II)

```
volatile long counter = 0;      // shared variable to be atomically incremented
```

## MCS

```
void MCS_lock() {
    MyPred = Get&Set(&Tail, MyNode);
    if (MyPred != NULL) {
        MyNode->locked = TRUE;
        MyPred->next = MyNode;
        while (MyNode->locked) noop;
    }
}

void MCS_unlock() {
    if (MyNode->next == NULL) {
        if (CAS(&Tail,MyNode,NULL)==TRUE)
            return;
        while(MyNode->next==NULL)
            noop;
    }
    MyNode->next->locked = FALSE;
    MyNode->next = null;
}
```

```
1.  MCS_lock();
2.  counter = counter + 1;
3.  MCS_unlock();
```

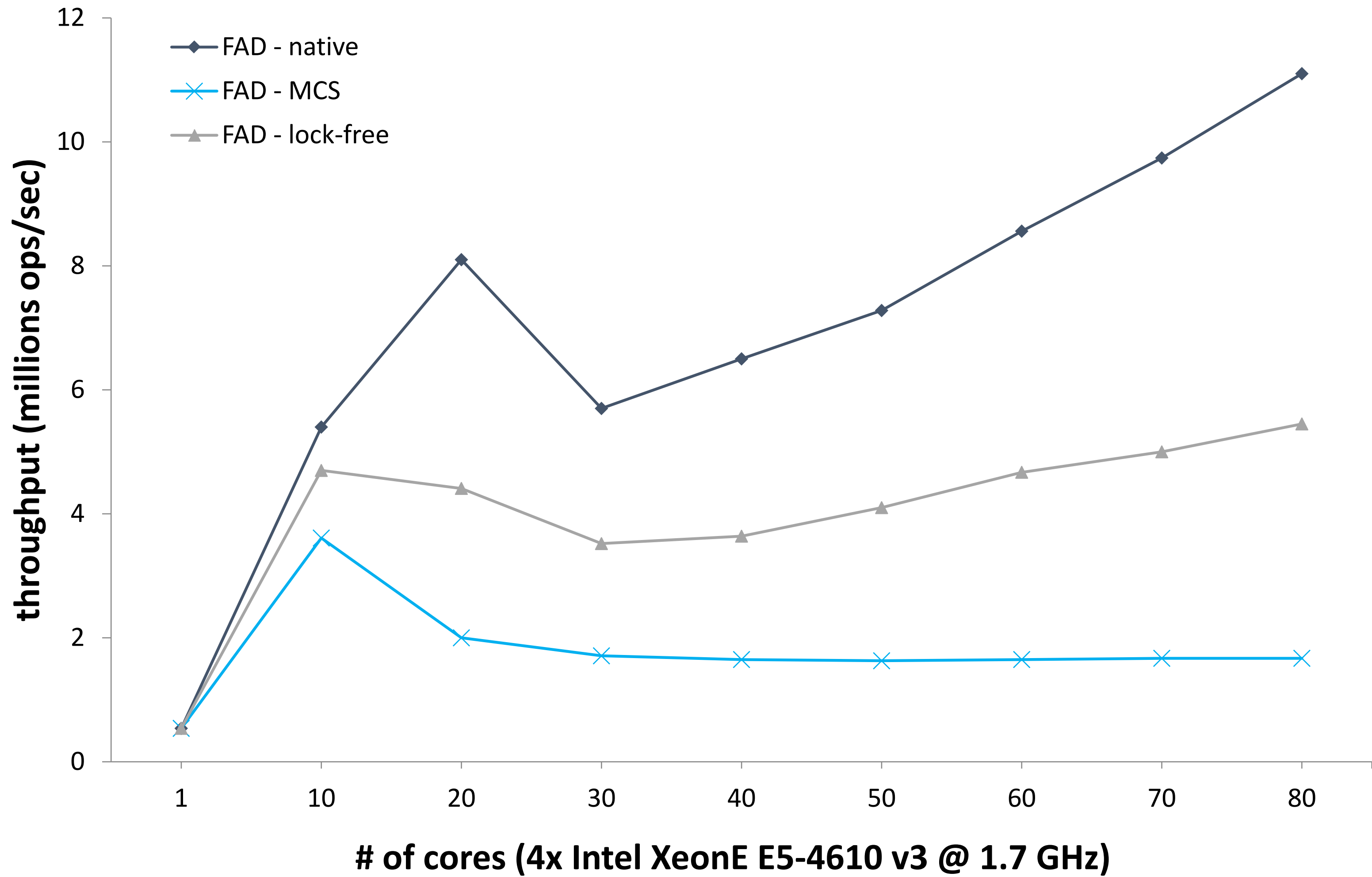
## Lock-Free

```
1.  long cur;
2.
3.  do {
4.      cur = counter;
5.  } while (CAS(&counter,cur,cur+1)==FALSE)
```

## Fetch&Add

```
1. __sync_fetch_and_add(&counter, 1);
```

# Locks vs Lock-Free vs Atomic Instructions (II)



What is the impact of local computation on performance?

# Atomically Incrementing a Shared Counter

with local computation

```
#define LOCAL_WORKLOAD RandomInteger(1, 512)
volatile long counter = 0;        // shared variable to be atomically incremented
```

## MCS

```
1.   void *threadMCS(void *arg) {
2.     volatile int w;
3.
4.     while (counter.value < 10^7) {
5.       MCS_Lock();
6.       counter = counter + 1;
7.       MCS_Unlock();
8.
9.       for (w=0; w<LOCAL_WORKLOAD; w++);
10.    }
11.    return NULL;
12.  }
```

## Lock-Free

```
1.   void *threadLockFree(void *arg) {
2.     volatile int w;
3.     while (counter.value < 10^7) {
4.       long cur;
5.       do {
6.         cur = counter;
7.       } while (CAS(&counter,cur,cur+1)==FALSE);
8.       for (w=0; w<LOCAL_WORKLOAD; w++);
9.     }
10.     return NULL;
11.  }
```

## Fetch&Add

```
1. void *threadFAD(void *arg) {
2.   volatile int w;
3.   while (counter.value < 10^7) {
4.      __sync_fetch_and_add(&x, 1);
5.      for (w=0; w<LOCAL_WORKLOAD; w++);
6.   }
7.   return NULL;
8. }
```

# The Impact of Local Computation