

Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček
jakub.marecek@fel.cvut.cz

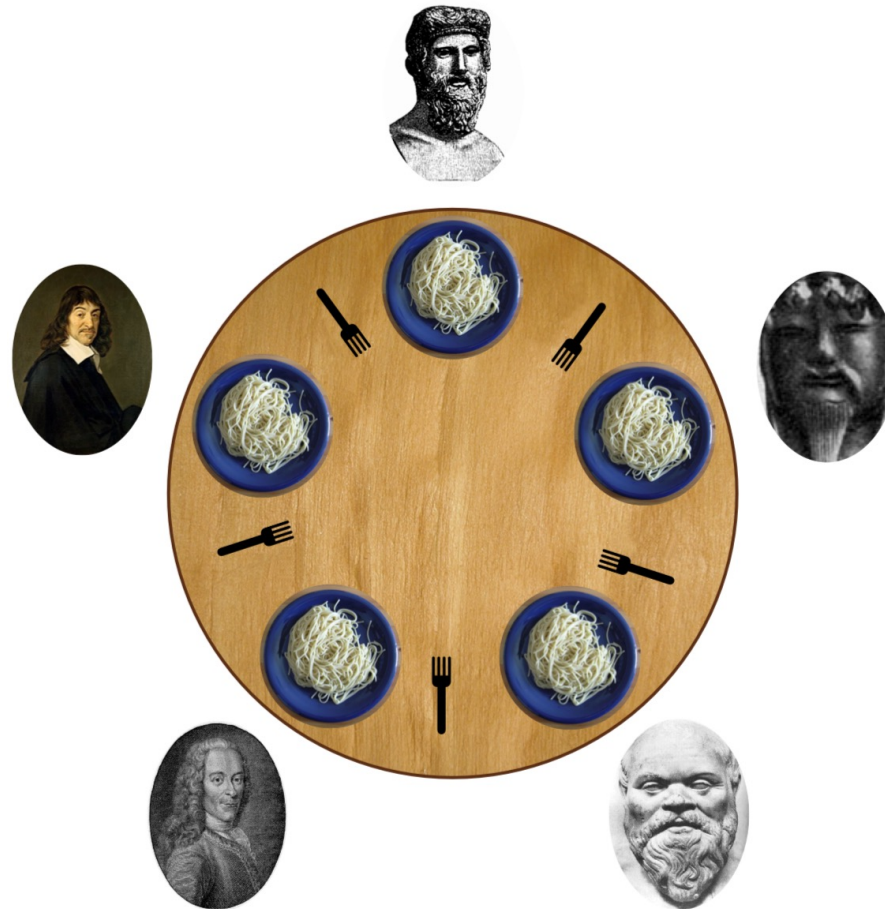
Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

In the previous lecture

- Parallel programming
 - Aim: speeding up computation
 - The influence of various approaches on the speedup
- Distributed programming
 - Aim: consistency across a large number of machines
 - The algorithms in state-of-the-art database engines
- The Promise:
 - A revision from APO and OSY.
 - Some fables.
- CourseWare
 - <https://cw.fel.cvut.cz/wiki/courses/b4b36pdv/start>

The Fables

From the textbook



The Fables

From the textbook

1.2 A Fable

Instead of treating coordination problems (such as mutual exclusion) as programming exercises, we prefer to think of concurrent coordination problems as if they were physics problems. We now present a sequence of fables, illustrating some of the basic problems. Like most authors of fables, we retell stories mostly invented by others (see the Chapter Notes at the end of this chapter).

Alice and Bob are neighbors, and they share a yard. Alice owns a cat and Bob owns a dog. Both pets like to run around in the yard, but (naturally) they do not get along. After some unfortunate experiences, Alice and Bob agree that they should coordinate to make sure that both pets are never in the yard at the same time. Of course, we rule out trivial solutions that do not allow any animals into an empty yard.

How should they do it? Alice and Bob need to agree on mutually compatible procedures for deciding what to do. We call such an agreement a *coordination protocol* (or just a *protocol*, for short).

The yard is large, so Alice cannot simply look out of the window to check whether Bob's dog is present. She could perhaps walk over to Bob's house and knock on the door, but that takes a long time, and what if it rains? Alice might lean out the window and shout "Hey Bob! Can I let the cat out?" The problem is that Bob might not hear her. He could be watching TV, visiting his girlfriend, or out shopping for dog food. They could try to coordinate by cell phone, but the same difficulties arise if Bob is in the shower, driving through a tunnel, or recharging his phone's batteries.

Alice has a clever idea. She sets up one or more empty beer cans on Bob's windowsill (Fig. 1.4), ties a string around each one, and runs the string back to her house. Bob does the same. When she wants to send a signal to Bob, she yanks the string to knock over one of the cans. When Bob notices a can has been knocked over, he resets the can.

Up-ending beer cans by remote control may seem like a creative solution, but it is still deeply flawed. The problem is that Alice can place only a limited number of cans on Bob's windowsill, and sooner or later, she is going to run out of cans to knock over. Granted, Bob resets a can as soon as he notices it has been knocked over, but what if he goes to Cancún for Spring Break? As long as Alice relies on Bob to reset the beer cans, sooner or later, she might run out.

So Alice and Bob try a different approach. Each one sets up a flag pole, easily visible to the other. When Alice wants to release her cat, she does the following:

1. She raises her flag.
2. When Bob's flag is lowered, she unleashes her cat.
3. When her cat comes back, she lowers her flag.

Bob's behavior is a little more complicated.

1. He raises his flag.
2. While Alice's flag is raised
 - a) Bob lowers his flag
 - b) Bob waits until Alice's flag is lowered
 - c) Bob raises his flag
3. As soon as his flag is raised and hers is down, he unleashes his dog.
4. When his dog comes back, he lowers his flag.

The Concepts

- Parallelism means two or more tasks can be executed simultaneously. This is an option, which the compiler and operating system and processor can exercise, but does not come with any guarantees.
- Often, this means no shared variables or other resources, and need not require any synchronization primitives.
- Concurrency means that two or more tasks start, run, and complete in overlapping time periods, while sharing some resources.
- If two tasks concurrently set shared variable x to 1 and 2, it is not clear what value it would have, subsequently.
- More broadly, concurrent access to a mutable shared memory can result in issues without the use of synchronization primitives (data race, problém souběhu) and with the use of synchronization primitives (deadlock, uváznutí).

Data Race

Problém souběhu

When we need to ensure mutual exclusion in access to two or more shared mutable variables, e.g., read value of one of the variables and add it to another variable, we may need to use some synchronization primitives (e.g., mutexes). Without the use of synchronization primitives, we are facing the risk of a data race.

For example, consider the a silly bank without a solid relational database management system, where there are three clients: Alice and Bob and Corporation C.

- Transaction T1: Bob has \$100 in his account, but will be paying a \$50 bill to Corporation C. At the same time, in
- Transaction T2, Alice will be paying \$100 to Bob.

Depending on the ordering of the reading and writing operations, one may obtain several outcomes.

Data Race

Problém souběhu

For example, consider the a silly bank without a solid relational database management system, where there are three clients: Alice and Bob and Corporation C.

- Transaction T1: Bob has \$100 in his account, but will be paying a \$50 bill to Corporation C. At the same time, in
- Transaction T2, Alice will be paying \$100 to Bob.

Depending on the ordering of the reading and writing operations, one may obtain several outcomes:

- Transaction T1 will read \$100 valued of Bob's account. Transaction T2 will read \$100 value. Transaction T2 will write \$200. Transaction T1 will write \$50 value.
- Transaction T1 will read \$100 valued of Bob's account. Transaction T1 will write \$50. Transaction T2 will read \$50 value. Transaction T2 will write \$150 value.
- Transaction T1 will read \$100 valued of Bob's account. Transaction T2 will read \$100 value. Transaction T1 will write \$50. Transaction T2 will write \$200 value.
- Transaction T2 will read \$100 value. Transaction T2 will write \$200 value. Transaction T1 will read \$200 valued of Bob's account. Transaction T1 will write \$150.

Either Bob or the bank could be up to \$100 short.

Deadlock

Problém uváznutí

- When we need to ensure mutual exclusion in access to two or more shared variables, e.g., two temporary results associated with two mutexes, one may naively lock the first mutex first, and subsequently lock the other mutex.
- This, however, can lead to a deadlock.
- Instead, one needs to lock both mutexes at the same time.
- Easily, one could run:

Locking multiple mutexes at once.

```
1 void thread_operation(){
2     std::lock(mutex1,mutex2);
3     ...
4     complicated_task();
5     ...
6     mutex1.unlock();
7     mutex2.unlock();
8 }
```


Deadlock

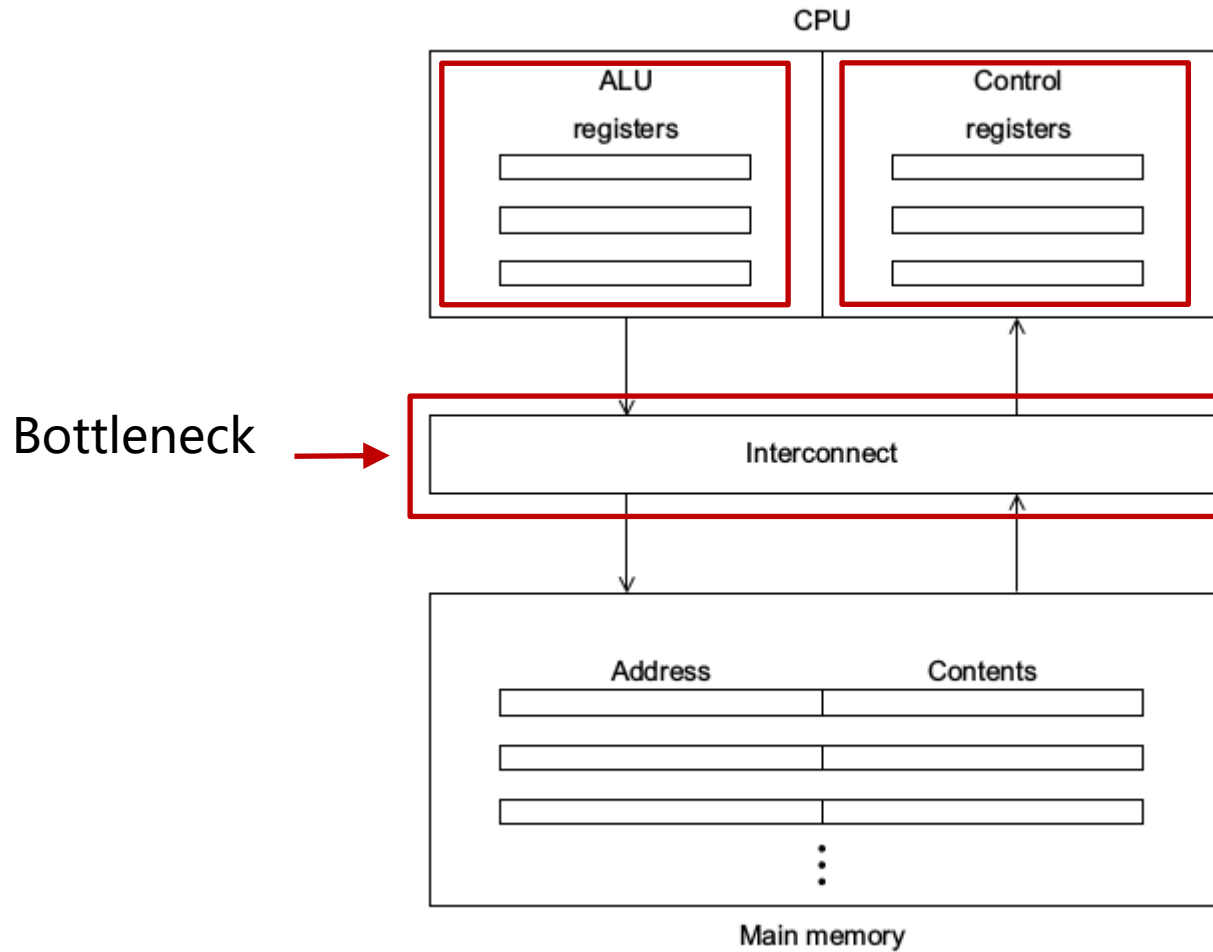
In Theory

In theory, a deadlock (Czech: ``problém uváznutí'') can occur when:

- each lock is owned by one thread
- each thread has locked at least one lock and needs to lock at least one more lock
- it is impossible to remove the lock ownership
- there is a cyclic dependency among the lock-using threads.

