The Multicore Programming Course in Lund

Welcome to the Multicore Programming course in Lund

- 12 lectures
- 6 labs on parallel programming — work in groups of two.
- During the labs you will make different implementations of an algorithm (dataflow analysis) on different platforms:
  1. Java
  2. Scala with Akka
  3. C with Pthreads (two parts)
- There will also be labs on atomic types in C++, and lock-free data structures
- The first and third might look similar but they are not!
- There is also a project and a competition which consists of chosing one of these versions and to tune as much as you can — also in groups of two.

Administration

- Lecturer is `Jonas.Skeppstedt@cs.lth.se` with office E:2190
- Office hours during LP1: every weekday at 12.30 - 13.00
- Lectures at 10 on Mondays and Thursdays
- Labs start next week.
- Course web page `http://cs.lth.se/edan26`
- You will get an account on the machine **power.cs.lth.se** — a 4 processor POWER/Linux machine 2.5 GHz
- You can work on other machines if you wish but performance measurements are to be done on it.
- You can access it with **ssh -Y user@power.cs.lth.se**...

Contents of the course

Introduction to Multicore Programming

Contents of Lecture 1

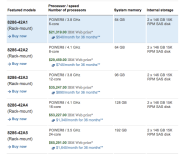
- Advantages with multithreading
  - Performance
  - Sometimes simpler programming
- Disadvantages with multithreading
  - Overhead
  - Usually more complex programming
- Parallel programming models
  - Message passing to compute nodes with private memories — e.g. the Cell (Sony PS3)
  - Shared memory with coherent caches — most common machine type
  - In Scala we will use message passing built on shared memory

Multicore Programming

- Parallel programming is much more fun than sequential programming!
- It's really cool to see many threads speeding up our code :-)
- The computer industry revolution to make parallel computing widespread is happening right **now**.
- In the 1990's researchers in industry and academia came to a consensus on how to build "easily" programmable parallel machines.
- What do such machines cost?
- The price of a phone, laptop or a desktop.

More expensive machines

- Our `power.cs.lth.se` cost USD 3299 as new in 2005.
- Now they are a few thousand krona on ebay (ours is from there)
- Large scale servers cost much more of course.



Machines for multithreaded Java, C, C++

- These machines are called **cache coherent shared memory multiprocessors**, and are also often called multicores.
- Multicore actually means a machine with multiple processors, which do not necessarily have private cache memories.
- Obviously, just because we have multiple processors our program does not become faster automatically.
- We need to divide it into threads which can compute partial results.

Potential Sources of Troubles in Multicore Programming

- What can go wrong in this process?
  - **Amdahl's Law:** limited speedup if we cannot find enough parallel tasks the threads can work on.
  - Multiple threads modify the same data concurrently and corrupt the result, ie we have created **data races**.
  - Threads wait for each other in a circular way and we have a **deadlock**.
  - Our program becomes slower than we hoped for because the memory access penalty is much longer and **we failed to exploit the cache memories**.
  - One processor modifies data in its cache and another reads **obsolete data** from memory and the program crashes :-(
- We will next go through these in more detail.

Amdahl's Law

- Assume a sequential program has an execution time 1 time units, and a fraction P can be parallelized.
- What is the maximum speedup with N processors?
- Speedup $S = \frac{T_{slow}}{T_{fast}}$
- Speedup $S_N = \frac{1}{(1-P)+P/N}$
- Assume $P = 0.9$

N	S
2	1.8
3	2.5
4	3.1
10	5.3
100	9.2
∞	10

- Parallel programming is interesting only for sufficiently large P!

Data Races

```
count += 1;
```

- If two or more threads can modify the variable concurrently there is a data race and the final value written to memory is not predictable.
- It would have been predictable if += was implemented as an atomic instruction which modified a certain memory location but it's not.
- The Valgrind tool Helgrind can detect data races.
- In the new version of ISO C (aka C11) data races are undefined behavior (= very serious programmer bugs).
- To avoid data races we need to assure that only one thread can modify the data in a **critical region** which are typically created with some form of a lock. In Pthreads we can write:

```
pthread_mutex_lock(L);
count += 1;
pthread_mutex_unlock(L);
```


Locks

- A lock is a data structure (possibly as simple as only an integer variable) which only one thread at a time can hold, like a unique door key.
- There are at least two operations:
  - **Acquire** the lock. In Pthreads this function is called `pthread_mutex_lock` and it takes a pointer to a `pthread_mutex_t` as parameter. If some other thread already has taken the lock the second must wait.
  - **Release** the lock. In Pthreads this function is called `pthread_mutex_unlock` and it also takes a pointer to a `pthread_mutex_t` as parameter.
- In Pthreads it is also possible to see if the lock is currently free and only take it then while cancel the operation if the lock is taken in order to avoid waiting. It is called `pthread_mutex_trylock`.

Using Locks

- Thus, locks are used to protect data so that only one thread at a time can modify it in a critical region.
- Locks are thus used to achieve **mutual exclusion** which means that only one thread can do something with some data at a time.
- In Java, every object has such a lock, called a **mutex**, which is used with **synchronized** blocks or methods.

Let us try Helgrind!

- Let us run a program with a data race to see what Helgrind tells us!
- Without going into details the program increments a counter outside any critical region.
- We run it with 10 threads as follows:

```
$ gcc -g datarace.c -lpthread
$ valgrind --tool=helgrind a.out 10
```

```
==14599== Possible data race during read of size 4 at 0x10011130 by thread #3
==14599==    at 0x10000900: work (datarace.c:67)
==14599==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==14599==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==14599==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
==14599== This conflicts with a previous write of size 4 by thread #2
==14599==    at 0x10000910: work (datarace.c:67)
==14599==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==14599==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==14599==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
```

- Such messages can be invaluable.

Deadlocks

- A deadlock occurs when threads wait for each other in a circular way so that none can proceed.
- There are four requirements that all must hold for a deadlock to exist:
  1. **Mutual exclusion**, i.e. only one thread can use a resource R at a time.
  2. **No preemption**, i.e. it's not possible to take away the resource from a thread which currently holds it.
  3. **Hold and wait**, a thread which holds a resource R_1 may request another resource R_2 which may currently be held by another thread.
  4. There must be the **circular wait**, e.g. T_1 waiting for T_2 and T_2 waiting for T_1.

How can we prevent deadlocks in our programs?

- Mutual exclusion and no preemption may be difficult to avoid in general.
- In some cases we can, however, permit only one thread modifying some data, or multiple threads reading that data. This is called a **read-write lock**.
- To avoid hold-and-wait, we can use the `pthread_mutex_trylock` if it makes sense in our program.
- To avoid the circular wait, we can have rules which specify the order in which multiple locks may be acquired. If all threads follow these rules, there cannot be any circular wait.
- Helgrind does not detect deadlocks but it actually detects something better.
- What could be better than pointing out deadlocks?
- See next slide.

Non-Deterministic Execution

- One **very** important aspect of parallel programming is that execution normally is not deterministic.
- A deadlock might happen in one of 1,000,000 executions so testing as for sequential programs is not sufficient.
- By observing the order in which the threads acquire the different locks, one can check if the programmer had no rule for which order should be used.
- How can we observe that?

Helgrind and Deadlocks 1(2)

- So: Helgrind does **not** detect deadlocks but does something much more useful.
- Helgrind observes the lock-acquire order and complains when two threads acquire a lock in the opposite order.

```
void* work(void* p)
{
        arg_t*  arg = p;

        if (arg->i == 0) {
                pthread_mutex_lock(&A);
                pthread_mutex_lock(&B);

                printf("got both!\n");

                pthread_mutex_unlock(&B);
                pthread_mutex_unlock(&A);
        } else {
                pthread_mutex_lock(&B);
                pthread_mutex_lock(&A);

                printf("got both!\n");

                pthread_mutex_unlock(&A);
                pthread_mutex_unlock(&B);
        }

        return NULL;
}
```


Helgrind and Deadlocks 2(2)

- There is only a small probability that there will be a deadlock when running the program since the threads are started one at a time and the first most likely finishes before the second is even started.
- Helgrind reports the following, however:

```
==17471== Thread #3: lock order "0x100112AC before 0x100112C4" violated
==17471==    at 0xFF69288: pthread_mutex_lock (hg_intercepts.c:464)
==17471==    by 0x10000A13: work (deadlock.c:84)
==17471==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==17471==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==17471==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
==17471==  Required order was established by acquisition of lock at 0x100112AC
==17471==    at 0xFF69288: pthread_mutex_lock (hg_intercepts.c:464)
==17471==    by 0x100009C7: work (deadlock.c:71)
==17471==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==17471==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==17471==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
==17471==  followed by a later acquisition of lock at 0x100112C4
==17471==    at 0xFF69288: pthread_mutex_lock (hg_intercepts.c:464)
==17471==    by 0x100009D3: work (deadlock.c:74)
==17471==    by 0xFF6DB63: mythread_wrapper (hg_intercepts.c:201)
==17471==    by 0xFDB64F7: start_thread (in /lib/libpthread-2.11.2.so)
==17471==    by 0x60B3FDF: clone (in /lib/libc-2.11.2.so)
==17471==
```


Deadlocks in Java!?

- What should you do if you have made a very complex Lab 1 which has deadlocks?

C11 Atomic Types

- As we will see in Lecture 6, we can do as follows in C11:

```
_Atomic int     count;

/* Thread 1 */
count += 1;
                        /* Thread 2 */
                        count += 1;
```


Cache Memories

- Cache memories are extremely important on uniprocessors and even more important on multicores.
- If all threads would perform each memory access from system RAM, programs would simply be too slow. For example, the bus would be overloaded.
- **Unfortunately**, it's even more difficult on multicores than on sequential machines to exploit cache memories.
- Put simply, on sequential machines we get cache misses because the data does not fit (or was never accessed before) in the cache.
- On multicores, we also get cache misses when new values are communicated between different caches, called **true sharing misses**.
- In addition, we can also get cache misses due to some other processor wrote to a variable we are **not** interested in! These are called **false sharing misses**.

Caches in Multicores

- Suppose processor P_1 has read the value of a variable X.
- A copy of X will be in the cache of P_1.
- Assume next P_2 writes a new value to X.
- That new value, X', will be in the cache of P_2.
- What happens if P_1 wants to read X again???
- Without caches P_1 would probably see the new value (if it has reached memory).
- Unless we take special actions, depending on the time between the write by P_2 and the second read by P_1, it is more or less likely that P_1 will read an **obsolete** value of X from its cache!
- There are two special actions that the programmer needs to use:
  - P_2 must have released a lock L after writing X, and
  - P_1 must have acquired the lock L before reading X.
- The release by P_2 must happen before the acquire by P_1.
- We will go into details about this in Lecture 3 on **memory consistency models**.

A First Model of a Multicore Machine

- The simplest model of a multicore machine has a shared memory and no private caches.
- Without cache memories we do not have to be concerned about whether all threads see a consistent memory.
- Of course we must still avoid data races and deadlocks!
- Unfortunately, as we know, such a machine becomes too slow due the huge memory access times.
- However, the first multicore machine, from 1962, is similar to this, but the relative cost of accessing memory was less at that time.

A Second Model of a Multicore Machine

- A simple (and fastest for some applications) multicore machine is actually the Cell processor used e.g. in the Playstation 3
- The Cell processor has a normal 64-bit POWER processor (actually, it is multithreaded but ignore that for the moment) and a number of special compute nodes called **Synergistic Processing Units** or SPUs.
- Each SPU has a small private memory of 256 KB, called the **local store**.
- View the local store as a software managed cache of system RAM.
- The SPUs are programmed in C or C++ and on Linux the kernel schedules the SPU threads as it wishes... i.e. you can have any number of SPU threads regardless of how many SPUs your machine actually has.
- Normally the POWER processor controls the SPU threads and tells them what to do.