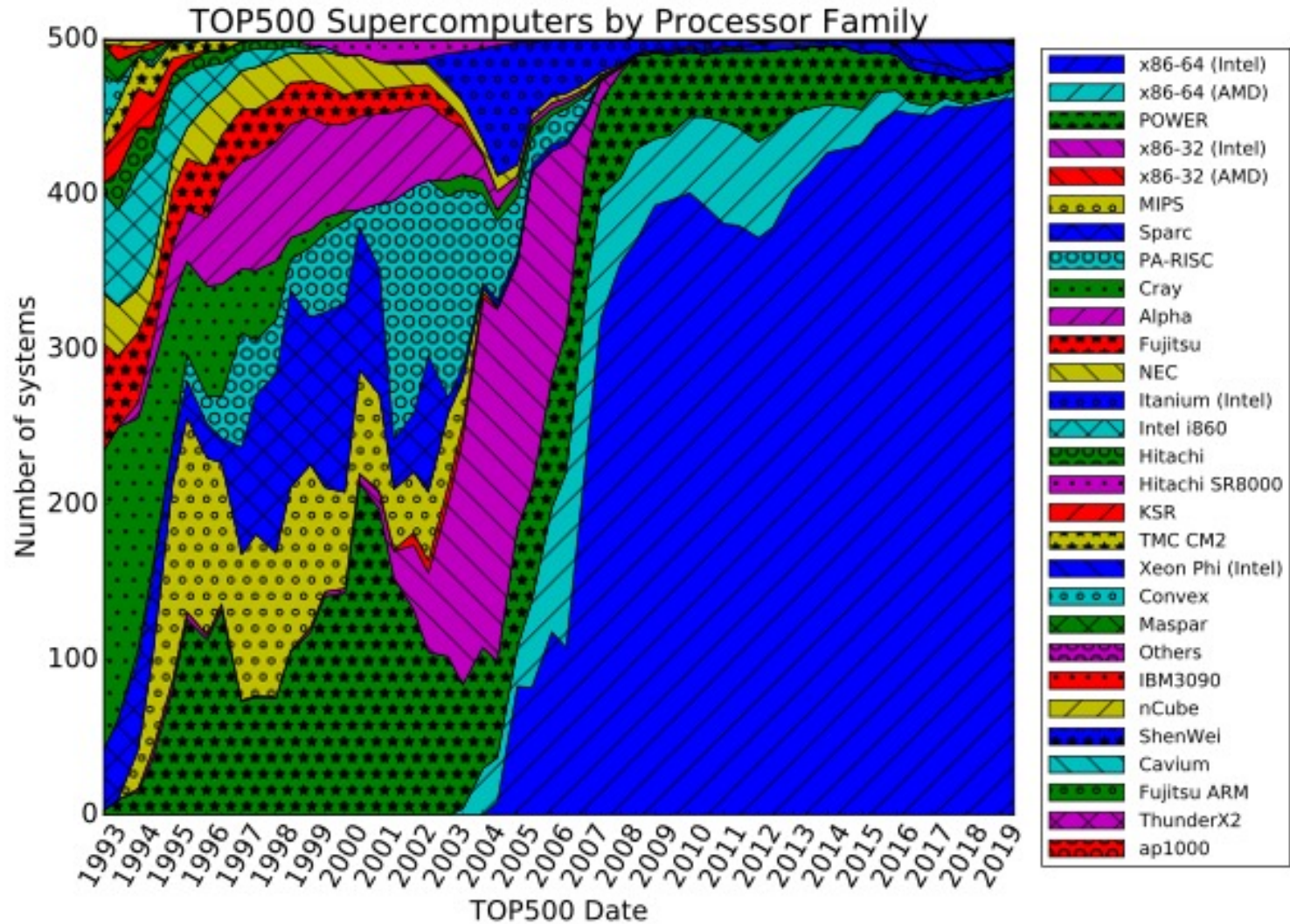
Von Neumann Architecture

The abstraction



Von Neumann Architecture in Practice

TOP 500 Supercomputers



Von Neumann Architecture in Practice

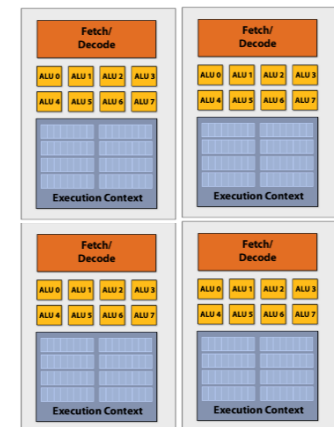
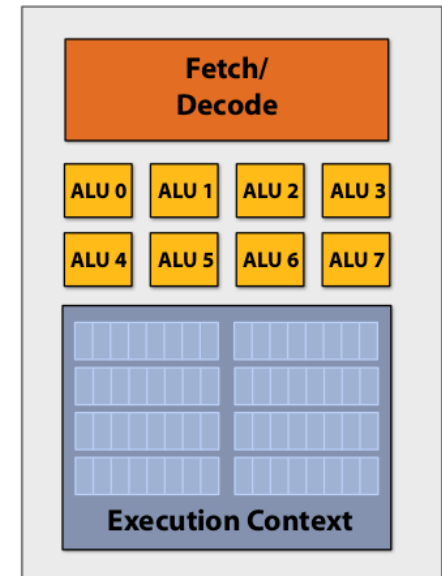
Today



Flynn Taxonomy

The abstraction

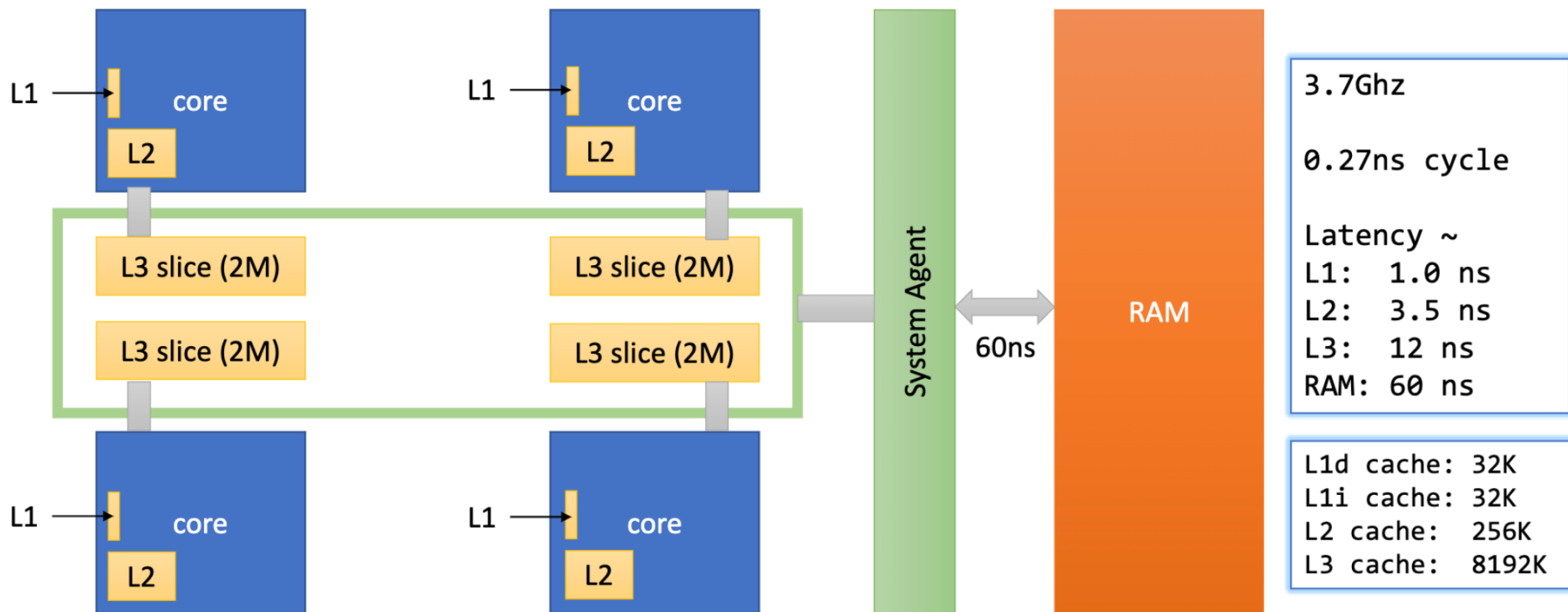
- SIMD (Single Instruction Multiple Data)
 - Single control, multiple ALUs
 - Data parallelism
 - GPGPU, vector instructions (SSE, AVX)
- MIMD (Multiple Instruction Multiple Data)
 - Multi-core processors
 - Multi-processor architectures