Computing on the Cell

- To process data on the SPUs we do the following:
  1. Copy input data from system RAM to an SPU's local store.
  2. Let the SPU compute a partial result.
  3. Copy the partial result to system RAM.
- Simple isn't it?

Yes, the Cell is simple and easy to understand

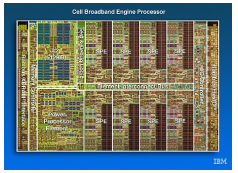
- The Cell is simple to understand for the programmer.
- The Cell is simple to make ***FAST*** in hardware.
- It's possible to write very fast software on the Cell due to its clever architecture (not because it's simple, of course)
- In addition the Cell uses relatively little energy — the most energy efficient supercomputers are all built with Cell processors.
- Here are some Gflops/s performance numbers:

Algorithm	Cell	Cray X1E	AMD Opteron	Intel Itanium2
dense float matrix multiply	204.7	29.5	7.8	3.0
sparse float matrix multiply	7.68	-	0.80	0.83
2-D FFT	40.5	8.27	0.34	0.15

There must be a catch...or?

- Being simple to understand is **not** the same as being simple to use.
- The copying of data to/from the local stores must be done using messages created by the programmer.
- You are all used to programming machines with cache memories, and the local store of an SPU can be seen as a cache memory, except that you must manually transfer the data!
- So why are there no hardware managed cache memories for the SPU's then?
- The researchers at IBM concluded that a sufficiently large number of applications can execute faster if we use the transistors for computing and not for cache memories — i.e. instead of only **one huge cache** we can have a number of small SPUs with their local stores.
- By letting the programmer manage the data transfers using a clever feature, very good performance can be achieved on the Cell.

A Cell Chip



- An SPE contains an SPU and a local store of 256 KB.
- On the Sony PS3 you can only use six of eight the SPU's.

A Third Model of a Multicore Machine

- The third model of a multicore machine consists of a number of nodes which can send control and data messages to each other through a general interconnection network.
- Each node has a processor, e.g. two levels of caches and a portion of the RAM memory.
- Each node implements in hardware a protocol for transferring data between caches and memory.
- The purpose of this protocol is to fetch data from a remote node at a read cache miss, and when writing to tell the other nodes which have a copy of that data that their copies have become obsolete.
- The protocol is called a **cache coherence protocol**.
- It is completely implemented in hardware.

Memory Ordering

- Consider a parallel program with two threads, T_1 and T_2, and two variables a and f, where f is a flag.
- Is the following code correct (in some sense) ? Both variables are initially zero.

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        f = 1;
}
```

```
// called by T2
void w(void)
{
        while (!f)
                ;
        printf("a = %d\n", a);
}
```

- Can T_2 print out zero? (unfortunately, yes)
- T_2 might in fact not print anything!

Why T_2 May Print Zero

- The compiler can legally reorder the writes to a and f.
- The compiler can allocate both a and f to registers (across multiple functions — not common but HP have done so for 20 years) and may never have to write back f to memory if v is called in an infinite loop...
- Even if both of T_1's writes are performed in source code order, a and f may be located in RAM in different nodes and it may take longer time for the write to a to reach its node.
- What is needed are strict rules about what may or may not happen. We will look at such in Lecture 3 but until then, the following rule should be used:

Don't use normal variables for synchronization — instead use locks!

More about Unlock/Lock

- Essentially, an unlock by T_1 will have the side effect of forcing the processor to wait until all previous writes have been noticed by all other processors.
- A lock has the side effect of performing all pending incoming invalidations so the old value of a is removed.
- Then the code will work. Unlock and lock execute special instructions for this — see Lecture 5.

Parallel Programming in Java

- Creating Java threads
- Java synchronization
- The Java memory model and `volatile` attributes.

Creating Java Threads

- Either extend the `Thread` class or implement the `Runnable` interface.
- Your thread needs a `public void run()` method.
- Don't call `run()` — you should instead call `start()`.
- To wait for a thread to terminate, you use `join` in a try-catch block.

Java Synchronization

- A program which uses synchronization properly to avoid data races is said to be **thread safe**.
- Every Java object has a lock. Before entering a method declared `synchronized`, the JVM checks if the calling thread is the owner of the lock. If the lock is owned by another thread, the calling thread is blocked and is put into an **entry set** for the object.
- If no thread owns the lock the calling thread becomes the owner.
- Using `synchronized` we can avoid data races but we may end up with a deadlock, as illustrated on the next page.

Deadlock with Synchronized Methods

```
public synchronized void insert(Object item)
{
        while (count == BUFFER_SIZE)
                Thread.yield();
        ++count;
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
}
```

- The `remove` and `insert` methods use the same lock to avoid data races.
- Suppose the producer calls `insert`, finds the buffer full, and calls `yield()` which lets another thread run.
- When the consumer calls `remove` it is blocked since the producer still owns the lock and a deadlock has occured.

The Java Object Wait-Set

- There are two methods `wait()` and `notify()` which are used to avoid the previous deadlock problem.
- In addition to the `entry set` there is also a `wait set`.
- Calling `wait()` releases the lock, blocks the thread, and puts it into the `wait set`.
- Calling `notify()` wakes up one thread in the `wait set` if there was any there, puts it it into the `entry set` and sets its thread state to Runnable.
- Calling `notify()` does **not** release the lock.
- One can also call `notifyAll()` which wakes up all threads in the `wait set`.

More Details

- The Java object lock is a so called **recursive lock** which means that we can call other synchronised methods for the same object without being blocked.
- Of course, a thread may own multiple object locks (ie, call synchronised methods for different objects).
- A notification for an object with an empty `wait set` has no effect (ie, the object does not remember the number of notify calls, as it does remember the number of V calls for a semaphore).

Using `wait()`

- A thread waiting in the `wait set` can be interrupted, ie another thread can call its `interrupt()` method. This results in the `InterruptedException` being thrown:

```
try {
        wait();
}
catch (InterruptedException ie) { /* ignore */ }
```


The Java Memory Model

- The Java Memory Model has been changed after William Pugh identified problems with it several years ago.
- A relatively new keyword is `volatile`. It's meaning has changed since it was introduced.
- Initially, the accessing of a particular volatile attribute of an object was serialized, i.e. all threads saw these accesses in one total order.
- These accesses were, however, unrelated to accesses of other variables.
- If you updated some variables and then set a volatile flag to indicate you were done, your program was buggy since the accesses were not ordered, i.e. the updates may be seen after the new value of the flag!

New Semantics of Volatile in Java

- In more recent versions of Java an access to a volatile attribute is ordered with respect to other accesses in the same way as synchronized blocks:
  - When entering a synchronized block, or reading a volatile attribute, the cache is conceptually made invalid.
  - When leaving a synchronized block or writing a volatile attribute the cache is, again conceptually, copied to memory.
- With this semantics using a `volatile int flag` as in the previous C code works as expected.

Introducing Pthreads

- Pthreads stands for POSIX Threads and is available on all UNIX machines, including Linux and MacOS X.
- POSIX stands for Portable Operating System Interface and is an API for UNIX programmers.
- Pthreads are quite similar to Java threads so nothing dramatic will happen to you...
- In the next slides we will show you how to start a thread and synchronize threads.

Getting started

- `#include <pthread.h>`
- `pthread_t` is the type of a thread.
- Create threads using `pthread_create()`.
- Wait for a thread using `pthread_join()`.
- Terminate a thread using `pthread_exit()`.

`pthread_create()` 1(2)

```
int pthread_create(
        pthread_t*              thread,         // output.
        const pthread_attr_t*   attr,           // input.
        void*                   (*work)(void*), // input.
        void*                   arg);           // input.

struct { int a, b, c } arg = { 1, 2, 3 };
status = pthread_create(&thread, NULL, work, &arg);
```

- The thread identifier is filled in by the call and attributes are optional.
- The created thread runs the `work` function and then terminates.
- A class is called a `struct` in C — no methods and everything public.
- Typically multiple arguments are passed in a struct as above.

`pthread_create()` 2(2)

- A zero return value from `pthread_create()` indicates success, while a nonzero describes an error printable with `perror`.
- The `work` function can return a void pointer.
- Calling `pthread_join` waits for the termination of another thread and also gives access to the returned void pointer.
- A thread can only be joined once.

`pthread_join()`

```
int pthread_join(
        pthread_t       thread,         // input.
        void**          result);        // output.
```

- The call causes the caller to wait for the termination of a thread.
- If non-NULL, the terminated thread's return value is stored in result.
- A thread can only be joined by one thread.

`pthread_exit()`

```
void pthread_exit(void*);       // return value from work.
```

- Either use this or a return from the work function to terminate a thread.
- At termination of the main thread using `exit` or return, all other threads are killed, so use `pthread_exit` by the main thread instead.
- After a thread has terminated, the Pthreads system waits until some other thread joins with it. Then the terminated thread's resources are recycled.
- If a thread will never be joined, it should have been detached so that the system can recycle resources at termination.

`pthread_detach()`