Cache Hierarchy

In Practice

Skylake Xeon(R) CPU E3-1505M v5

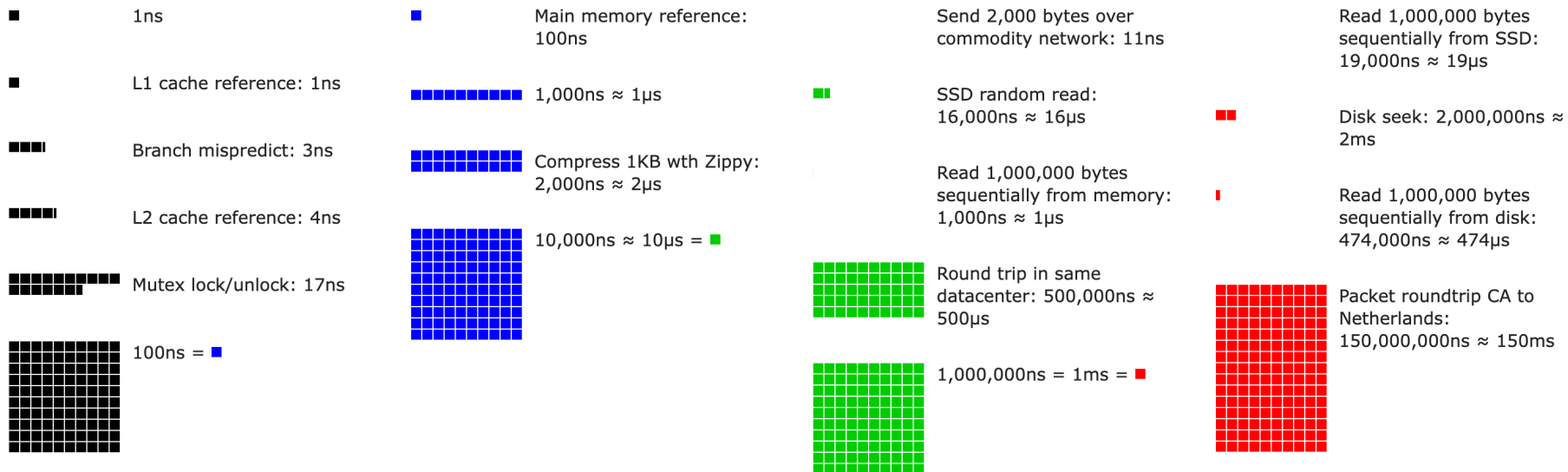


Cache Hierarchy

In Practice

Latency Numbers Every Programmer Should Know

2020



Cache Hierarchy

In Practice

Get 12 weeks for \$29.99 \$6

THE
NEW YORKER

ANNALS OF TECHNOLOGY

THE FRIENDSHIP THAT MADE GOOGLE HUGE

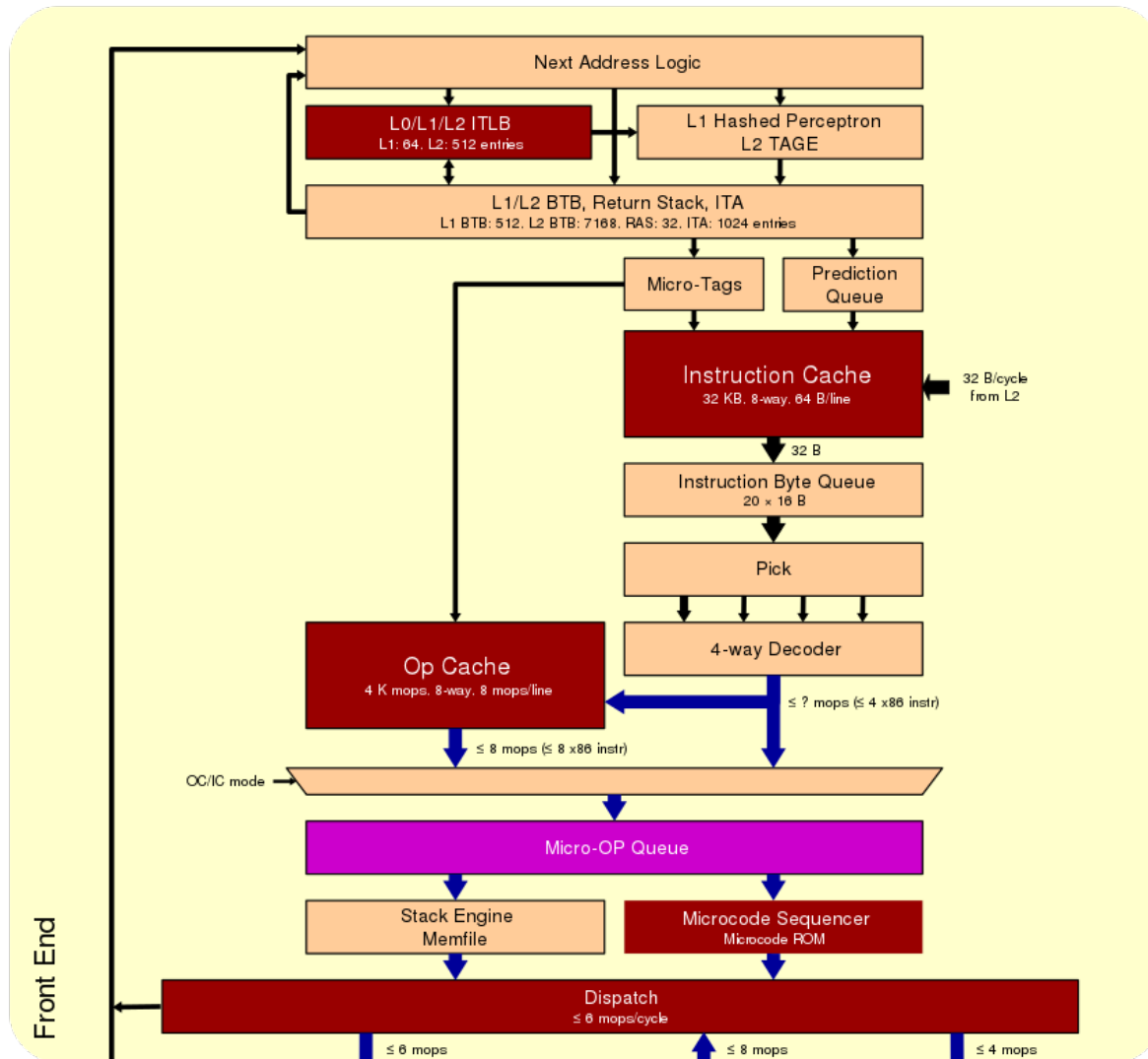
Coding together at the same computer, Jeff Dean and Sanjay Ghemawat changed the course of the company—and the Internet.

By James Somers

December 3, 2018

Cache Hierarchy

In Practice



Instruction-Level Parallelism

Co ještě z APO?

- Instruction-Level Parallelism (ILP).
 - Let us add 2 vectors of floats, each with 1000 elements
 - One addition needs 7 operations:
 - Fetch (načtení)
 - Compare (porovnání exponentů)
 - Shift (posun)
 - Add (součet)
 - Normalize (normalizace)
 - Round (zaokrouhlení)
 - Store (Uložení výsledku)
 - How long does this take?

Instruction-Level Parallelism

Co ještě z APO?

- Instruction-Level Parallelism (ILP).
 - Let us add 2 vectors of floats, each with 1000 elements
 - One addition needs 7 operations:
 - Fetch (načtení)
 - Compare (porovnání exponentů)
 - Shift (posun)
 - Add (součet)
 - Normalize (normalizace)
 - Round (zaokrouhlení)
 - Store (Uložení výsledku)
 - How long does this take?

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Speculative Execution

Superscalar processor

- Multiple instructions in one cycle

- Let us consider a cycle

```
for (i=0; i<1000; i++)  
  z[i]=x[i]+y[i];
```

- One ALU can compute z[0], another z[1], ...

- Speculative execution

```
z = x + y;  
if (z > 0)  
  w = x;  
else  
  w = y;
```

https://cw.fel.cvut.cz/wiki/_media/courses/b35apo/en/lectures/06/b35apo_lecture06-speculative.pdf

Why does this matter?

- How would you implement matrix-vector multiplication?

```
int x[MAXIMUM], int y[MAXIMUM], int A[MAXIMUM*MAXIMUM]
```

Option A

Option B

```
for ( int i = 0; i < MAXIMUM ; i ++)  
  for ( int j = 0; j < MAXIMUM ; j ++)  
    y[i] += A->at(i * MAXIMUM + j)*x[j];
```

```
for ( int j = 0; j < MAXIMUM ; j ++)  
  for ( int i = 0; i < MAXIMUM ; i ++)  
    y [i] += A->at(i * MAXIMUM + j)*x[j];
```

Which is faster?



Why does this matter?

Performance

```
for ( int i = 0; i < MAXIMUM ; i ++)  
  for ( int j = 0; j < MAXIMUM ; j ++)  
    y[i] += A->at(i * MAXIMUM + j)*x[j];
```



```
for ( int j = 0; j < MAXIMUM ; j ++)  
  for ( int i = 0; i < MAXIMUM ; i ++)  
    y [i] += A->at(i * MAXIMUM + j)*x[j];
```



- Arrays are stored sequentially (row-wise)
- Access to A[0][0] accesses one cache line

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

- Write to A[1][0] invalidates cache line

False sharing!

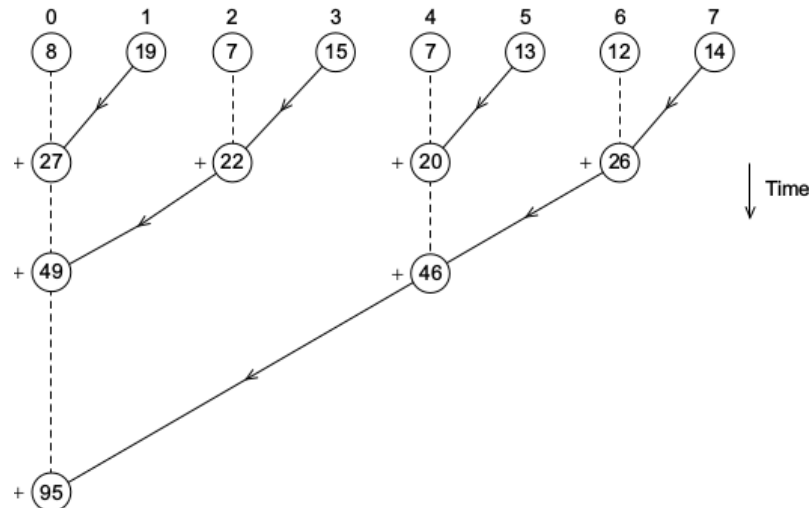
Why does this matter?

Another example

- Let us sum:

0	1	2	3	4	5	6	5×10^9
17	2	9	4	22	0	1			8

How to do this fast?



Why does this matter?

- What if you saw a code like this?

```
long sum(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {  
    for (int i=thread; i<SIZE; i += thread_count)  
        sums[thread] += sqrt(vector_to_sum[i]);  
  
    for (int j=1; j<log2(thread_count)+1; j++) {  
        if ((thread % (int)pow(2,j)) != 0) break;  
        int k = (int)pow(2,j-1);  
        if ((thread + k) >= thread_count) break;  
        if (threads[thread + k].joinable()) threads[thread + k].join();  
        sums[thread] += sums[thread + k];  
    }  
}
```

0	1	2	3	4	5	6	7	8	9
17	2	9	4	22	0	1	0	0	8

False Sharing

Why does this matter?

- How would you aggregate some function outputs?