```
void pthread_detach(pthread_t thread);  // recycle at exit.
```

- A thread can be detached from the beginning by specifying an attribute saying so at `pthread_create`.
- Or, any thread can call `pthread_detach`.
- A detached thread cannot be joined.

sum.c and Dataflow.java

- Next we will look at two programs:
  - `sum.c` demonstrates a Pthread program.
  - You will parallelize `Dataflow.java` in Lab 1.
- You will later translate your `Dataflow.java` to Scala and then to C.

Parallelization of an Application

- Quite difficult usually!
- The programmer must identify parallel parts of a program, eg:
  - One outer loop iteration, or
  - one function call.
- The parallel parts which read or write the same data must communicate and be synchronised:
  - Either using message passing to communicate and wait for each other, or
  - communicate data through shared memory and synchronise with eg locks.
- Communication and synchronisation should be **minimised** to improve performance.

The Three Main Issues

- Ignore for the moment that memories are slow and caches are useful.
- Now the two main issues are:
  - Program correctness, and
  - Load-imbalance: some processors might have less work to do and must wait for others before proceeding.
- Unfortunately, memories can be hundreds of times slower than microprocessors:
  - Problem 1: communication usually creates new cache misses, and
  - Problem 2: caches in multiprocessors can introduce obscure bugs.
- All multiprocessors have some form of caches: tuning usually means exploiting them better and this is the third main issue.

Programming on a Multiprocessor

- Global variables and data allocated with malloc/calloc are shared
- Data allocated on the stack is private to a thread (but nothing prevents you from giving away a pointer to stack variables but doing so is uncommon).
- The most convenient way to parallelize a program is to do it ***incrementally***, one loop at a time.
- There is a standard for this which is an extension to C/C++/Fortran called OpenMP.
- The extensions are specified by the programmer as comments in Fortran and as pragma (preprocessor directives) for C/C++.
- GCC 4.4 supports OpenMP 3.0 from 2008.

An Example: Simulating Ocean Currents 1(2)

- For climate modelling of the earth, we can study interactions between oceans and the athmosphere.
- Our example program simulates the motion of water currents in an ocean.
- Predicting the state of the ocean at any time requires simulating:
  - athmospheric effects,
  - wind,
  - friction with the ocean floor and walls.
- An ocean is represented as a set of cross-sections, like chess boards put on each other.

An Example: Simulating Ocean Currents 2(2)

- The Atlantic is about 2,000 km $\times$ 2,000 km.
- Using 100 $\times$100 points in a cross-section gives 20 km between each point.
- We actually want to simulate at a much finer granularity.
- Time is represented by steps at which all variables are re-calculated.
- Simulating eg five years with 8 hours steps requires 5,500 steps.
- The computations are enourmous but can be parallelised.

Terminology

- A **task** is a unit of work that should be performed, and which can be performed in parallel with other tasks.
- A **thread** is a software entity which runs on a processor and works on tasks, (one at a time)
- A **processor** is the real hardware which executes one thread.

Parallelization of a Sequential Application

Four main steps in performance programming for a multiprocessor

1. Decomposition — dividing the work into parallel tasks.
2. Assignment — deciding which thread should do which tasks.
3. Orchestration — communication and synchronisation among the threads.
4. Mapping — deciding which threads should run on which processors.

- Decomposition and assignment are called **partitioning**.

Decomposition

- The number of tasks available at any time limits the amount of concurrency.
- We might want our program to have as many tasks as possible: for Ocean we can have
  - one task per set of cross-sections,
  - one task per cross-section,
  - one task for a subset of a cross-section, or
  - one task per grid point.
- What is best cannot be stated without knowing more details!
- ***We want our program to have as many tasks as can be efficiently handled.***
- Unfortunately, one partitioning may be best for one computer but not for others!
- ***Performance portability*** is complicated by varying cache sizes, number of CPU's and how the CPU's and memories are interconnected.

An example related to Ocean 1(3)

Assume a program has two phases.

- In phase 1 $n \times n$ operations are performed on grid points in parallel.
- In phase 2 the sum of values of the $n \times n$ points are accumulated.
- With p processors phase 1 can be done in time n^2/p.
- Without considering time for communication and synchronisation, phase 2 can be done such that each processor adds n^2/p values to the global sum.
- What will the speedup be if we consider synchronisation?

An example related to Ocean 2(3)

- Phase 1 can be done in parallel, taking time n^2/p.
- Since a single variable is incremented n^2 times in phase 2, it must be incremented in a critical region to avoid race conditions. This results in time n^2.
- The sequential execution time is $2n^2$.
- The parallelised execution time is $n^2/p + n^2$.
- The speedup will be $\frac{2n^2}{n^2/p+n^2} = \frac{2p}{1+p} < 2$.
- So, we need to find a better decomposition. See next slide.

An example related to Ocean 3(3)

- Let each processor sum up its n^2/p points to a private variable in phase 2.
- Add a phase 3 which adds all theses private variables, taking p time.
- New speedup becomes linear in the number of processors, p.

Assignment of tasks to threads

- Balance computation, I/O, data access, and communication.
- Easiest if it can be done well statically (at compile-time or just when the program starts and has read some specific input parameters), think eg of outer loops in matrix multiplication with a fixed amount of work for each processor.
- For other programs, eg a parallel chip simulator, it can be difficult to divide the work. A too sophisticated scheme for load balancing can also lead to too much run-time overhead.
- Run-time assignment is called dynamic assignment.
- Partitioning (decomposition and assignment) is an algorithmic step in the parallelisation and is mostly independent of the details of the machine.

Orchestration

- It is now we write source code.
- The programming model (message passing vs shared memory) determines how to do this.
- Questions in orchestration include:
  - How to organise data structures to reduce cache misses?
  - How to reduce the cost of communication and synchronisation?
  - How to reduce serialisation of access to shared data?
  - How to schedule tasks to satisfy dependences early?

Mapping

- Mapping is done in cooperation with the operating system at run-time.
- Mapping specifies which thread should run on which processor.
- Sometimes the programmer can specify on which processor a thread should run, called to **pin** it.
- Pinning can be important for large multiprocessor with a non-trivial topology (something other than a bus).
- It may very well be useful to have more threads than processors.

Owner computes

- So far we have parallelised a program with respect to computation.
- Sometimes it is better to parallelise it with respect to data (or both).
- The basic rule then is **owner computes** (and modifies) and other threads read the data.
- An extension to Fortran, called High Performance Fortran allows the programmer to distribute data structures to threads.
- In Ocean, it is natural to use the owner computes rule.

Multicore Architectures

Contents of Lecture 2

- SIMD architectures
- Vector architectures
- Distributed memory architectures
- Shared memory architectures
- Multithreaded architectures
- Dataflow architectures
- Systolic arrays
- Multithreaded architectures

Why is it important to study computer history?

Why Is Important to Study Computer History?

- You will notice that most of the fancy designs have been tried already
- You will understand why some designs survive and others die
- You will not be impressed when somebody presents the new breakthrough which will change everything and make parallel computing easy
- You will understand where today's machines come from which is important to help you to predict which architectures will be around in 10 or 20 years... since you might need to know that

Technology in Computer Generations

1. -1954, vacuum tubes and relay memories
2. 1955-1964, discrete transistors and core memories
3. 1965-1974, integrated cicruits, pipelining, cache memories
4. 1975-1990, DRAM memory, vector and multiprocessors
5. 1991-present scalable architectures

Zuse Z3 1(3)

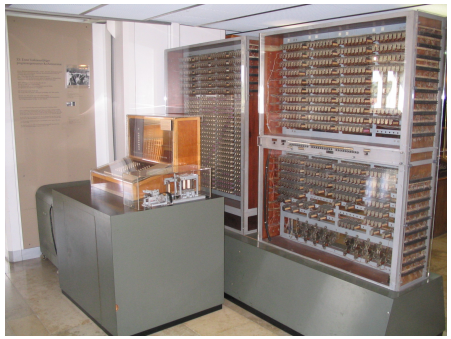
- Konrad Zuse was born in 1910 and became a civil engineer
- He found it boring and started at Henschel to design aircrafts
- He also found the many calculations made by hand boring and started to dream about a machine that could do them for him
- He wanted to automate his calculations and filed a patent for the computer in 1936, and then started to build several machines...Z3:

number representation	binary floating point with NaN and ∞
Turing complete	yes, shown 1998
clock frequency	5.3 Hz
average add delay	0.8 s
average multiplication delay	3 s
memory size	64 words of 22 bits
power consumption	4 kW
weight	1,000 kg
in operation	1941 - 1943 (when it was bombed)

Zuse Z3 2(3)

- After the war, in 1946, Zuse sold patents to IBM to fund his computer company Zuse KG which built a replica of Z3 now at Deutsche Museum in München, where also original machines by Pascal and Leibniz and others can be seen. Very inspiring museum of engineering dreams, successes and failures.
- According to Zuse, IBM didn't understand the coming computer revolution then but wanted his patents for other things.
- His company produced approximately 250 computers (which was a good number as we will see below)
- Siemens bought Zuse KG in 1968.
- When the Computer Science and Engineering program (i.e. D-Linjen), started in 1982, celebrated its 101 anniversay in 1987 Konrad Zuse accepted an invitation by Professor Lars Philipson to give a speech here but unfortunately he could not come. He died 1995.
- So, the inventor of the computer wanted to talk to the engineering students at LTH!

Zuse Z3 3(3)



The Princeton IAS Machine

- The Princeton IAS machine was built from 1942 to 1951 and became fully operational in 1952.
- The chief designer was John von Neumann.
- It used two's complement to represent negative numbers.
- Both instructions and data were stored in memory, so loops could be implemented by modifying the conditional branch instruction...
- An addition took 62 μs
- Many machines were built as derivates from this.
- The founder of the computer science department, Carl Erik Fröberg, was sent by Vetenskapsakademien 1947-1948 to the U.S. to study the development of electronic computers, and built the SMIL computer in Lund (Sweden's second after BESK).

Commerical Computers

- The first commercial computer was in a sense multicore machine: the BINAC.
- It had two bit serial CPUs each with a 512 word memory, and was built by the Eckert-Mauchly Computer Corporation in 1949.
- It's not regarded as a multicore machine, however. The first multicore was delivered by Burroughs to the US defense only 13 years later, in 1962.
- Compared with the Z3, it was **very much** faster.
- It could compute for over 31 hours without any error at EMCC.
- Then they disassembled and packaged it for delivery to the Northrop Aircraft Company.
- The customer was so concerned about security that no employee from EMCC was allowed to come and assemble it...

BINAC

- Unfortunately it never worked properly after delivery to the Northrop Aircraft Company.
- EMCC and Northrop blamed each other...
- If you buy the world's first commercial computer it might have been a good idea to let the seller assemble it!!!
- EMCC is now called Unisys and has about 20,000 employees.
- A few months later a mathematician at ETH in Zürich visited Zuse and asked him to program his new Z4 to solve a differential equation which he did on the spot and then ETH bought it. The Z4 was the only commercial computer in Europe during 1950 and 1951.
- Zuse invented the programming language Plankalkül for it.

IBM

- IBM machines were called electronic calculators and were programmed by manually plugging wires.
- Cuthbert Hurd instead wanted to store the program in memory (as in the von Neumann architecture) and convinced the new IBM president to make a commercial machine that also could be programmed using punch cards for that purpose.
- He hired the first team of programmers, including John Backus and Fred Brooks, as well as and John von Neumann as a consultant.
- This started the amazing story of IBM computers.

IBM 701

- The IBM 701 was IBM's first commercial scientific computer.
- Introduced on April 29 1952.
- Memory consisted of 2048 36-bit words.
- Two registers accessible to programmers.
- 19 systems were installed.

IBM 704

- The IBM 704 was introduced 1955.
- Chief architect was Gene Amdahl (who built a computer as PhD thesis in 1952).
- It was the world's first mass produced computer (i.e. more than 100 machines) with floating point hardware.
- It had 5 programmer accessible registers.
- 123 systems were sold until 1960.
- Both LISP and FORTRAN were created on the IBM 704.
- 40,000 instructions per second could be executed.
- Floating point performance was 5 kFLOPS.
- Software was incompatible with IBM 701.

Amdahl

IBM 709 and IBM 7090

- IBM researchers John Cocke (one of the persons to whom the course book is dedicated) and Daniel Slotnick writes a memo discussing **parallel computing**. Slotnick proposed SIMD processing.
- The IBM 709 was introduced 1958 as an improvement over IBM 704.
- Again, software was incompatible with the previous machine.
- An emulator was provided which could run IBM 704 software.
- On the IBM 704, I/O was done from the CPU but the IBM 709 improved I/O by introducing separate processors called I/O channels for this.
- The IBM 7090 was introduced 1959 and was similar in design as the IBM 709 but was implemented with transistors instead of vacuum tubes.

IBM 360 — the First Computer Architecture

- One of the main annoyances with previous machines was that every new machine required new software.
- The IBM 360 was instead an **architecture**, introduced in 1964.
- IBM wanted to sell "inexpensive" slow machines and scale the performance and price but they should be **compatible** for software.
- The price was initially set based on the performance.
- Later machines include the IBM 370 series, and the IBM 3090.
- A recent compatible machine was announced 2010 and has quad core processors clocked at 5.2 GHz and can have up to 3 TB of **RAM** memory.

More About the IBM 360

- Introduced the 8-bit byte.
- It had 32-bit words.
- It was byte addressable.
- 16 general purpose registers.
- 24-bit addresses.
- First TLB was in IBM 360/Model 67 (and GE 645)
- First data cache was in IBM 360/Model 85
- Tomasulo's algorithm for out-of-order execution was first used in IBM 360/Model 91

IBM Stretch

- IBM 7030, or IBM Stretch, was the fastest machine from 1961-1964.
- The design goal was to be 100 times faster than the previous machine!
- It was neither technically nor commercially successful but important technologies were developed for it, including:
  - Instruction pipelining: also available to some extent in both of Z1 and Z3, and Illiac IV.
  - The pipeline stage names **fetch, decode, execute** come from Stretch
  - Memory interleaving

Control Data Corporation

- Control Data Corporation, founded 1958, was a small competitor with IBM during early 1960's.
- One of their engineers, Seymour Cray, set in 1960 out to build the CDC 6600, a machine which should be 50 times faster than their previous, CDC 1604 (a machine similar to the IBM 7090).
- After four years of development, the management was quite worried about what was happening...
- Cray's response was that he wanted his own lab in his home town, near Minneapolis, and nobody was allowed to go there except by invitation — otherwise he would quit.
- The CDC's president accepted his demand and Cray with some engineers moved to the new lab.
- He chose the location so that it was too distant for a one-day visit by car.

Supercomputing

- In 1964 the CDC 6600 was released and Control Data Corporation then dominated the supercomputer industry during the 1960's.
- The CDC 6600 became the fastest computer in the world and started the supercomputing era.
- The CDC 6600 is regarded as the first superscalar computer — execution was controlled by a scoreboard.
- Now the IBM president asked approximately ***How is it that this tiny company of 34 people including the janitor can be beating us when we have thousands of people?***
- Cray's reply: ***You just answered your own question.***

The IBM Response

- IBM's response was to also set up a remote lab of 200 engineers, to build an even faster machine.
- It was intended to be able to issue seven instructions per clock cycle.
- It turned out to be impossible to achieve the desired performance levels if binary compatibility with the 360 architecture was to be maintained.
- The project was cancelled in 1969.
- It is extremely important to have a good instruction set architecture, because all future CPUs and optimizing compilers will have to live with it.
- This cannot be overstated. Compare X86 and Itanium, and AMD's X86_64

Vector Supercomputers

- Daniel Slotnick (who wrote the memo at IBM about parallel computing with John Cocke) had moved to the company Westinghouse where he designed a parallel machine called Solomon.
- Solomon was intended to have one CPU called the control unit and 256 processing units with their own memories
- The control unit specified the instruction that all processing units should perform, i.e. it's a SIMD vector machine.
- The Solomon machine was built with funding from the US Air Force but they withdrew from the project in 1964, and it was cancelled.

ILLIAC IV

- When Solomon was cancelled, Slotnick convinced Burroughs and the University of Illinois to build a similar machine, the ILLIAC IV.
- It was built from 1965 - 1975.
- It was intended to reach 1 GFLOPS using 256 SIMD units clocked at 13 MHz, but a smaller machine could only be built which reached 200 MFLOPS.
- As Solomon, it relied on the application to be suitable for SIMD processing. The ILLIAC IV was the fastest machine from 1975 until 1981 for ***suitable*** applications.
- A lot of research on optimizing compilers was done at the University of Illinois.

Additional Machines at CDC and Cray's Departure

- The next machine from Control Data Corporation, the CDC 7600, was about five times faster than the CDC 6600
- These machines were very expensive but fastest in the world and good investments for some customers
- Development costs of each of 6600 and 7600 almost bankrupted CDC.
- The next CDC machine, CDC 8600, was too complex and Cray told the president they had to redesign it from scratch... :-(
- In the meantime CDC was working on another, less complex machine.
- The president, Norris, didn't want to prioritize Cray's redesign.
- He left to fund Cray Research in 1972.
- Norris invested USD 300,000 in Cray's startup.
- The new CDC machine was not a success due to it relied too much on fast parallel parts but was slow at sequential parts — Amdahl's Law!!!

Cray Research

- First a word from a CTO of HP, Joel Birnbaum: ***"It seems impossible to exaggerate the effect he had on the industry; many of the things that high performance computers now do routinely were at the farthest edge of credibility when Seymour envisioned them."***
- If it was too expensive to develop the next machine at CDC, what could he do with his startup???
- It turned out that Seymour Cray was well known and respected at Wall Street and they got all funding they needed.

The Cray-1

- In 1975 the Cray-1 was announced with a clock rate of 80 MHz — it was very fast both for scalar and vector parts of an application.
- Vector machines before (and sometimes after) the Cray-1 were keeping the vectors in memory, this allows flexible vector sizes.
- Cray instead used vector registers of fixed sizes.
- Due to the fact that vectors often were used several times this increased performance.
- After the announcement of their machine, there was an auction for who would get the first Cray-1.
- Seymour Cray was not impressed by the ideas behind ILLIAC IV: he said ***"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"***
- Many years later, high performance microprocessors changed that, of course.

Cray-1

The First Real Multicore was Released in 1962

- Stepping back to 1962, Burroughs released the Burroughs D825.
- This is the first multicore machine. It was called a multiprocessor.
- It had up to four CPUs and 16 memory modules.
- It was symmetric in that all CPUs could access any memory module.
- It's operating system had a shared ready queue of tasks.

Multics Operating System

- AT&T, General Electric and MIT set out to build the Multics operating system in 1965.
- It was intended to run on multiprocessors.
- In 1969 it ran (to some extent) on a 8 CPU multiprocessor built by Honeywell.
- It was a failure but out of it came UNIX and later Mac OS X and Linux.
- The same year Dijkstra formulated the critical regions problem.

Summary so far

- What have we got so far?
- Software is FORTRAN
- Optimizing compilers which can parallelize and vectorize numerical codes are being developed, they are very good at vectorization but less at parallelization (and still are).
- By the end of 1970's and beginning 1980's supercomputers with few CPU plus vector instructions are the fastest machines, either from Japanese companies or Cray.
- A typical multiprocessor was bus-based.

CISC Processors

- Before the 1960's hardware was more expensive than software.
- There was a software crisis in the 1960's when it was realized that software was becoming the expensive part.
- Most of the computer industry went in the direction of creating CPUs which they thought would help programming in high-level languages.
- For instance instructions to set up a stack frame and save registers and later restore registers were common, e.g. in the VAX.
- Instructions to copy memory were taken for granted. With virtual memory and page faults such instructions are tough to implement efficiently.
- The VAX even had an instruction to evaluate a polynomial.
- Fancy instructions were "selling features"

IBM 801

- After finishing his PhD in mathematics, John Cocke worked for IBM from 1956 to 1992.
- In 1979 John Cocke designed a new processor which went completely against the mainstream of more complex instruction sets.
- As we all know this was the foundation for the RISC revolution — all your mobile phones use it and some of the fastest computers.
- The commercialization of superscalar RISC was in the POWER architecture.
- POWER and PowerPC are identical today.
- Now Google, Samsung, IBM, Canonical (Ubuntu) and others are collaborating on POWER — see `openpowerfoundation.org`.
- The labs and project will be performed on the 4 CPU POWER, which thus is the brain child of one of the best computer architects ever.

Many New Parallel Computing Principles

- With increased transistor budgets many new ideas could be realised.
- Today we know which one won the battle: shared memory multiprocessors with coherent caches, i.e. essentially the Burroughs D825 but with added cache memories.
- It's relatively easy to invent new faster parallel machines, if you can ignore the constraint of having paying commercial customers to make your company survive (by instead letting the US Department of Defence pay).
- Many niche market machines were developed by companies which spent huge amounts of money before cancelling their projects.
- Typical killing features for **hardware companies** include
  - insufficient performance
  - requiring new programming languages
  - requiring nonstandard extensions to C or FORTRAN
  - taking too long to reach market so your technology becomes obsolete
- New programming languages are of course not bad but need time to have impact which companies often cannot afford.

Examples of Parallel Architectures

- MPP (massively parallel machines)
- Dataflow machines are developed at MIT by Jack Dennis and Arvind
- Systolic arrays are described by H.T. Kung and Charles Leiserson
- The C.mmp multiprocessor consisting of 16 PDP-11 connected by a crossbar to a shared memory was built at CMU.
- The multithreaded machine Denelcor HEP developed by Burton Smith
- Stanford DASH
- KSR-1 and DDM

MPP Machines

- By Massively Parallel Machines are meant distributed memory machines where each node has a CPU and a private memory.
- The number of such nodes are typically hundreds or thousands.
- To communicate, programmers (or optimizing compilers) must create messages to send to some other node.
- MPI (message passing interface) is an industry standard for high performance computing and is actively used today as well.

Thinking Machines, Inc

- It was founded 1983 and they were the most "ultra cool" machines in the 1980's.
- Famous persons like Guy Steele worked for them (who coauthered the Java Language Specification among **many** other things).
- The first **connection machine**, the CM-1, consisted of up to 65,536 one-bit processors, each with 4 KB RAM.
- To improve performance, the CM-2, used normal floating point processors (Weitek), and the last model, the CM-5, used normal SPARC processors from Sun.
- Sun later bought the company.
- They were programmed in FORTRAN, as well as a parallel version of Lisp, called *Lisp (star Lisp).

The CM-5

Dataflow Machines

- Dataflow machines have a completely different hardware architecture than anything you are used to.
- Instead of a program counter which addresses the next instruction to execute, a dataflow program is a dataflow graph.
- The dataflow graph is worked on in parallel. An operation is called a token and as soon as an operation's operands have been computed that operation can be computed by a processor, and the result then stored back to a memory in a special way (see below).
- The memory is called a **token store**.
- In a normal computer the data is stored in memory and an instruction accesses that memory location.
- In a dataflow machine, the data is stored in the instruction!

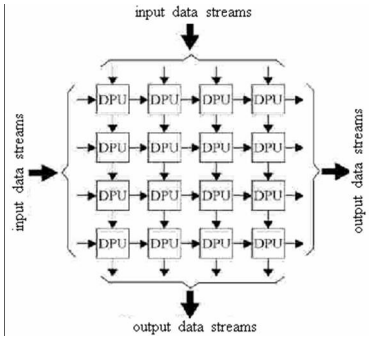
More about Dataflow Architectures

- When an operation has been computed, and the result should be stored in memory, the result is stored in each instruction which needs the data as an operand.
- Instead of shared memory and a program counter, a token matching mechanism is needed to propagate data to the instructions (tokens).
- The main architecture and machine designer at MIT has been Arvind, who built the Monsoon Dataflow machine together with Motorola, but it was never a commercial machine.

Systolic Arrays

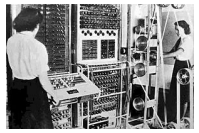
- The name systolic arrays was coined by H.T. Kung and Charles Leiserson (one of the authors of Introduction to Algorithms) who wrote the first paper on this architecture.
- By systolic is meant computing at a certain pace, like heart beats.
- The idea is to put processors in an array and let a processor get data from its left neighbour and produce data to its right neighbour.
- Alternatively input comes from two sources and are written to two destinations.
- The key idea then is to have very efficient communication since the data is just copied from one processor to the next.
- The potential performance is huge.
- The main problem is to map algorithms to this architecture.

Systolic Array Idea



A Somewhat Earlier Systolic Array from 1944

- The British Collosus code breaking machines also used the idea of systolic arrays. 10 such machines were used until the end of WWII.



The iWarp Machine

- CMU and Intel set out to explore the possibilites of building a parallel supercomputer based on the CMU Warp systolic array machine.
- An iWarp machine had 64 compute cells each of which was a 32-bit processor.
- They were produced around 1990 and in 1992 Intel created a supercomputing division (now no longer existing).
- One result from the systolic array project at CMU is the software pipelining algorithm called modulo scheduling which is now widely used in optimizing compilers also for normal CPUs.

Multithreaded Architectures

- To avoid the problem of long latency operations, instead of waiting an alternative is to switch thread to execute.
- Obviously this must be done in hardware.
- There are many different ways to design such multithreaded machines.
- Some switch threads every clock cycle, others when there is a long latency operation such as a cache miss.
- After designing the HEP machine for Danelcor, Burton Smith designed the Tera machine in his own company.
- The Tera parallel machine had no caches but some interesting ideas, e.g. with each memory word was a full or empty bit that could be used for synchronization. Writing a word set the bit and reading a word cleared it.
- They had very few commercial customers.

Cache Coherent Shared Memory Multiprocessors

- From the few samples we have seen, there are many ways to build parallel computers.
- In the 1980's multiprocessors with cache memories were becoming commercial products.
- The main question was how to make them scalable to a large number of CPUs.
- A single bus is of course out of the question.
- A mesh is one acceptable topology and was used in the Stanford DASH machine.
- We will look at the details in Lecture 3, but consider a multiprocessor with caches where a memory word may be copied to any number of caches.
- Sometimes we need to tell every node with a copy to forget about its copy — and fetch a new value from memory next time.
- How should this information be organized?

A Directory

- A so called directory at a memory node keeps track of which nodes have a copy.
- Note that a directory can become large if there are many nodes and we allow any number of copies.
- An alternative is to have a fixed maximum number of copies stored in the directory and move the directory information for a certain memory block to the operating system kernel.
- So each memory address has a **home location** i.e. a node in whose memory it is located.
- The home location, or node, is typically static.

COMA

- Cache-Only Memory Architectures, COMA.
- Consider two threads in different nodes accessing the same data in a third node.
- If both threads read and write the data, the home node of the data will be involved.
- The latency is increased by having to go to that home node for telling the other thread (i.e. cache) to write it's modified copy back to memory.

Two Machines: KSR-1 and DDM

- Researchers at SICS (Swedish Institute of Computer Science) as well as a company called Kendall Square Research (close to MIT) invented a different design.
- Instead of having a static home location, data can "diffuse" to a suitable node that is currently using it and let that be the new home. That way only two nodes need to be involved in the above scenario.
- The Swedish machine was called the Data Diffusion Machine.
- For some applications this is useful while for others it is not very important.

Summary

- It is now with the difficulty of increasing the clock rate that chip companies want us to use multicores everywhere
- The most important lessons from the past include:
  - Programmability and mass market are essential
  - Amdahl's Law is extremely important
  - Cache coherent shared memory multiprocessors are here now and we must write general purpose applications in Java and C/C++ for them.
  - It is essential to understand that all the fancy ideas have been around many decades and we should be sceptical when somebody tells us they have the ultimate solution for faster parallel machines...
  - GPU's are SIMD with multiple threads and work very well for some application types
- Next time we will look at shared-memory multiprocessors in detail.

Memory Consistency Models

Contents of Lecture 3

- The need for memory consistency models
- The uniprocessor model
- Sequential consistency
- Relaxed memory models
- Weak ordering
- Release consistency

The Need for Memory Consistency Models

- In this lecture we assume all threads are executing on different CPUs. This makes the presentation simpler so that we can use the words "processor" and "thread" interchangeably.
- A memory consistency model is a set of rules which specify when a written value by one thread can be read by another thread.
- Without these rules it's not possible to write a correct parallel program. We must reason about the consistency model to write correct programs.
- The memory consistency model also affects which programmer/compiler and hardware optimizations are legal.

Who Must Know About the Consistency Model?

- The memory consistency model affects
  - System implementation: hardware, OS, languages, compilers
  - Programming correctness
  - Performance
- There are several different consistency models.
- One lets us use normal variables for synchronization, while most don't.
- Instead we must use some form of lock/unlock calls to ensure that our threads will see the data most recently written by another thread.

Uniprocessor Memory Consistency

- To the programmer, the last written value to a memory location will always be the value read from that same location.
- This is the contract to the programmer.
- Most programming languages, including C99, are specified using this model for single threaded programs. In Lecture 6 we will look at the new C11 standard which specifies it's memory model with support for multiple threads and atomic variables.
- Why is the following program invalid?