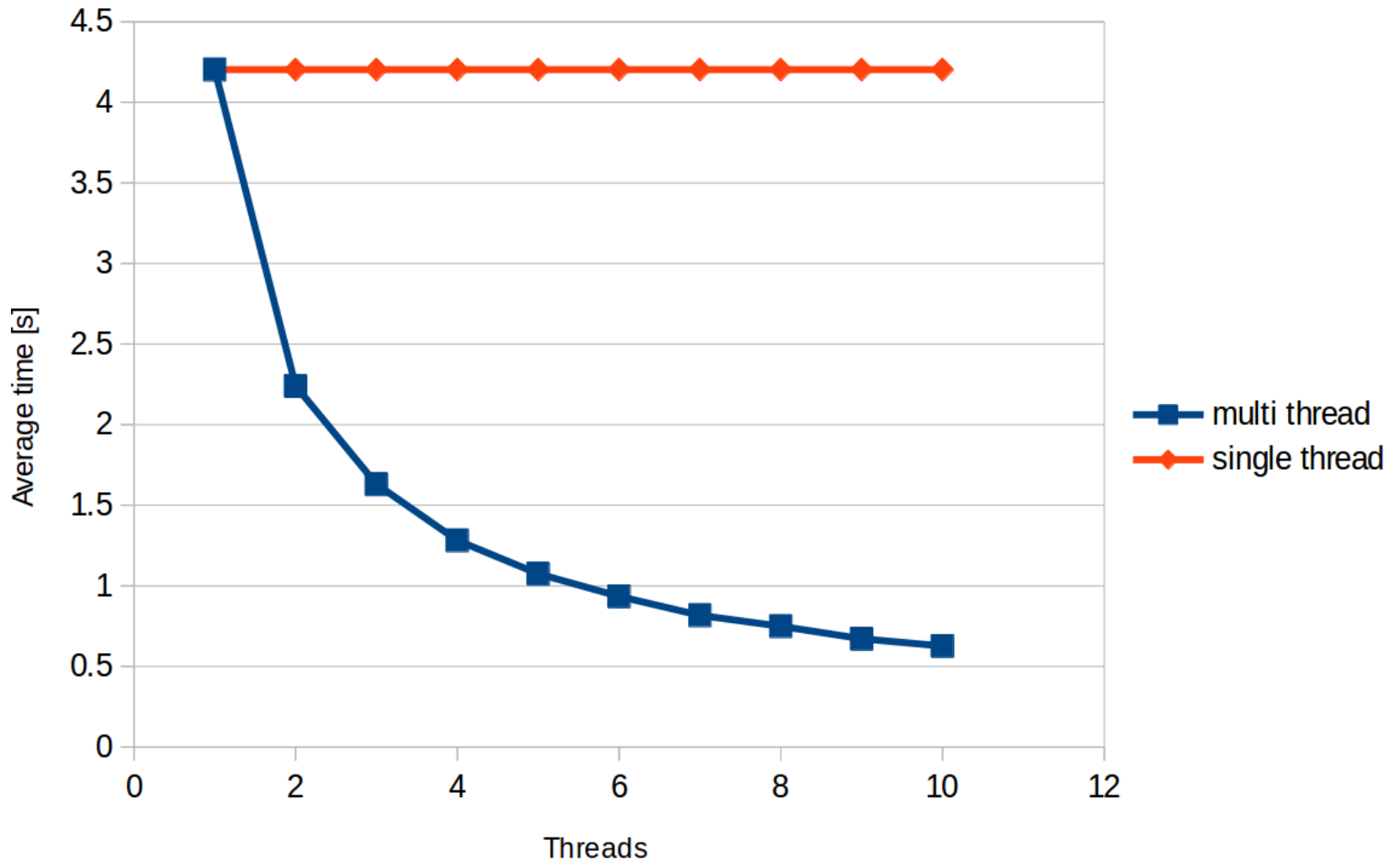
```
long sum_local(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {  
    long local = 0;  
    for (int i=thread; i<SIZE; i += thread_count) {  
        local += sqrt(vector_to_sum[i]);  
    }  
    sums[thread] = local;  
  
    for (int j=1; j<log2(thread_count)+1; j++) {  
        if ((thread % (int)pow(2,j)) != 0) break;  
        int k = (int)pow(2,j-1);  
        if ((thread + k) >= thread_count) break;  
        if (threads[thread + k].joinable()) threads[thread + k].join();  
        local += sums[thread + k];  
    }  
    sums[thread] = local;  
}
```

Introduce a local
variable

Only the local variable is
written in the vector.

Why does this matter?

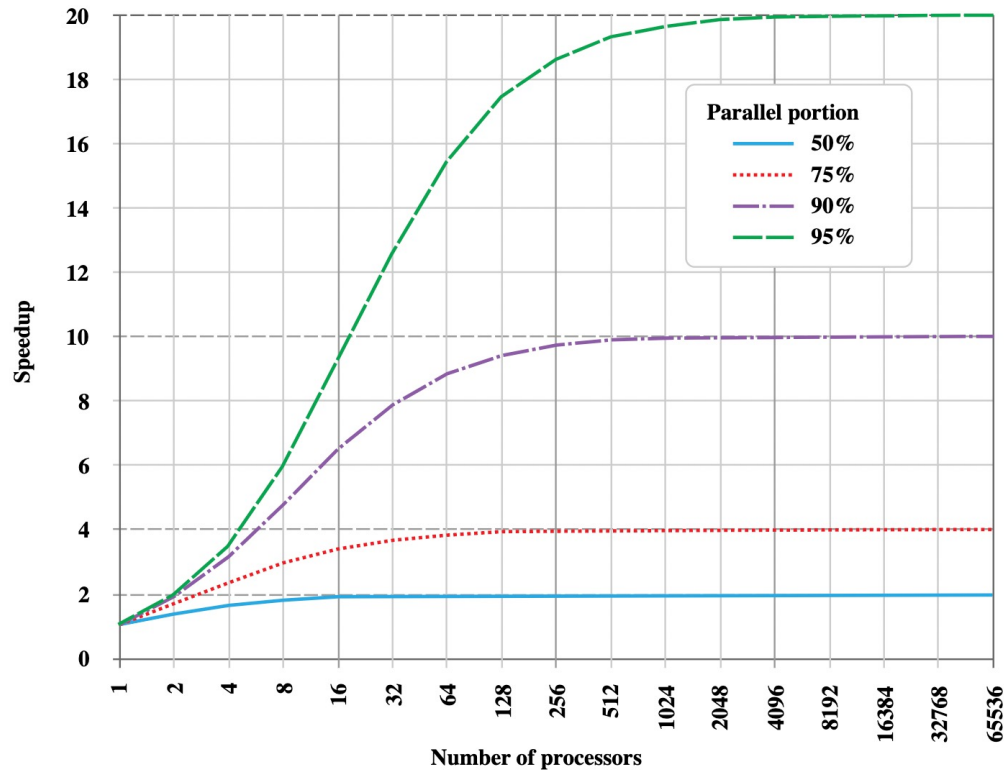
Yes, it does help!



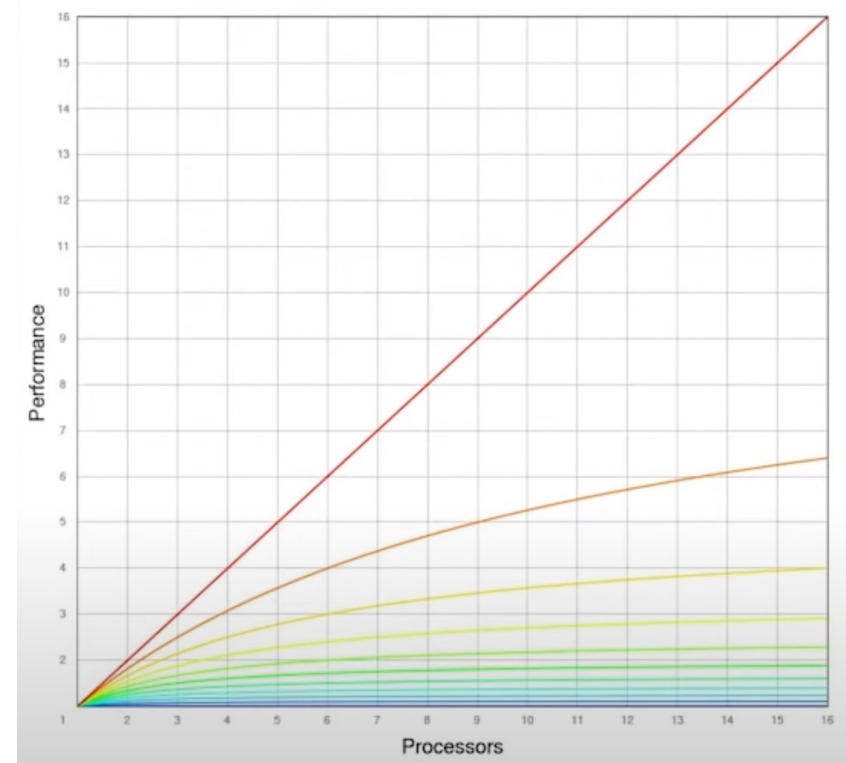
Amdahl's law

In the previous lecture

Log-linear plot for certain proportions of non-parallelizable code.



Linear plot, for multiples of 10% of non-parallelizable code



Grafy z:

- <https://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg>
- <https://www.youtube.com/watch?v=QIHy8pXbnel>

Amdahl's law

In Theory

- There is almost always some overhead in parallel programming (e.g., synchronization primitives)
- There is almost always some nonparallelizable code:
 - Let us consider the speed-up $S = \frac{T_{serial}}{T_{parallel}}$ of the parallel code
 - E.g. if 10% of the code is nonparallelizable and there are p processors (hardware threads):
 - $$S = \frac{T_{serial}}{0.9 \times \frac{T_{serial}}{p} + 0.1 \times T_{serial}} \leq \frac{T_{serial}}{0.1 \times T_{serial}}$$
 - In general, for a fraction n of nonparallelizable code and p processors (hardware threads):
 - $$S = \frac{T_{serial}}{(1-n) \times \frac{T_{serial}}{p} + n \times T_{serial}} \leq \frac{T_{serial}}{n \times T_{serial}}$$

Concurrent programming

The Options

There are two essential models for concurrent programming: shared memory and message passing. In sharing memory, we have broadly four options:

- **Confinement:** Do not share memory between threads. This is often impossible.
- **Immutability:** Do not share any mutable data between threads.
- **Thread-safe code:** Use data types with additional guarantees for storing any mutable data shared between threads, or even better, use implementations of algorithms that are already parallelized and handle the concurrency issues for you.
For example in C++, one can use the standard template library with a suitable execution policy.
In particular, the header `execution` defines objects `std::execution::seq`, `std::execution::par`, `std::execution::par_unseq`, which can be passed as the first argument of any standard algorithm, e.g., `std::vector<int> v`
`std::sort(std::execution::par, v.begin(), v.end());`
- **Synchronization:** Use synchronization primitives to prevent accessing the variable at the same time.

Eventually, we will see that message passing can be implemented using the synchronization primitives and may be the least challenging to use correctly.

Concurrent programming

The Options Revisited

HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent

Feng Niu, Benjamin Recht, Christopher Re, Stephen J. Wright

Stochastic Gradient Descent (SGD) is a popular algorithm that can achieve state-of-the-art performance on a variety of machine learning tasks. Several researchers have recently proposed schemes to parallelize SGD, but all require performance-degrading memory locking and synchronization. This work aims to show using novel theoretical analysis, algorithms, and implementation that SGD can be implemented without any locking. We present an update scheme called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. We show that when the associated optimization problem is sparse, meaning most gradient updates only modify small parts of the decision variable, then HOGWILD! achieves a nearly optimal rate of convergence. We demonstrate experimentally that HOGWILD! outperforms alternative schemes that use locking by an order of magnitude.

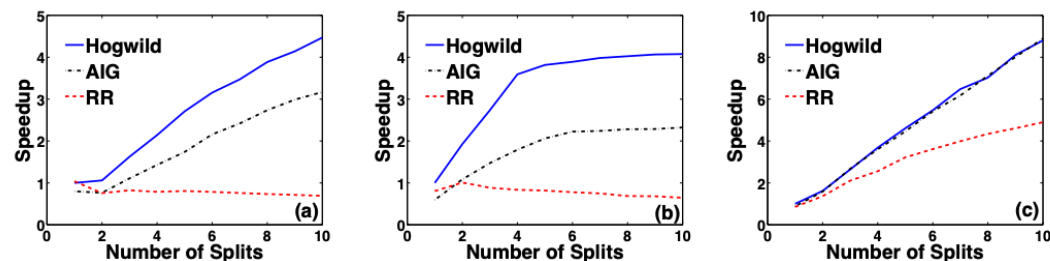


Figure 2: Total CPU time versus number of threads for (a) RCV1, (b) Abdomen, and (c) DBLife.

Lock-free approaches more broadly:

https://www.youtube.com/watch?v=Yl8Or0afcfg&ab_channel=ChurchillCompSciTalks

Concurrent programming

The Options Revisited

Intel® Transactional Synchronization Extensions (Intel® TSX) is perhaps one of the most non-trivial extensions of instruction set architecture introduced in the 4th generation Intel® Core™ microarchitecture code name Haswell. Intel® TSX implements hardware support for a best-effort “transactional memory”, which is a simpler mechanism for *scalable* thread synchronization as opposed to inherently complex fine-grained locking or lock-free algorithms. The extensions have two interfaces: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

In this blog I will show how you can write your first RTM code and execute it in an emulated environment now, without waiting until the 4th generation Intel® Core™ processors become available for purchase.

Before diving in, please make sure you have a basic understanding of the new RTM instructions. I refer you to [this blog](#) as an introduction. Check out also the Intel Developer Forum’12 presentation by Ravi Rajwar & Martin Dixon discussing the details Intel TSX implementation in Haswell hardware and a presentation by Andi Kleen on [adding lock elision \(also using RTM\) to Linux](#).

My plan was to write a toy bank account processing application using popular C++ thread-unaware data structures from STL with concurrent access to bank records managed by Intel TSX. This way the implementation should be very simple, thread-saf and scalable.

<https://www.intel.com/content/www/us/en/developer/articles/community/exploring-tsx-with-software-development-emulator.html>

<https://www.blackhat.com/docs/us-16/materials/us-16-Jang-Breaking-Kernel-Address-Space-Layout-Randomization-KASLR-With-Intel-TSX-wp.pdf>

Breaking Kernel Address Space Layout Randomization with Intel TSX

Yeongjin Jang, Sangho Lee, and Taesoo Kim
Georgia Institute of Technology

Abstract

Kernel hardening has been an important topic, as many applications and security mechanisms often consider the kernel as part of their Trusted Computing Base (TCB). Among various hardening techniques, Kernel Address Space Layout Randomization (KASLR) is the most effective and widely adopted defense mechanism that can practically mitigate various memory corruption vulnerabilities, such as buffer overflow and use-after-free. In principle, KASLR is secure as long as no memory leak vulnerability exists and high entropy is ensured.

In this paper, we introduce a novel timing attack against KASLR, called DrK, that can precisely de-randomize the memory layout of the kernel without violating any such assumptions. DrK exploits a hardware feature called Intel Transactional Synchronization Extension (TSX) that is readily available in most modern commodity CPUs. One surprising behavior of TSX, which is essentially the root cause of this security loophole, is that it aborts a transaction without notifying the underlying kernel even when

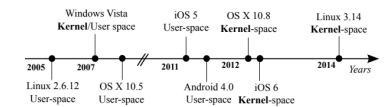


Figure 1: The adoption status of both user-space and kernel-space ASLR in popular operating systems, ordered by year [62].

1 Introduction

Enhancing the security of operating systems (OSes) has been an active, important research topic as the underlying OS is commonly considered to be the Trusted Computing Base (TCB) for user applications and their security mechanisms. Among various hardening techniques, Kernel Address Space Layout Randomization (KASLR) is

Concurrent programming

The Options Revisited

Following the side-channel timing attack on TSX, Intel disabled TSX on all processors released until January 2023 (!).

Only since Sapphire Rapids (e.g. Intel® Xeon® W-3400), there is support for restricted transactional memory in selected Intel processors.

Gracemont	Gracemont	Intel 7	Golden Cove	Golden Cove (Alder Lake) Raptor Cove (Raptor Lake)
Crestmont	Crestmont	Intel 4	Redwood Cove	Meteor Lake
Skymont	Skymont	Intel 20A	Lion Cove	<i>Arrow Lake</i>
TBA	TBA	Intel 18A/20A	TBA	<i>Lunar Lake</i>

Lock-free approaches more broadly:

https://www.youtube.com/watch?v=Yl8Or0afcfg&ab_channel=ChurchillCompSciTalks

Pthreads vs. C++ vs. OpenMP vs. SYCL

Co znáte z OSY (pthreads)

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

const int thread_count = 10;
void* Hello(void* rank);

int main(int argc, char* argv[]) {
    long thread;
    pthread_t *thread_handles;
    thread_handles = (pthread_t*)malloc(thread_count * sizeof(pthread_t));
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL,
                      Hello, (void *) thread);
    printf("Hello from the main thread\n");
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);

    free(thread_handles);
    return 0;
}

void* Hello(void* rank) {
    long my_rank = (long) rank;
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);
    return NULL;
}
```

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (OpenMP)

```
#include <iostream>
#include <vector>
#include "omp.h"

const int thread_count = 10;

void Hello() {
    int my_rank = omp_get_thread_num();
    int threads = omp_get_num_threads();
    std::cout << "Hello from thread " << my_rank << " of " << threads << std::endl;
}

int main(int argc, char* argv[]) {
    #pragma omp parallel num_threads(thread_count)
    Hello();
    return 0;
}
```

- nutno překládat s přepínačem `-fopenmp`
 - (např. `g++ -fopenmp openmp-hello.cpp -o openmp-hello`)

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (C++11)

```
#include <iostream>
#include <thread>
#include <vector>

const int thread_count = 10;
void Hello(long my_rank);

int main(int argc, char* argv[]) {
    std::vector<std::thread> threads;
    for (int thread=0; thread < thread_count; thread++) {
        threads.push_back(std::thread(Hello, thread));
    }

    std::cout << "Hello from the main thread\n";

    for (int thread=0; thread < thread_count; thread++) {
        threads[thread].join();
    }

    return 0;
}

void Hello(long my_rank) {
    std::cout << "Hello from thread " << my_rank << " of " << thread_count << std::endl;
}
```

Nicolai Josuttis: "it is almost impossible to use it easily and right"

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (C++20)

```
#include <iostream>
#include <thread>
#include <vector>

const int thread_count = 10;
void Hello(long my_rank);

int main(int argc, char* argv[]) {
    std::vector<std::thread> threads;
    for (int thread=0; thread < thread_count; thread++) {
        threads.push_back(std::jthread(Hello, thread));
    }

    std::cout << "Hello from the main thread\n";