```
a = 1;
a = ++a + a--;
printf("a = %d\n", a);
```

- In C/C++ there is a rule which says a variable may only be modified once between two sequence points (e.g. semicolons) so for C/C++ the uniprocessor model has additional constraints.

Effects of the Uniprocessor Model

- We can as programmers reason about which values are legal (provided we know about the additional rules in C/C++).
- Performance optimizations are possible both for optimizing compilers and processor designers.
- Compilers can put a variable in a register as long as it wishes and can produce code which behaves **as if** the variable was accessed from memory.
- Compilers can thus swap the order of two memory accesses if it can determine that it will not affect program correctness under the uniprocessor model.
- A superscalar processor can execute two memory access instructions in any order as long as they don't modify the same data.
- A processor may reorder memory accesses e.g. due to lockup-free caches, multiple issue, or write buffer bypassing.

Lockup-Free Caches

- Suppose a cache miss occurs.
- In a lockup-free cache other subsequent cache accesses can be serviced while a previous cache miss is waiting for data to arrive.
- **A lockup-free cache can reorder memory accesses.**
- This is useful in superscalar processors since execution can proceed, until the next instruction to execute actually needs the data from the cache miss.
- Another name for lockup-free caches is non-blocking caches.
- Data prefetching fetches data before it is needed and loads it into the cache (but not into a processor register — but the compiler may do so if it wishes anyway).
- Data prefetching requires the cache to be lockup-free — otherwise the processor would be stalled and the prefetching would be rather pointless.

Write Buffer Bypassing

We want our reads to be serviced as quickly as possible

- Between the L1 cache and L2 cache is a buffer, called the first level write buffer.
- A second level write buffer is located between the L2 cache and the bus interface.
- By letting read misses bypass writes in the buffer to other addresses, the reads can be serviced faster.
- **Such bypasses reorders the memory accesses.**
- A superscalar processor has queues in the load-store functional unit where accesses also can be reordered.

Sequential Consistency

- Sequential consistency (SC) was published by Leslie Lamport in 1979 and is the simplest consistency model.
- Neither Java, Pthreads, nor C11/C++ require it. They work on relaxed memory models.
- Sequential consistency can be seen from the programmer as if the multiprocessor has no cache memories and all memory accesses go to memory, which serves one memory request at a time.
- This means that program order from one processor is maintained and that all memory accesses made by all processors can be regarded as atomic (i.e. not overlapping).

Definition of SC

- Lamport's definition: ***A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.***
- Consider the program execution:

```
int A = B = C = 0;

T1:             T2:             T3:
A = 1;
                if (A)
                    B = 1;
                                if (B)
                                    C = A;
```

- Since all memory accesses are atomic, writes are seen in the same order so T3 must read the value 1 when reading A.

Dekker's Algorithm

```
bool    flag[2] = { false, false };
int     turn = 0;

void work(int i)
{
        for (;;) {
                flag[i] = true;
                while (flag[!i]) {
                        if (turn != i) {
                                flag[i] = false;
                                while (turn != i)
                                        ;
                                flag[i] = true;
                        }
                }

                /* critical section */

                /* leave critical section */

                turn = !i;
                flag[i] = false;

        }
}
```

- SC ensures that Dekker's algorithm works.

Implementing SC in a System Without Caches

- We will examine the effects of each of the following hardware optimizations:
  - Write buffer with read bypass
  - Overlapping writes
  - Non-blocking reads
- As we will see, none of these can be used even if there are no caches.
- This reduces performance considerably.

Write Buffer with Read Bypass

- Assume a bus-based multiprocessor.
- Since there are no caches, the write buffer, a FIFO queue, sits between the CPU and the bus interface.
- With read bypass, it is thus meant that a read skips the queue in the buffer and goes first to the bus before any write (to a different address).
- In Dekker's algorithm both CPUs can set their `flag[i]` to true and put that write into it's write buffer.
- Then the reading of the other thread's flag will bypass the write in the write buffer.
- When bypassing the old values of the `flag[!i]` can be returned (e.g. if there were other writes before in the buffers) and both can enter the critical section!

The read bypass destroys the atomicity and hence the sequential order.

Overlapping Writes

- In a bus-based system, the FIFO write buffer queue ensures that all writes from the CPU are ordered.
- In a general topology, however, different nodes typically are located at different distances and writes easily can arrive in an order different from the program order.
- In the example with variables A, B, and C, the new value of B may reach T3 before A does which violates SC.

T2 should not be allowed to start its write to `b` ***before T3 has become aware of the write to*** `a`.

Non-Blocking Reads

- Consider the following example from the first lecture again.
- With speculative execution and non-blocking reads, T2 can read the value of a before it leaves the while-loop, which violates SC.

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        f = 1;
}

// called by T2
void w(void)
{
        while (!f)
                ;
        printf("a = %d\n", a);
}
```


Implementing SC in a System With Caches

- The three issues in systems without caches can violate SC also with caches.
- For example a read cache hit must not be allowed to precede a previous read miss.
- In addition, since there can be multiple copies of a variable, there must be a mechanism which controls e.g. whether a cached copy is allowed to be read — it is not if another processor just has modified it, for example.
- This mechanism is called a ***cache coherence protocol***.
- A cache coherence protocol has three main tasks, as we will see next.

Cache Coherence Protocols

1. At a write, the cache coherence protocol should either remove all other copies, including the memory copy, or send the newly written data to update each copy.
   - A protocol which uses the former technique is called a **write invalidate protocol** while the latter is called a **write update protocol**.
   - Which is best depends on the sharing behaviour of the application but write invalidate is almost always better. More details follow!
2. Detecting when a write has completed so that the processor can perform the next memory access.
3. Maintaining the illusion of atomicity — with memory in multiple nodes the accesses cannot be atomic but a SC machine must behave as if they are.

Detecting Write Completion

- Consider a write to a memory location which is replicated in some caches.
- How can the writing processor know when it's safe to proceed?
- The write request is sent to the memory where the data is located.
- The memory knows which caches have a copy (recall from the previous lecture this information is stored in a directory, e.g. as a bit vector).
- The memory then sends either updates or invalidations to the other caches.
- The receiving caches then must acknowledge they have received the invalidation message from memory.
- The acknowledgement is typically sent to the memory and then when all acknowledgements have been received, a message is sent to the writing processor (actually, its cache) to tell it the write has completed.
- After that, the processor can proceed.

Write Atomicity 1(2)

- There are two different problems:
  (1) Write atomicity for a particular memory location, i.e. ensuring all CPUs see the same sequence of writes to a variable.
  (2) Write atomicity and reading the modified value.
- For (1) it is sufficient to ensure that writes to the same memory location are serialized, but not for (2). See next page.
- The memory controller can easily enforce serialization.
- Assume writes from two different caches are sent to it.
- One of them must arrive first. The other can simply be replied to with a negative acknowledgement of "try again later!"
- When the cache receives that message it will simply try again and after a while it will be its turn.

Write Atomicity 2(2)

- Let us now consider (2): reading the modified value.
- A write has completed when all CPUs with a copy have been notified.
- However, if one CPU is allowed to read the written value before the write has completed, SC can be violated.

```
int A = B = C = 0;

T1:                T2:                T3:
A = 1;
                   if (A)
                       B = 1;
                                      if (B)
                                          C = A;
```

- Assume all variables are cached by all threads, and T2 reads A before T3 knows about the write.
- Then T2 can write to B which might be so close to T3 that T3 can read A from its cache before the invalidation of A reaches T3.
- The solution is to disallow any CPU from reading A before the write is complete, which can be implemented in write invalidate as for case (1).

Write Atomicity in Write Update Protocols

- Recall that in a write update protocol, instead of invalidating other copies, new values are sent to the caches which replicate the value.
- So A is sent to T2 and to T3.
- While it's tempting to read A for T2 it's not permitted to.
- In write update, the updates are done in two phases. First is the data sent, and all CPUs acknowledge they have received it. Second each CPU is sent a message that it may read the new data.
- There are other problems with write update as well, for instance updates may be sent to CPUs which no longer are interested in the variable, thus wasting network traffic.

Optimizing Compilers and Explicitly Parallel Codes

- We have now seen the restrictions on hardware so that it does not reorder memory accesses and thus violate SC.
- The same restrictions must of course be put on optimizing compilers.
- The compiler must preserve "source code order" of all memory accesses to variables which may be shared — but not the order of stack accesses or other data known to be private to the thread.
- Examples of optimizations which cannot (in general) be used:
  - Register allocation
  - Code motion out of loops
  - Loop reordering
  - Software pipelining
- It's easy to imagine that these restrictions will slow down SC.
- The solution has often been to compile code for uniprocessors and use the `volatile` qualifier for shared data.
- Recall that volatile in C is different from volatile in Java!

Parallelizing Compilers

- If the compiler is doing the parallelization, these restrictions don't apply since the compiler writer hopefully knows what he or she is doing!
- After this course, however, you will probably not have too high hopes for automatic parallelization, except for numerical codes.
- In my view, parallelization of "normal" programs needs so drastic changes to the source code that automatic tools hardly can do that very well.
- But I may be wrong of course!

Cache Coherence Protocol States

- The cache coherence protocol maintains a state for each cache and memory block.
- The cache state can for instance be:
  - SHARED
  - INVALID
  - EXCLUSIVE — memory and this cache has a copy but it's not yet modified.
  - MODIFIED — only this cache has a copy and it's modified
- There are similar states for a memory block and also the bit-vector with info about which cache has a copy.
- In addition, a memory block can be in a so called **transient state** when acknowledgements are being collected, for instance.

Memory Access Penalty

- The time the processor is stalled due to waiting for the cache is called the memory access penalty.
- Waiting for read cache misses is difficult to avoid in any computer with caches (reminder: the Cell's SPU's have no caches...)
- Waiting for obtaining exclusive ownership of data at a write access is one of the disadvantages for SC
- How significant it is depends on the application
- Of course, once the CPU owns some data, it can modify it's cached copy without any further write access penalty, until some other CPU also wants to access that data, in which case the state becomes SHARED again.

Optimizing SC in Hardware

- Data prefetch can either fetch data in shared or exclusive mode
- By prefetching data in exclusive mode, the long delays of waiting for writes to complete can possibly be reduced or eliminated.
- Since exclusive mode prefetching invalidates other copies it can also increase the cache miss rate.
- Somewhat more complex cache coherence protocols can monitor the sharing behaviour and determine that it probably is a good idea to grant exclusive ownership directly instead of only a shared copy which is then likely followed by an ownership request.
- In superscalar processors it can be beneficial to permit speculative execution of memory reads. If the value was invalidated, the speculatively executed instructions (the read and subsequent instructions) are killed and the read is re-executed.

Optimizing SC in the Compiler

- Another approach is to have a memory read instruction which requests ownership as well.
- This can be done easily in optimizing compilers but needs a new instruction.
- It's very useful for data which moves from different caches and is first read and then written:

```
p->a += 1;
p->b += 1;
p->c += 1;
```

- Here the compiler can very easily determine that it's smarter to request ownership while doing the read.

Summary of Sequential Consistency

- Recall the two requirements of SC:
  1. Program order of memory accesses
  2. Write atomicity
- While SC is "nice" since it's easy to think about, it has some serious drawbacks:
  - The above two requirements... :-)
  - ...which limit compiler and hardware optimizations, and...
  - introduce a write access penalty
- The write access penalty is due to the processor cannot perform another memory access before the previous has completed.
- This is most notable for writes, since read misses are more difficult to optimize away by any method
- We will next look at relaxed memory consistency models

Relaxed Memory Models

- Relaxed memory models do not make programming any more complex (many would disagree however).
- However, you need to follow additional rules about how to synchronize threads.
- C11, Pthreads and Java are specified for relaxed memory models.
- In relaxed memory models, both the program order of memory references and the write atomicity requirements are removed and are replaced with different rules.
- For compiler writers and computer architects this means that more optimizations are permitted.
- For programmers it means two things:
  1. you must protect data with special system recognized synchronization primitives, e.g. locks, instead of normal variables used as flags.
  2. your code will almost certainly become faster, perhaps by between 10 and 20 percent, due to eliminating the write access penalty.

System Recognized Locks

- Under SC, you normally must protect your data using locks to avoid data races
- However, there are programs in which data races are acceptable
- Data races are forbidden in C11 and result in undefined behavour.
- Under SC you can write your own locks by spinning on a variable `int flag` as you wish.
- Under relaxed memory models you should use whatever the system provides.

Pthreads in Linux

- On UNIX the system recognized locks normally mean Pthreads mutexes.
- On Linux Pthreads are implemented with a low-level lock called a Futex — fast mutex.
- These have the advantage of not involving the Linux kernel in the case when there is no contention for a lock
- If many threads want a lock and must wait, the futex mechanism will ask the kernel to make a context switch to a different thread.
- Short time busy waiting for a release of a lock is better than context switching but if the predicted waiting time is long enough to justify a context switch, that will be performed if you use a futex.

Write Your Own Locks!?!?

- You should not need to, but if you wish, you can write your own locks.
- That is not so difficult on SC.
- As we will see shortly, to do so on relaxed memory models requires the use of special **synchronization machine instructions** which must be executed as part of the lock and unlock.
- Use e.g. inlined assembler to specify them.

Relaxed Memory Models

- Recall that relaxed memory models relax the two constraints of memory accesses: program order and write atomicity.
- There are many different relaxed memory models and we will look only at a few.
- We have the following possibilities of relaxing SC.
- Relaxing A to B program order: we permit execution of B before A.
  1. write to read program order to different addresses
  2. write to write program order to different addresses
  3. read to write program order to different addresses
  4. read to read program order to different addresses
  5. read other CPU's write early
  6. read own write early
- Different relaxed memory models permit different subsets of these.

Assumptions

- All writes will eventually be visible to all CPUs.
- All writes are serialized (recall: this can be done at the memory by letting one write be handled at a time — other pending writes must be retried).
- Uniprocessor data and control dependences are enforced.

Relaxing the Write to Read Program Order Constraint

- Obviously different addresses are assumed!
- A read may be executed before a preceding write has completed.
- With it, Dekker's Algorithm fails, since both can enter the critical section. ***How?***

```
bool    flag[2] = { false, false };
int     turn = 0;

void work(int i)
{
        for (;;) {
                flag[i] = true;
                while (flag[ti]) {
                        if (turn != i) {
                                flag[i] = false;
                                while (turn != i)
                                        ;
                                flag[i] = true;
                        }
                }

                /* critical section */

                turn = !i;
                flag[i] = false;

                /* not critical section */
        }
}
```


Some Models Which Permit Reordering a (Write, Read) Pair

- Processor consistency, James Goodman
- Weak Ordering, Michel Dubois
- Release Consistency, Kourosh Gharachorloo
- IBM 370, IBM Power
- Sun TSO, total store ordering
- Sun PSO, partial store ordering
- Sun RMO relaxed memory order
- Intel X86

Additionally Relaxing the Write to Write Program Order Constraint

- Recall the program below.

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        f = 1;
}

// called by T2
void w(void)
{
        while (!f)
                ;
        printf("a = %d\n", a);
}
```

- By relaxing the write to write program order constraint, the write to `f` may be executed by T_1 even before the function call to `u`, resulting in somewhat unexpected output.

Some Models Which Permit Reordering a (Write, Write) Pair

- Weak Ordering
- Release Consistency
- IBM Power
- Sun PSO, partial store ordering
- Sun RMO relaxed memory order

Relaxing All Memory Ordering Constraints

- The only requirements left is the assumption of uniprocessor data and control dependences.
- Models which impose no reordering constraints for normal shared data include:
  - Weak Ordering
  - Release Consistency
  - IBM Power
  - Sun RMO relaxed memory order
- These consistency models permit very useful compiler and hardware optimizations and both Java and Pthreads (and other platforms) require from the programmer to understand and use them properly!
- In the preceding example, the two reads by T_2 are allowed to be reordered.
- The obvious question then becomes: how can you write a parallel program with these memory models???

What do you say?

Special Machine Instructions for Synchronization

- The short answer is that machines with relaxed memory models have special machine instructions for synchronization.
- Consider a machine with a **sync** instruction with the following semantics:
  - When executed, all memory access instructions issued **before** the sync must complete before the sync may complete.
  - All memory access instructions issued **after** the sync must wait (i.e. not execute) until the sync has completed.
  - Assume both T_1 and T_2 have cached `a`.

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        asm("sync");
        f = 1;
}

// called by T2
void w(void)
{
        while (!f)
                ;
        asm("sync");
        printf("a = %d\n", a);
}
```

- With `asm` we can insert assembler code with gcc and most other compilers.
- The sync instructions are required, as explained next...

Sync Instruction Example 1(3)

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        asm("sync");
        f = 1;
}

// called by T2
void w(void)
{
        while (!f)
                ;
        asm("sync");
        printf("a = %d\n", a);
}
```

- The write by T_1 to a results in an invalidation request being sent to T_2.
- At the sync, T_1 must wait for an acknowledgement from T_2.
- When T_2 receives the invalidation request, it acknowledges it directly and then puts it in a queue of incoming invalidations.
- When T_1 receives the acknowledgement, the write is complete and the sync can also complete, since there are no other pending memory accesses issued before the sync.

Sync Instruction Example 2(3)

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        asm("sync");
        f = 1;
}

// called by T2
void w(void)
{
        while (!f)
                ;
        asm("sync");
        printf("a = %d\n", a);
}
```

- The write by T_1 to `f` also results in an invalidation request being sent to T_2.
- When T_2 receives the invalidation request, it acknowledges it directly and then puts it in the queue of incoming invalidations.
- T_2 is spinning on `f` and therefore requests a copy of `f`.
- When that copy arrives, with value one, it must wait at the `sync` until the invalidation to `a` has been applied — it might already have been.
- Without the sync by T2 its two reads could be reordered:
  - The compiler could have put `a` in a register before the while-loop.
  - The CPU could speculatively have read `a` from memory.
  - The incoming transactions may be reordered in a node.

Sync Instruction Example 3(3)

- Instead of this sync instruction used in order to be concrete, we can use the Linux kernel memory barrier:

```
int a, f;

// called by T1
void v(void)
{
        a = u();
        smp_mb();
        f = 1;
}

// called by T2
void w(void)
{
        while (!f)
                ;
        smp_mb();
        printf("a = %d\n", a);
}
```

- The memory barrier is a macro which will expand to a suitable machine instruction.

Weak Ordering

- The memory consistency model introduced with the sync instruction is called Weak Ordering, WO, and was invented by Michel Dubois.
- The key ideas for why it makes sense are the following:
  - Shared data structures are modified in critical sections.
  - Assume N writes to shared memory are needed in the critical section.
  - In SC the processor must wait for ***each*** of the N writes to complete in sequence.
  - In WO, the processor can pipeline the writes and only wait at the end of the critical section.
  - Sync instructions are then executed as part of both the lock and unlock calls.
- Of course, some or all of the N writes may be to already owned data in which case there is no write penalty.
- Measurements on different machines and applications show different values but 10-20 % percent of the execution time can be due to writes in SC.

Release Consistency

- Release Consistency, RC, is an extension to WO, and was invented for the Stanford DASH research project.
- Two different synchronization operations are identified.
- An **acquire** at a lock.
- A **release** at an unlock.
- An acquire orders all subsequent memory accesses, i.e. no read or write is allowed to execute before the acquire. Neither the compiler nor the hardware may move the access to before the acquire, and all acknowledged invalidations that have arrived before the acquire must be applied to the cache before the acquire can complete (i.e. leave the pipeline).
- A release orders all previous memory accesses. The processor must wait for all reads to have been performed (i.e. the write which produced the value read must have been acknowledged by all CPUs) and all writes made by itself must be acknowledged before the release can complete.

WO vs RC

- Recall: by $A \rightarrow B$ we mean B may execute before A
- WO relaxation: data $\rightarrow$ data
- RC relaxation:
  - data $\rightarrow$ data
  - data $\rightarrow$ acquire
  - release $\rightarrow$ data
  - release $\rightarrow$ acquire
- acquire $\rightarrow$ data: forbidden
- data $\rightarrow$ release: forbidden
- In practice not very significant difference on performance.

Succeeding in Progamming with Relaxed Memory Models

- The bad news are that an execution might produce correct output 99% of the time and suffer painful data races only rarely.
- Recall Helgrind can only detect when things are wrong but not prove the code cannot have any data race!
- If you use system recognized locks such as Pthreads mutexes or Java synchronization you are likely to have more fun.