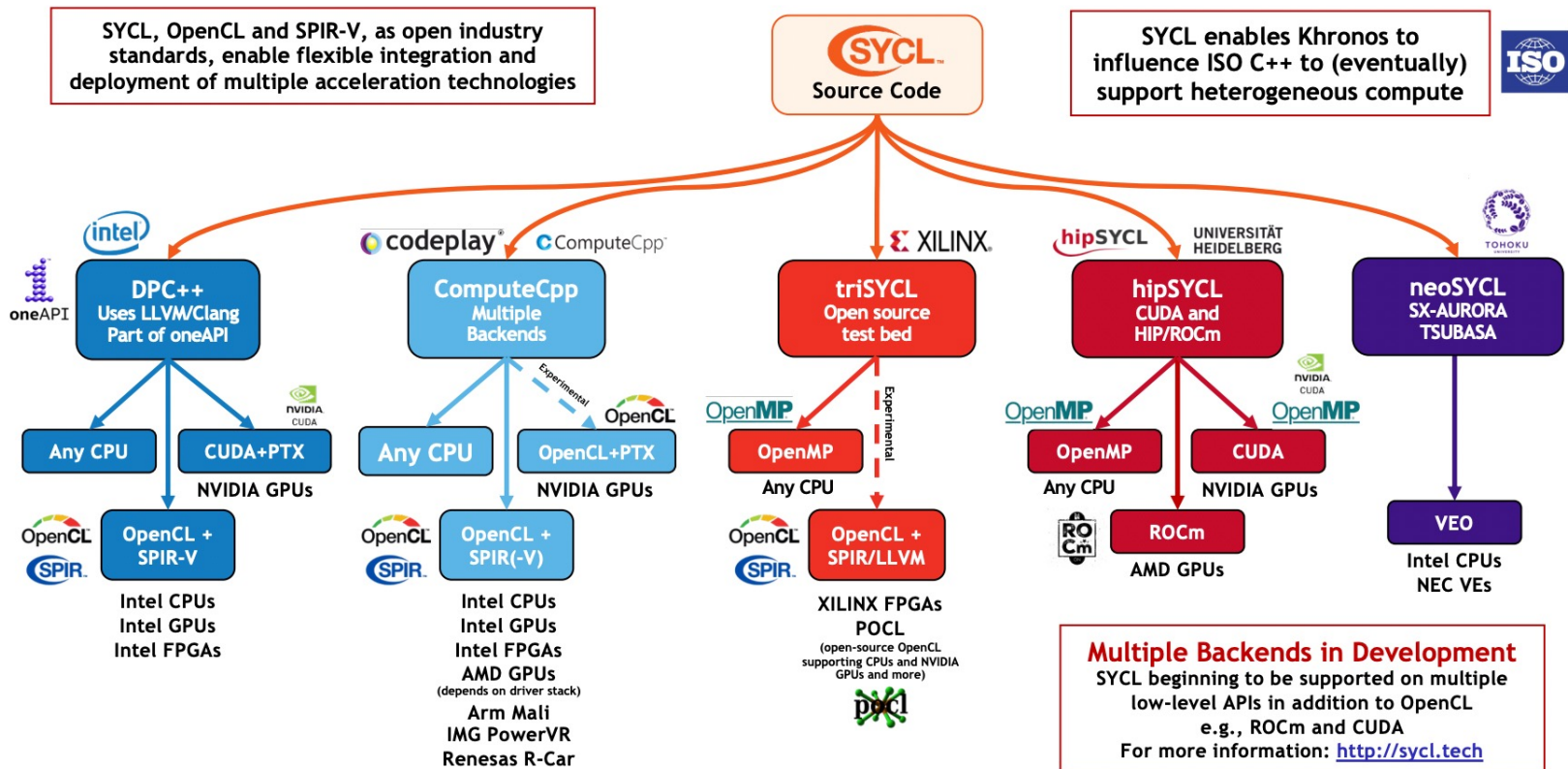
    return 0;
}

void Hello(long my_rank) {
    std::cout << "Hello from thread " << my_rank << " of " << thread_count << std::endl;
}
```

C++ spec was approved in 2020 and GCC has a good support since v. 11.
(There is GCC 12 in the labs.)

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (SYCL)



Viz <https://www.khronos.org/assets/uploads/developers/presentations/SYCL-2020-Launch-Feb21.pdf>

Pthreads vs. C++ vs. OpenMP vs. SYCL

Ochutnávka (SYCL)

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;
const int nElems = 64u;

class assign_elements;

int main() {
    int data[nElems] = {0};
    try {
        default_selector selector;
        queue myQueue(selector, [](exception_list l) {
            for (auto ep : l) {
                try {
                    std::rethrow_exception(ep);
                } catch (const exception& e) {
                    std::cout << "Asynchronous exception caught:\n" << e.what();
                }
            }
        });
        buffer<int, 1> buf(data, range<1>(nElems));
        myQueue.submit([&](handler& cgh) {
            auto ptr = buf.get_access<access::mode::read_write>(cgh);
            auto myRange = nd_range<1>(range<1>(nElems), range<1>(nElems / 4));
            auto myKernel = ([=](nd_item<1> item) {
                ptr[item.get_global_id()] = item.get_global_id()[0];
            });
            cgh.parallel_for<assign_elements>(myRange, myKernel);
        });
    } catch (const exception& e) {
        std::cout << "Synchronous exception caught:\n" << e.what();
        return 2;
    }
    return 0;
}
```

Structuring code

Processes, Threads, Tasks, Coroutines

- Processes, threads, tasks, and coroutines execute instructions.
- A **process** provides all of the prerequisites for executing instructions: loads an executable program, sets up a virtual address space, sets up the environment (e.g. environment variables and a security context), sets up the process control block (PCB, often stored in registers of the processor and on a per-process stack in kernel memory), opens handles to system objects (e.g., files, sockets), and often much more.
- In some sense, one can imagine "a virtual machine".
- All modern operating systems (OS) are multitasking, i.e., running multiple processes with the operating system forcibly interrupting the run one process to execute another process after a certain amount of time ("preemptive scheduling"). Switching between the processes involves swapping the process control block (PCB). In Intel architectures, this is known as the task state segment (TSS), and there is hardware support for the switch. AMD64 does not support task switches in hardware.

Structuring code

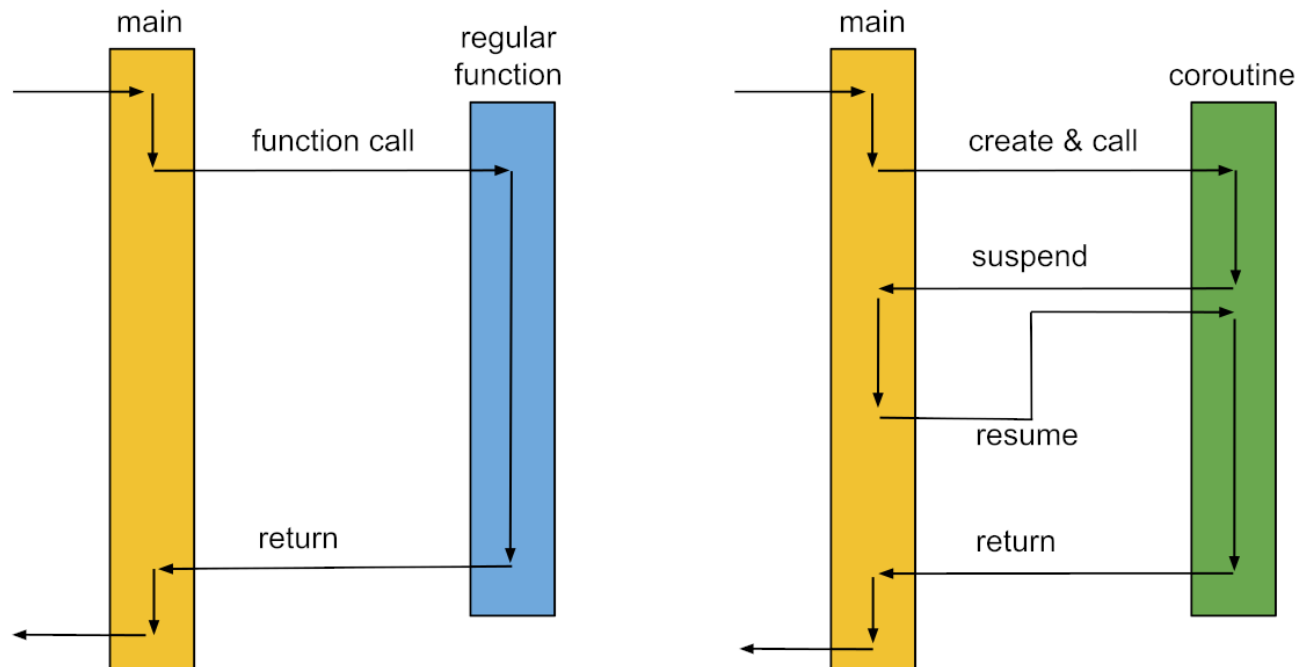
Processes, Threads, Tasks, Coroutines

- Within a particular process, there is at least one **thread**. All threads of a particular process share the same virtual address space and handles to system objects. Each thread, independently, operates its own context (registers, stack, exception handlers).
- Unless declared otherwise, threads of a particular process share memory and are allocated "time slices" by the operating system.
- This can be seen as a "virtual processor" within a "virtual machine" of a process, often with no guarantees on the time slicing.
- Most modern processors are multi-core and support multithreading in some form. This means that each process can execute multiple "**hardware threads**" and there is some support for switching between those. In Intel architectures, hyper-threading means each hardware core can execute multiple threads, e.g., two, to take advantage of idle time (e.g., loading data, network communications).

Structuring code

Processes, Threads, Tasks, Coroutines

- Within a particular thread, one may utilize multiple **coroutines**, which can be seen as subroutines that can run in multiple steps, but sometimes can serve as a light-weight alternative to hardware threads.



Structuring code

Processes, Threads, Tasks, Coroutines

- Within a particular thread, one may utilize multiple **coroutines**, which can be seen as subroutines that can run in multiple steps, but sometimes can serve as a light-weight alternative to hardware threads. Coroutines can be called, can return when completed, but also can suspend themselves, yielding control and partial results, and be resumed by another co-routine. Typical uses involve generators and factories and various other concepts within "lazy evaluation", as well as event-driven architectures within cooperative multi-tasking.
- That is: two coroutines within one thread never run in parallel, but one can have the runs of two or more coroutines interleaved. We can suspend a co-routine in one thread and resume it within another thread.
- As it turns out, the "context switch" with user-level threads has a similar cost to a function call or suspending a coroutine (**co_yield**). Indeed, coroutines are typically implemented with user-level threads, which leads to cheaper context-switch compared with hardware threads. Within the user-level threads, one can distinguish stackful and stackless versions, where coroutine state is saved on the heap (as in C++).

Structuring code

Processes, Threads, Tasks, Coroutines

- Coroutines can be called, can return when completed, but also can suspend themselves, yielding control and partial results, and be resumed by another co-routine.
- Typical uses involve generators and factories and various other concepts within "lazy evaluation", as well as event-driven architectures within cooperative multi-tasking.
- That is: two coroutines within one thread never run in parallel, but one can have the runs of two or more coroutines interleaved. We can suspend a co-routine in one thread and resume it within another thread.
- As it turns out, the "context switch" with user-level threads has a similar cost to a function call or suspending a coroutine (**co_yield**). Indeed, coroutines are typically implemented with user-level threads, which leads to cheaper context-switch compared with hardware threads. Within the user-level threads, one can distinguish stackful and stackless versions, where coroutine state is saved on the heap (as in C++).

Structuring code

Processes, Threads, Tasks, Coroutines

```
4 #include <coroutine>
5 #include <iostream>
6
7 // The caller-level type
8 struct Generator {
9     // The coroutine level type
10    struct promise_type {
11        using Handle = std::coroutine_handle<promise_type>;
12
13        Generator get_return_object() {
14            return Generator{Handle::from_promise(*this)};
15        }
16        std::suspend_always initial_suspend() { return {}; }
17        std::suspend_always final_suspend() noexcept { return
            → {}; }
18        std::suspend_always yield_value(int value) {
19            current_value = value;
20            return {};
21        }
22        void unhandled_exception() { }
23        int current_value;
24    };
25
26    explicit Generator(promise_type::Handle coro) :
27        → coro_(coro) {}
28    // Make move-only
29    Generator(const Generator&) = delete;
30    Generator& operator=(const Generator&) = delete;
31    Generator(Generator&& t) noexcept : coro_(t.coro_) {
32        → t.coro_ = {}; }
33
34    Generator& operator=(Generator&& t) noexcept {
35        if (this == &t) return *this;
36        if (coro_) coro_.destroy();
37        coro_ = t.coro_;
38        t.coro_ = {};
39        return *this;
40    }
41
42    int get_next() {
43        coro_.resume();
44        return coro_.promise().current_value;
45    }
46
47 private:
48    promise_type::Handle coro_;
49 };
```

```
Generator myCoroutine() {
    int x = 0;
    while (true) {
        co_yield x++;
    }
}
```

```
int main() {
    auto c = myCoroutine();
    int x = 0;
    while ((x = c.get_next()) < 10) {
        std::cout << x << "\n";
    }
}
```

Structuring code

Processes, Threads, Tasks, Coroutines

An example of the use of coroutines, which currently does not compile in GCC 12.2.

```
1 #include <coroutine>
2 #include <generator>
3 #include <iostream>
4 #include <syncstream>
5
6 std::generator<int> work() {
7     for (int i = 0; i < 10; i++) {
8         co_yield i;
9     }
10 }
11
12 int main() {
13     for (int i : work()) {
14         std::osyncstream(std::cout) << i << '\n';
15     }
16 }
```

[Open in Compiler Explorer](#)

Structuring code

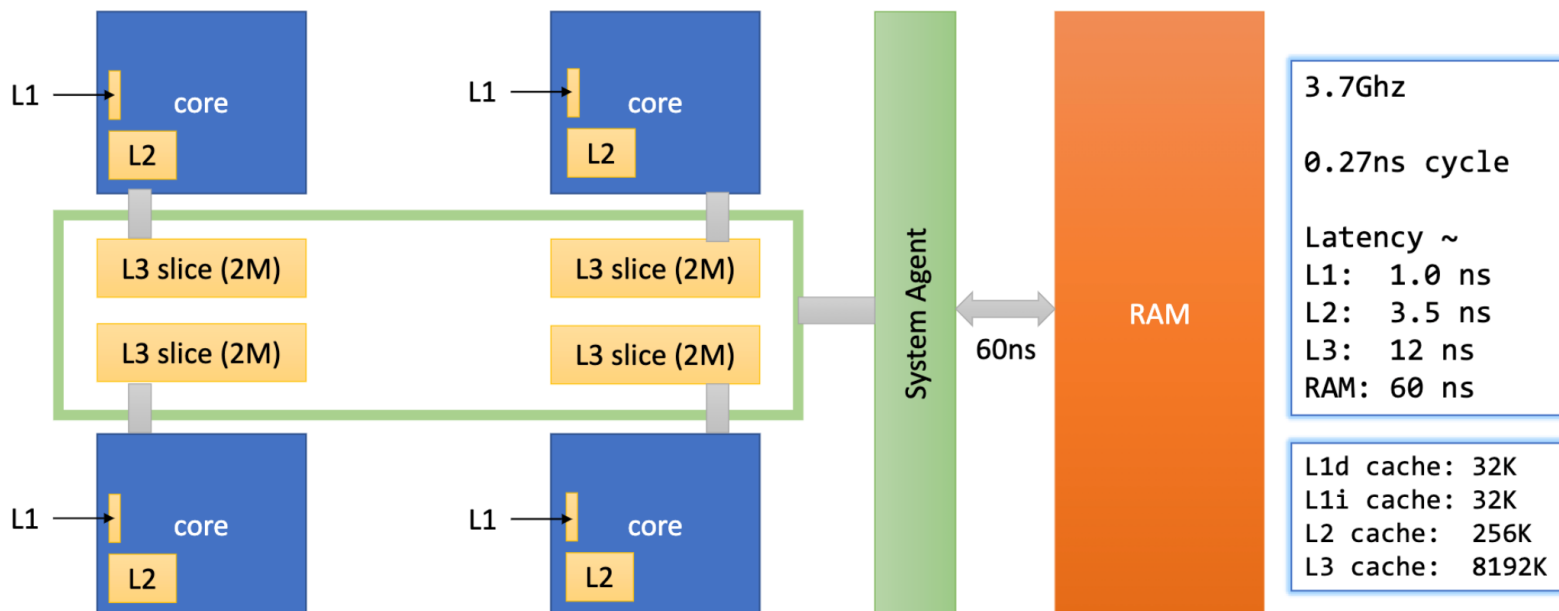
Processes, Threads, Tasks, Coroutines

- A **task** is a rather abstract unit of work, e.g., a function, which can be executed by any thread, but often allocated to one of a many threads within a pool.

Memory order

- First, one should like to understand several options for implementing synchronization primitives, known as memory orders. All guarantee atomicity and modification-order consistency.

Skylake Xeon(R) CPU E3-1505M v5



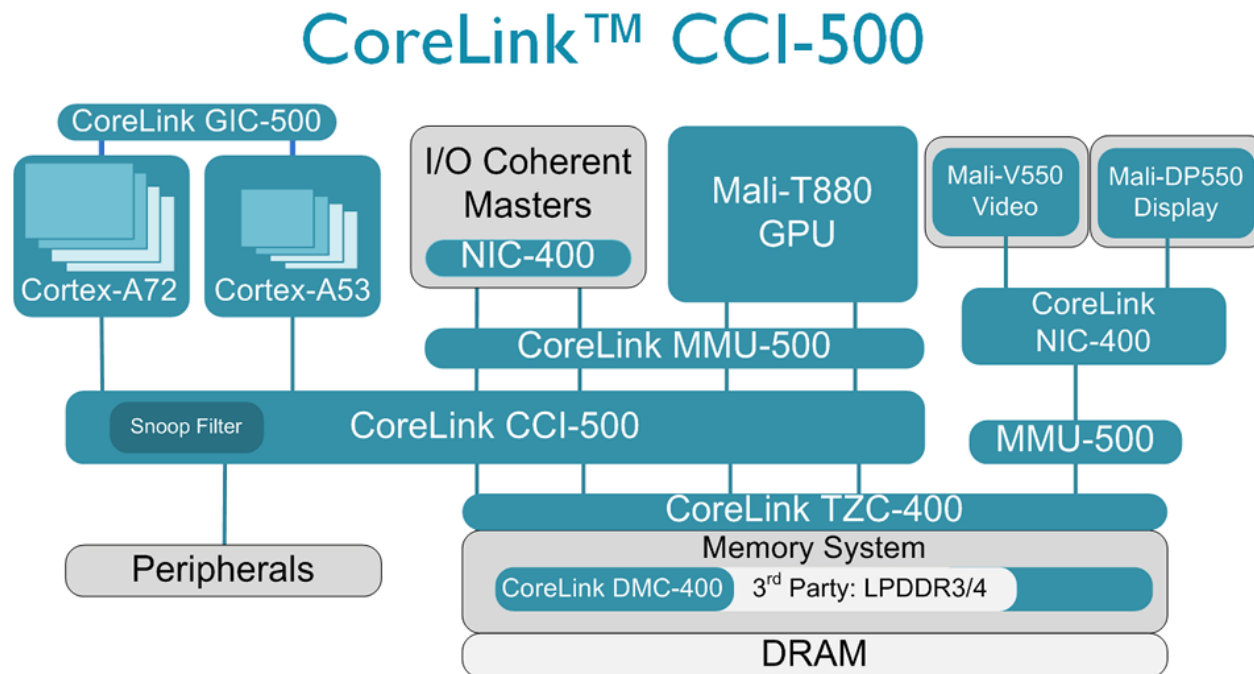
Memory order

- First, one should like to understand several options for implementing synchronization primitives, known as memory orders. All guarantee atomicity and modification-order consistency.



Memory order

- First, one should like to understand several options for implementing synchronization primitives, known as memory orders. All guarantee atomicity and modification-order consistency.



Memory order

- Let us focus on ARM in particular:

The ARMv8 architecture employs a **weakly-ordered** model of memory. In general terms, this means that

- the order of memory accesses is not required to be the same as the program order for load and store operations.
- The processor is able to re-order memory read operations with respect to each other.
- Writes may also be re-ordered (for example, write combining).

As a result, hardware optimizations, such as the use of cache and write buffer, function in a way that improves the performance of the processor, which means that the required bandwidth between the processor and external memory can be reduced and the long latencies associated with such external memory accesses are hidden.

<https://developer.arm.com/documentation/den0024/a/Memory-Ordering?lang=en>

Memory order

In C++11

- In **memory_order_relaxed**, no further guarantees are provided and specifically no order is imposed on concurrent memory accesses. This is also how weakly-ordered architectures (e.g. ARM) operate, by default: if two threads access shared memory the load in one thread does not have to read a value written by another thread very recently.
- With **memory_order_release** and `memory_order_acquire` specifiers, we force weakly-ordered architectures to behave closer to strongly-ordered architectures (e.g., Intel). If one thread writes into shared memory atomically with **memory_order_release** and another thread reads the memory atomically with `memory_order_acquire`, the load in the second thread is guaranteed to read the value written by another thread.
- With **memory_order_seq_cst**, we additionally require a single total ordering of all modifications (with this specifier). A load with this specifier gets its value either from the last store with this specifier or from some store without this specifier that did not precede the most recent **memory_order_seq_cst store**. This is the default option.

Compare and swap

In General

Synchronization primitives are typically implemented using some hardware instructions, typically **compare-and-swap**. In locking, these make it possible to test whether the lock is free, and if so, acquire the lock within a single operation that the hardware guarantees to execute atomically.

The atomic compare and swap (CAS) instruction compares the value of an atomic variable against a given value. If there is a match, CAS stores a given new value in the atomic variable. That is:

- we declare an atomic variable (and a pointer to it)
- (*) we save the value of an atomic variable to a local, private variable (by dereferencing the pointer)
- based on the saved value in a local, private variable, we compute the new value, which we would like to store in the atomic variable
- the CAS instruction is used. If the current value matches the value saved in the local, private variable, we will overwrite the value with the newly computed value. If the current value no longer matches the value saved in the local, private variable, we wait (some random and growing from a small starting value) and repeat from (*).

Compare and swap

In C++

In C++, the **atomic** header defines two variants of "compare and swap" and a specialization thereof for pointers:

- `bool compare_exchange_weak(_Tp& __e, _Tp __i, memory_order __s, memory_order __f) noexcept`
- `bool compare_exchange_strong(_Tp& __e, _Tp __i, memory_order __s, memory_order __f) noexcept`

Both are called with the desired value **e**, the new value **i**, and the memory orders to consider if there is a match and if there is no match.

Typically, if there is a match and we want to replace the value, we may use **std::memory_order_release**. If there is no match, we are just reading the value and **std::memory_order_acquire** would suffice. In the latter variant, we pass two pointers.

Compare and Swap

Weak and strong variants

- The difference between the **weak** and **strong** variant is in that the weak variant may return false even if there is a match, in certain cases, but can be much faster in certain architectures. This notably entails ARM architectures (RISC-V and MIPS), where the weak variant will be implemented using the so called load-link/store-conditional pair of instructions (load exclusive register / ldxr and store exclusive register / stxr in ARM version 8). These are much faster than the comparable instructions issuing a barrier (ldaxr/stlaxr in ARM version 8).
- All four ARM instructions utilize only two registers, compared to three registers for CAS proper in Intel architectures (Compare and exchange / cmpxchg since 80486 and cmpxchg8b and cmpxchg16b since Intel Core 2). On recent Intel and AMD processors, cmpxchg is only marginally slower than a non-cached load.

Memory order

In C++11

If you want to understand memory orders in more detail:

- See <https://arxiv.org/abs/1803.04432>

[Submitted on 12 Mar 2018]

Memory Models for C/C++ Programmers

[Manuel Pöter](#), [Jesper Larsson Träff](#)

The memory model is the crux of the concurrency semantics of shared-memory systems. It defines the possible values that a read operation is allowed to return for any given set of write operations performed by a concurrent program, thereby defining the basic semantics of shared variables. It is therefore impossible to meaningfully reason about a program or any part of the programming language implementation without an unambiguous memory model.

This note provides a brief introduction into the topic of memory models, explaining why it is essential for concurrent programs and covering well known memory models from sequential consistency to those of the x86 and ARM/POWER CPUs. Section 4 is fully dedicated to the C++11 memory model, explaining how it can be used to write concurrent code that is not only correct and portable, but also efficient by utilizing the relaxed memory models of modern architectures.

- See also:
https://www.youtube.com/watch?v=A_vAG6LIHwQ&ab_channel=ACCUConference

Synchronization primitives

- Synchronization primitives make it possible to synchronize or restrict access of multiple threads to some resources (e.g., global variables, file handles, sockets). You can use them as an interface, without knowing their implementation.
- **Raw synchronization primitives:** Lock, Mutex, Semaphore, Atomic, Memory Fence, Condition Variable are synchronization primitives, which make it possible to synchronize or restrict access of multiple threads to some resources.
- Lock is a very general term for a synchronization primitive. Mutexes are usually used by one thread only, while semaphores are shared between multiple threads.
- The binary semaphore is the most simple type of a lock, which provides exclusive access for both reading and writing.
- The counting semaphore limits the use of a single resource by at most a given number of threads.
- A spinlock, the thread simply waits ("spins") until the lock becomes available. This is efficient if threads are blocked for a short time, because it avoids the overhead of operating system process re-scheduling. It is inefficient if the lock is held for a long time, or if the progress of the thread that is holding the lock depends on preemption of the locked thread.

Synchronization primitives

In C++

- In C++, the **only** synchronization primitive that is guaranteed to be hardware implemented is a particular atomic boolean type, which is known as **std::atomic_flag**.
- Unlike all specializations of **std::atomic**, it is guaranteed to be lock-free.
- Prior to C++20, it has been very restricted, because there was no way to check the value of **std::atomic_flag** without setting it. C++20 adds method **test()**.

Synchronization primitives

And how to implement them

A silly implementation of a spin lock.

```
1 // based on https://en.cppreference.com/w/cpp/atomic/atomic_flag
2
3 class SpinLock {
4     std::atomic_flag locked = ATOMIC_FLAG_INIT ;
5 public:
6     void lock() {
7         while (locked.test_and_set(std::memory_order_acquire)) {
8             #if defined(__cpp_lib_atomic_flag_test)
9                 while (locked.test(std::memory_order_relaxed))
10            #endif
11                ;
12        }
13    }
14    void unlock() {
15        locked.clear(std::memory_order_release);
16    }
17};
```

Open in Compiler Explorer [↗](#)

Further features

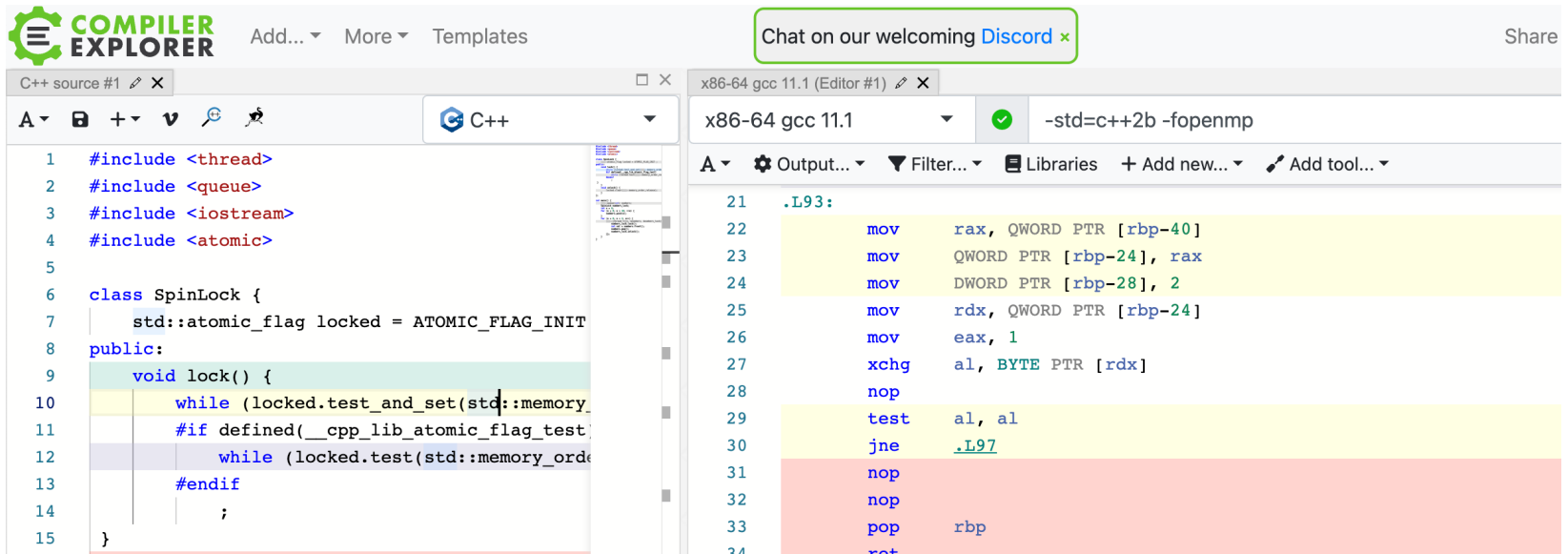
In C++23

Further synchronization features

- Fences help order non-atomic and atomic memory accesses, without any associated operations. On Intel architectures (including x86-64), **atomic_thread_fence** do not issue any instructions, except **std::atomic_thread_fence(std::memory_order::seq_cst)**.
- Barrier provides a thread-coordination mechanism that blocks a group of threads until all threads in that group have reached the barrier. Such a barrier can be used repeatedly to wait until a number of threads have finished their operations.
- Latch and is a downward counter, whose initial value is initialized and then threads may block on the latch until the counter is zero. One thread may decrement a latch multiple times, but no thread can increment the latch. Thus, it serves as a single-use barrier.
- We will also see synchronized output streams. The synchronized buffer is flushed only when the destructor of the synchronized buffer is called, but provides for guarantees of atomicity for the access. (That is, **std::endl** and **std::flush** no longer flush!)

Debugging

<https://godbolt.org/>



The image shows a screenshot of the Compiler Explorer interface. On the left, the C++ source code for a `SpinLock` class is displayed. The code includes headers for `<thread>`, `<queue>`, `<iostream>`, and `<atomic>`. The `lock()` method is shown with a `while` loop that uses `std::memory_order` and `std::atomic_flag` to implement a spin lock. On the right, the assembly output for the `lock()` method is shown, starting at label `.L93:`. The assembly includes instructions for moving pointers, setting values, testing, and jumping. A green box highlights a link to a Discord chat: "Chat on our welcoming [Discord](#)".

```
1 #include <thread>
2 #include <queue>
3 #include <iostream>
4 #include <atomic>
5
6 class SpinLock {
7     std::atomic_flag locked = ATOMIC_FLAG_INIT
8 public:
9     void lock() {
10         while (locked.test_and_set(std::memory_order_acquire))
11             #if defined(__cpp_lib_atomic_flag_test_order)
12                 while (locked.test(std::memory_order_acquire))
13             #endif
14             ;
15     }
```

```
21 .L93:
22     mov     rax, QWORD PTR [rbp-40]
23     mov     QWORD PTR [rbp-24], rax
24     mov     DWORD PTR [rbp-28], 2
25     mov     rdx, QWORD PTR [rbp-24]
26     mov     eax, 1
27     xchg   al, BYTE PTR [rdx]
28     nop
29     test   al, al
30     jne   .L97
31     nop
32     nop
33     pop   rbp
34     ret
```

<https://godbolt.org/z/cEdE7r5fq>

Debugging

<https://godbolt.org/>



Add... More Templates

Chat on our welcoming [Discord](#)

Share Policies

C++ source #1

x86-64 gcc 11.1 (Editor #1)



C++

x86-64 gcc 11.1

-std=c++2b -fopenmp

```
1 #include <mutex>
2 #include <thread>
3 #include <iostream>
4 #include <syncstream>
5
6 std::mutex m1;
7 std::mutex m2;
8
9 void f(int id) {
10     std::lock(m1, m2);
11     std::lock_guard<std::mutex> lock1(m1, std::adopt_lock);
12     std::lock_guard<std::mutex> lock2(m2, std::adopt_lock);
13 }
14
15 int main(int argc, char* argv[]) {
16     std::jthread t1(f, 1);
17     std::jthread t2(f, 2);
18 }
19
```

```
A Output... Filter... Libraries + Add new... Add tool...
1 m1:
2     .zero    40
3 m2:
4     .zero    40
5 f(int):
6     push    rbp
7     mov     rbp, rsp
8     sub     rsp, 32
9     mov     DWORD PTR [rbp-20], edi
10    mov     esi, OFFSET FLAT:m2
11    mov     edi, OFFSET FLAT:m1
12    call    void std::lock<std::mutex, std::mutex>(std::mutex&, std::mutex&)
13    lea     rax, [rbp-8]
14    mov     esi, OFFSET FLAT:m1
15    mov     rdi, rax
16    call    std::lock_guard<std::mutex>::lock_guard(std::mutex&, std::adopt_lo
17    lea     rax, [rbp-16]
18    mov     esi, OFFSET FLAT:m2
19    mov     rdi, rax
20    call    std::lock_guard<std::mutex>::lock_guard(std::mutex&, std::adopt_lo
21    lea     rax, [rbp-16]
22    mov     rdi, rax
23    call    std::lock_guard<std::mutex>::~~lock_guard() [complete object destruct
24    lea     rax, [rbp-8]
25    mov     rdi, rax
```

Output (0/0) x86-64 gcc 11.1 i - 3071ms (500953B) ~29010 lines filtered Compiler License

Debugging

<https://godbolt.org/>

The image shows the Godbolt online compiler interface. On the left, a C++ code editor displays the following code:

```
1 #include <iostream>
2
3 struct A
4 {
5     void foo() {std::cout << "1\n";}
6
7     template <typename T = int>
8     void foo() {std::cout << "2\n";}
9 };
10
11 int main()
12 {
13     A x;
14     x.template foo();
15 }
```

The code defines a struct `A` with two `foo()` methods: one for `int` (outputting "1\n") and one for a generic type `T` (outputting "2\n"). The `main` function creates an instance `x` of `A` and calls `x.template foo();`, which is underlined with a red squiggly line, indicating a compiler warning or error.

On the right, the execution results are shown for three different compilers:

- x86-64 gcc 11.2**: Program returned: 0, Program stdout: 1. Execution time: - 1264ms.
- x86-64 clang 13.0.0**: Program returned: 0, Program stdout: 2. Execution time: - 1639ms.
- x64 msvc v19.0 (WINE)**: Program returned: 0, Program stdout: 2. Execution time: - 1639ms.

The interface also shows various settings like "Wrap lines", "Libraries", and "Compilation" options.

Debugging

<https://clang.llvm.org/docs/ThreadSanitizer.html>

- <https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual>

```
$ clang++ simple_race.cc -fsanitize=thread -fPIE -pie -g
$ ./a.out
=====
WARNING: ThreadSanitizer: data race (pid=26327)
  Write of size 4 at 0x7f89554701d0 by thread T1:
    #0 Thread1(void*) simple_race.cc:8 (exe+0x000000006e66)

  Previous write of size 4 at 0x7f89554701d0 by thread T2:
    #0 Thread2(void*) simple_race.cc:13 (exe+0x000000006ed6)

  Thread T1 (tid=26328, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:19 (exe+0x000000006f39)

  Thread T2 (tid=26329, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:20 (exe+0x000000006f63)

=====
ThreadSanitizer: reported 1 warnings
```