High-level Parallel Programming: Scala on the JVM

Contents of Lecture 4

- The Scala programming language
- Parallel programming using actors

Purpose of this Lecture

- That you will understand why Scala is interesting.
- You will see examples of what is different between Scala and Java.
- You will understand how and why Scala and Java can be used in the same program.
- You will understand the key concepts of message passing using **actors**, which were first described by Carl Hewitt at MIT 1973. Essentially, an actor is a thread — however, there are certain differences as we will see.
- You will understand enough about Scala actors that you can write a parallel version of the dataflow program in Scala (to save you time for the lab, you can download an almost complete version).

Scala and the JVM

- Martin Odersky designed Generic Java and the Java compiler `javac` for Sun. He is a professor at EPFL in Lausanne.
- The Scala language produces Java byte code and Scala programs can use existing Java classes.
- When you run a Scala program, the JVM cannot see any difference between Java and Scala code.
- You can download Scala from `www.scala-lang.org`.
- It's easy to get started.
- Scala is not tied to Java byte code however and can produce output also for the Dalvik VM (for Android) and .Net.
- So what is Scala then?

Scala is a Functional and Object Oriented Language

- It is intended to be **scalable** and suitable to use from very small to very large programs.
- The Scala compiler, `scalac`, usually can infer the types of variables so you don't have to type them.

```
var capital = Map("Denmark" -> "Copenhagen", "France" -> "Paris", "Sweden" -> "Stockholm");

capital += ("Germany" -> "Berlin");

println(capital("Sweden"));
```

- There is no need to declare the type of the variable `capital` since `scalac` can do it for you.
- Less typing can potentially lead to faster programming — at least if the tedious part of the typing can be eliminated.

Shorter Class Declarations

- A short class declaration in Scala:

```
class MyClass(index: Int, name: String)
```

- The same class in Java:

```
class MyClass {
    private int index;
    private String name;

    public MyClass(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```


Redefinable Operators as in C++

```
def factorial(x: BigInt): BigInt = if (x == 0) 1 else x * factorial(x - 1)
```

- A function is defined using `def` and `=` and an expression.
- Or using the Java class `BigInteger`:

```
import java.math.BigInteger;

def factorial(x: BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
```

- Which version do you prefer?

Scala is Statically Typed

- Lisp, Smalltalk, Ruby, Python and many other languages are dynamically typed, which means type checking is performed at runtime.
- Scala and to a very large extent also C are statically typed.
- Of course, C is a very small language and much easier to type check.
- For C, if you use `<stdarg.h>` (which you usually shouldn't) or insane casts (which result in undefined behaviour = serious bug) the C compiler will not help you.
- Look at this program:

```
(defun sumlist (h)
       (if (null h) 0 (+ (car h) (sumlist (cdr h)))))

(setq b '(1 2 3 4))
(setq c '("x" "y" "z"))

(print (sumlist b))
(print (sumlist c))
```

- Which language is it?
- When is the error detected during dynamic type checking?

Answers

```
(defun sumlist (h)
       (if (null h) 0 (+ (car h) (sumlist (cdr h)))))

(setq b '(1 2 3 4))
(setq c '("x" "y" "z"))

(print (sumlist b))
(print (sumlist c))
```

- The language is Common Lisp.
- The error is detected when adding the string `"z"` to zero:

```
> time clisp a.lisp

10
*** - +: "z" is not a number


real    0m0.062s
user    0m0.045s
sys     0m0.017s
```


A C Compiler Must Issue a Diagnostic Message

```
#include <stdlib.h>

typedef struct list_t   list_t;

struct list_t {
        list_t*         next;
        int             value;
};

list_t* cons(int value, list_t* list)
{
        list_t*         p;

        p = malloc(sizeof(list_t));
        if (p == NULL)
                abort();
        p->value = value;
        p->next = list;

        return p;
}

int sumlist(list_t* h)
{
        return h == NULL ? 0 : h->value + sumlist(h->next);
}

int main(void)
{
        list_t*         p;
        list_t*         q;

        p = cons(1, cons(2, cons(3, cons(4, NULL))));
        q = cons("x", cons("y", cons("z", NULL)));      // static type error
}
```


GCC Output

```
> time gcc a.c
a.c: In function 'main':
a.c:35: warning: passing argument 1 of 'cons' makes integer from pointer without a cast
a.c:11: note: expected 'int' but argument is of type 'char *'
a.c:36: warning: passing argument 1 of 'cons' makes integer from pointer without a cast
a.c:11: note: expected 'int' but argument is of type 'char *'
a.c:35: warning: passing argument 1 of 'cons' makes integer from pointer without a cast
a.c:11: note: expected 'int' but argument is of type 'char *'

real    0m0.150s
user    0m0.103s
sys     0m0.047s
```


Clang Output

```
> time clang -S a.c
a.c:35:31: warning: incompatible pointer to integer conversion passing
      'char [2]' to parameter of type 'int'
        q = cons("x", cons("y", cons("z", NULL)));
                                ^~~
a.c:11:18: note: passing argument to parameter 'value' here
list_t* cons(int value, list_t* list)
                 ^
a.c:35:21: warning: incompatible pointer to integer conversion passing
      'char [2]' to parameter of type 'int'
        q = cons("x", cons("y", cons("z", NULL)));
                    ^~~
a.c:11:18: note: passing argument to parameter 'value' here
list_t* cons(int value, list_t* list)
                 ^
a.c:35:11: warning: incompatible pointer to integer conversion passing
      'char [2]' to parameter of type 'int'
        q = cons("x", cons("y", cons("z", NULL)));
                 ^~~
a.c:11:18: note: passing argument to parameter 'value' here
list_t* cons(int value, list_t* list)
                 ^
3 warnings generated.

real    0m0.050s
user    0m0.030s
sys     0m0.020s
```


A Scala Compiler Must Issue a Diagnostic Message

```
class Test {
  def sumlist(h:List[Int]) : Int = if (h.isEmpty) 0 else h.head + sumlist(h.tail);
  var a = List(1,2,3,4);
  var b = sumlist(a);
  var c = List("x", "y", "z");
  var d = sumlist(c);
}

> time scalac a.scala
a.scala:6: error: type mismatch;
 found   : List[java.lang.String]
 required: List[Int]
  var d = sumlist(c);
                  ^
one error found

real    0m11.884s
user    0m11.653s
sys     0m0.196s
```

- The type analysis for Scala is of course more complex than for C.
- Don't use a compiler with this compilation speed for multi million lines software — probably you don't have to because Scala programs are shorter...
- Compilation speed of Scala will probably improve significantly in the future.

The Fast Scala Compiler

- There is a server program `fsc` which is faster than `scalac` because it avoids some initializations.
- Clang and Common Lisp were fastest.
- `clisp` was originally written in assembler and Lisp for Atari machines but has been rewritten in portable C and Lisp.

Scala Basics: `val` vs `var`

- Writing `var a = 1`, we declare an initialized `Int` variable that we can modify.
- With `val a = 1`, a becomes readonly instead.
- The following declares an array:

```
val a = new Array[String](2);
a(0)  = "hello";
a(1)  = "there";
```

- Note that it is a that is readonly, not its elements.
- We can iterate through an array like this, for example:

```
for (s <- a) println(s);
```

- We should not declare the variable `s`.

Numbers are Objects

- Consider

```
for (i <- 0 to 9) println(i);
```

- Here the zero actually is an object with the method `to`.
- In many cases a method name can be written without the dot but rather as an operator.

Companion Classes

- In Java you can have `static` attributes of a class which are shared by all objects of that class.
- In Scala, you instead create a **companion class** with the keyword **object** instead of **class**:

```
object A {
  var a = 44;
}

class A {
  println("a = " + A.a);
}

object Main {
  def main(args: Array[String]) {
    val a = new A;
  }
}
```

- By default attributes are `public` in all Scala classes, but an object may access a private attribute of its companion class.

Class Declarations with Attributes

- You can declare a class like this:

```
class B(u: Int, var v: Int) {
  def f = u + v;
}
```

- The parameters of a constructor by default become `val` attributes of the class.
- Therefore only `v` can be modified.
- Even if you only need the parameters in the constructor, they become attributes and cheerfully consume memory for you.

Code Reuse in Scala using Traits

- Scala uses single inheritance as Java with the same keyword `extends`.
- Instead of Java's interfaces which only provide abstract methods, Scala has the concept of a `trait`.
- Unlike an interface, a trait can contain attributes and code, however.
- A trait is similar to a class except that the constructor cannot have parameters.

```
object Main {
  def main(args: Array[String]) {

    val a = new C(44);

    a.hello;
    a.bye;
  }
}

class A(u: Int) {
  def bye { println("bye bye with u = " + u); }
}

trait B {
  def hi { println("hello"); }
}

class C(v: Int) extends A(v) with B {
  def hello { hi; }
}
```

Lists

- The standard class `List` is singly linked and consists of a pair of data and a pointer to the next element.
- An empty list is written either as `Nil` or `List()`.
- Five (of many) methods are:
  - `::` — create a list: `val h = 1 :: 2 :: 3 :: Nil`, which means: `val h = (1 :: (2 :: (3 :: Nil)))`.
    This can also be written as `val h = new List(1, 2, 3)`
  - `:::` — create a new list by concatenating two lists.

```
val a = new List(1, 2, 3);
val b = new List(4, 5, 6);
val c = a ::: b;
```

  - `isEmpty` — boolean
  - `head` — data in first element
  - `tail` — the rest of the list starting with the 2nd element.

Reversing a List 1(2)

```
def rev[T](h:List[T]) : List[T] = {
  if (h.isEmpty)
    h;
  else
    rev(h.tail) ::: List(h.head);
}
```

- This function is generic with element type `T`.
- It's not the most efficient way to reverse a list since list concatenation must traverse the left operand list.
- This version of reverse has quadratic time complexity.
- How can we do it in linear time?

Reversing a List 2(2)

```
def rev1[T](h : List[T], q : List[T]) : List[T] = {
  if (h.isEmpty)
    q;
  else
    rev1(h.tail, h.head :: q);
}

def rev[T](h : List[T]) : List[T] = {
  rev1(h, List());
}
```

- Better.

Pattern Matching in Scala

- Pattern matching means we provide a sequence of cases against which the input data is matched.
- The first case that matches the input data is executed.

```
def rev[T](xs: List[T]) : List[T] = xs match {
  case List()   => xs;
  case x :: xs1 => rev(xs1) ::: List(x);
  }
```

- The reverse of the empty list is the parameter `xs`.
- The non-empty list matches a list with at least one element, as in the second case.
- Pattern matching is used extensively in functional programming.
- We will use pattern matching when receiving actor messages.

Programming with Actors in Scala

- Programming with actors is in one sense just writing another multithreaded program.
- In another sense it's completely different because you should use no locks or condition variables or the like.
- An actor is like a thread which sits and waits for a message to arrive.
- In Scala you can have two kinds of actors which differ in how they wait for messages:
  - using `receive` syntax: one JVM thread per actor, or
  - using `react` syntax: essentially one JVM thread per CPU which handles numerous actors.

React vs Receive

- With `receive` the response time should be faster.
- With `react` you can have hundreds of thousands actors without too much overhead.
- For instance, in my version of the dataflow program, I have two actors per vertex using `react`.
- There are certain limitations when using `react` such as when waiting using `react` the actor's stack frame is discarded when it starts waiting — you cannot return to anywhere since no return address is remembered.
- Therefore no value can be returned either.
- These limitations don't make it noticably difficult to use `react`.

Sending a Message

- With actors, messages are sent to an actor — and not to a mailbox or channel as in other systems.
- Assume one actor `A` has a reference to another actor `B`, then `A` can send a message type `C` to `B` using:

```
B ! new C();
```

- The sending actor immediately continues execution without waiting for the message to arrive.
- The receiving actor may live in the same machine or in another country.

Evaluating a Message

- With pattern matching on the message, the action to perform is selected.
- If there is no match, the message is discarded.
- After performing it, the actor repeats the waiting for another message, or does nothing which terminates it.
- It's not necessarily easier to program with actors than with locks.
- For instance, you can end up with a deadlock if two actors are waiting for messages from each other.
- A message is not an interrupt: only when the actor has finished one message, it will evaluate the next.

Message Arrival Order

- In some actor-based systems the arrival order of messages is not specified — even for messages from the same sender.
- For Scala actors, messages sent from one actor to another always arrive in the same order as they were sent.
- Consider the liveness algorithm and assume each vertex is an actor.
- Assume a vertex/actor sends a message to each of its successor vertices requesting the successor's in-set.
- Obviously, the actor cannot expect the replies will arrive in any particular order. Instead the replies can be counted to see whether all have arrived.

Copying Data or Immutable Messages

- By copying instead of sharing data you can avoid data races.
- Since copying degrades performance, and our machine has shared memory, you probably want to write your Scala program without copying data.
- Instead of copying the same data **for each message** you send, you can do as follows:
  - Create the new data.
  - When another actor wants your data, send a reference to it.
  - Forbid the receiving actors from modifying the data.
- If you need to recompute your data, do that in a different object (e.g. a newly created one).
- By having the messages immutable, there cannot be any data race.
- One of the research projects on Scala in Lausanne is to try to do static analysis to determine whether a Scala program certainly is free from data races.
- Thus, it's not guaranteed by the language but by your design — and in the future checked by the Scala compiler.

Declaring an Actor

- An actor can be declared as follows:

```
case class A();
case class B();

class Controller extends Actor {
  def act() {
    react() {
      case A() => {
        println("recieved an A " + sender);
        act();
      }

      case B() => {
        println("recieved a B and says goodbye");
      }
    }
  }
}
```

- To send two messages to a controller, we can do as follows:

```
val controller = new Controller();

controller ! new A();
controller ! new B();
```

- The messages can have parameters as any other class constructor.
- Simple and elegant syntax.
- The sending actor is available as `sender`.

Receiving a Message

- When an actor has determined which case matched, it will execute undisturbed by other incoming messages.
- The execution of a message is any Scala code.
- If the actor wants to proceed and wait for new messages, it should end with a recursive call to `act()`.
- The Scala compiler eliminates this tail recursion so new stack space will not be used.

Implementation of Actors in Scala

- Scala actors are implemented using Java threads.
- Recall, there are two constructs to receive messages:
  - `react`
  - `receive`
- An actor waiting with `receive` uses its own JVM thread.
- Instead, with `react` Java threads are shared between different actors.
- In a `receive` the actor calls the Java `wait` method and is resumed when either `notify` or `notifyAll` is called.
- At the `react` statement, the actor registers an event handler (the matching code) and the JVM thread is free to work for some other actor.
- When a message arrives to an actor waiting with `react`, an existing thread can process the message. Recall:
  - Using `react` the Scala runtime library creates as many Java threads as the machine has processors.
  - If you use `react` you can create hundreds of thousands of actors without problems.

Actor Loop

- Instead of a tail recursive call to `act()` for waiting on a new message, we can write:

```
def act() {
  loop {
    react {
      case A() => println("got A");

      case B() => println("got B");
    }
  }
}
```

- A problem with this, however, is that the loop is infinte so your actors will never terminate.
- Instead of `loop`, we can use `loopWhile` which terminates, when its condition becomes false.

Combining React Actors

- The loops we just saw are examples of **combinators** which essentially let us continue execution after a `react` without a recursive call to `act()`.
- Another combinator is `andThen` which also let's us do something after we wake up:

```
{
  react { /* ... */ }
} andThen {
  react { /* ... */ }
}
```


Syntax for Synchronous Messages

- As mentioned, the sending actor does not wait for a reply, i.e. normal messages sent with `!` are asynchronous.
- We can send synchronous message using: `!?`

```
server !? new Request("hello") match {
  case response: String => println(response);
}
```

Futures

- There is a concept of so called **futures** which simply mean you don't wait until you need to — similar to data prefetching or properly used DMA in the Cell:

```
val future = server !! new Request("hello");

/* do other things while server is working */

/* invoking the future waits until it has arrived. */
println(future());
```

- An example of using futures is to request data from multiple sources in parallel and then wait for them to arrive before working on them.

Message Timeouts

- With `receiveWithin` we can specify a maximum waiting time.
- If no other message is received before that, a `TIMEOUT` is created and received.

```
receiveWithin(1000) {
  case msg: String => /* ... */
  case TIMEOUT:    => /* ... */
}
```


Programming Hints

- You can send the actor as part of a message, using `this`.
  - The receiving actor should use it for sending a reply only.
  - If the receiving actor would invoke methods in the received thread, you can create data races which you need to avoid.
- If an actor has created some data and then sends it to an actor A, then actor A should not modify that data.
- Communicate only through messages.
- The Scala program terminates when the last actor terminates.

Scala Dataflow Lab 2

- You can download a subset of the file I will go through next

Akka

- You will notice Scala actors are slow
- A faster implementation of the actor model is Akka
- It has some disadvantages though, which makes it less convenient to work with
- As you saw in `Dataflow.scala` you create and work with actor objects directly in the old model
- In Akka, you essentially use `void*` and don't know which type the reference to the actor object really is. It is called `ActorRef`

Creating Actors in Akka

```
class Vertex extends Actor { /* ... */ }
object Lab2 extends App {
  val system: ActorSystem = ActorSystem("Lab2")

  /* ... */

  val cfg    = new Array[ActorRef](nvertex);
  val nsucc  = new Array[Int](nvertex);
  val system = ActorSystem("Lab2");

  val controller = system.actorOf(Props[Controller], "controller");

  for (i <- 0 until nvertex) {
    nsucc(i) = (rand.nextInt() % maxsucc).abs;
    cfg(i) = system.actorOf(Props[Vertex], "vertex"+i);
  }

  /* ... */
}
```

- You cannot refer to attributes of e.g. a Vertex actor — due to the use of `ActorRef`
- Instead send a message and request the data
- I will give you a small Akka program and related project files.
- In Lab 2 you should do at least one of the actor implementations (either the old or Akka)

POSIX Threads

Contents of Lecture 5

- POSIX Threads architecture — threads, synchronization, and scheduling.
- Functions and types for starting and stopping threads
- Functions and types for synchronization
- The memory consistency model for POSIX Threads
- Thread cancellation
- Thread private data in POSIX Threads, GNU C, and C11
- POSIX Threads and OS kernel threads
- POSIX Threads realtime scheduling

POSIX Threads Overview

- POSIX stands for Portable Operating System Interface for UNIX.
- POSIX is thus not an implementation of UNIX but rather specifications of the APIs available for UNIX implementations.
- The standards in POSIX are specified by the IEEE.
- POSIX Threads are described by three parts:
  - Functions for runnning threads
  - Functions for synchronizing threads
  - Functions for scheduling threads

The `pthread_t` Type

- All types in POSIX Threads are opaque, which means client code should regard them as secret and not rely on any particular knowledge about them.
- `FILE` is another example of an opaque type.
- The most important type is: `phtread_t`.
- It can be regarded as a "key" to identify a thread between the client and the POSIX Threads implementation.
- When a thread is created, the client code passes a pointer to a variable of this type which is filled in by the library code.

Creating and Starting a POSIX Thread

```
int pthread_create(
        pthread_t*              thread,         // output param.
        const pthread_attr_t*   attr,           // input param.
        void*                   (*work)(void*), // a function.
        void*                   arg);           // param to work.

arg_struct_t arg_struct = { n, a, b, c }; // args to work.

status = pthread_create(&thread, NULL, work, &arg_struct);
```

- The thread identifier is filled in by the call and attributes are optional.
- The created thread runs the `work` function and then terminates.
- Typically multiple arguments are passed in a struct as above.

Return Value from `pthread_create()`

- A zero return value from `pthread_create()` indicates success, while a nonzero describes an error printable with `perror`.
- The `work` function can return a void pointer.
- Calling `pthread_join` waits for the termination of another thread and also gives access to the returned void pointer.
- A thread can only be joined once.

Terminating a Thread

- There are three ways to terminate a thread:

1. Return from the work function.
2. The thread can call the function `void pthread_exit(void* value)`. The Standard C Library function `void exit(int status)` should normally not be used since it terminates the entire program.
3. Calling the function `int pthread_cancel(pthread_t thread)` makes a request to terminate the specified thread. See cancellation below.

- The first two are used by the thread itself and the third is used to stop another thread.
- Stopping another thread can be useful e.g. when a user has hit a "Cancel" button or another thread already has found a winning chess move.

Thread Cancellation Overview

- Simply terminating a thread can be disastrous if for example it has locked a mutex and is modifying shared data.
- Therefore the `pthread_cancel` is asynchronous and simply requests that the thread should terminate. To actually know that the thread has terminated, it must be joined with.
- A thread that received a cancellation request is informed about this fact at certain points in the program, called **cancellation points**.
- The termination of the thread is started when it comes to such a cancellation point, if it has a **pending** cancellation request.
- A thread can install a function that is executed before the thread actually terminates.

Cancellation State and Type

- For cancellation, a thread has two variables each with two possible values.
- The variables cannot be accessed directly but only through function calls.
- They are:
  - **State** — cancellation is either enabled or disabled.
  - **Type** — cancellation is either asynchronous or deferred.
- These result in three different cancellation modes:
  1. **Disabled** — any cancellation request received is saved until cancellation is enabled in the future.
  2. **Deferred** — cancellation is started at a cancellation point if there is a pending cancellation request.
  3. **Asynchronous** — cancellation can start at any time.

Modifying the Cancellation Mode

- The functions to modify the cancellation mode returns the thread's old value of the respective variable.
- `int pthread_setcancelstate(int state, int* old);` The state must be one of:
  - `PTHREAD_CANCEL_ENABLE`
  - `PTHREAD_CANCEL_DISABLE`
- `int pthread_setcanceltype(int type, int* old);` The type must be one of:
  - `PTHREAD_CANCEL_DEFERRED`
  - `PTHREAD_CANCEL_ASYNCHRONOUS`
- The default mode for new threads is **deferred**.

Cancellation Points

- A number of functions are cancellation points, including

`pthread_cond_wait`	`pthread_cond_timedwait`
`pthread_testcancel`	`pthread_join`
`close`	`creat`
`open`	`read`
`system`	`wait`
`waitpid`	`write`

- POSIX guarantees that the above (and some others) are cancellation points.
- Another list contains possible cancellation points, including

`printf`	`scanf`
`fopen`	`fclose`

- ANSI C and POSIX functions not on any of those lists are guaranteed not to be cancellation points.

Receiving a Cancellation Request

- Thus if a cancellation request is received while cancellation is disabled, the request is simply blocked until it is enabled again.
- A pending request is delivered when the thread comes to a function which is a cancellation point.
- Changing the cancellation mode is **not** a cancellation point.
- At a cancellation point the thread first executes any installed cleanup handler (see below) and then terminates the thread.
- The return value from a cancelled thread is `PTHREAD_CANCELED` which thus is a valid value of a void pointer.

Installing Cleanup Handlers

- Each thread has a stack of cleanup handlers.
- They are installed with the function:
  `void pthread_cleanup_push(void (*func)(void*), void* arg);`
- The argument `arg` will be passed to `func` when it is executed.
- To remove a cleanup handler, use the function:
  `void pthread_cleanup_pop(int execute);`
- If the argument `execute` is nonzero, the cleanup handler will first be executed and then popped.
- When a thread is about to be terminated, all cleanup handlers on the stack are executed, starting with what is on the top of the stack.

Execution of Cleanup Handlers

- There are three situations when one or all cleanup handlers are executed:
  1. When a thread is being terminated due to a cancellation.
  2. When a thread is being terminated due to it has called `pthread_exit`.
  3. When it has called `pthread_cleanup_pop` with a nonzero parameter.
- In the last case, only one cleanup handler is executed, as we just saw.

Synchronization

- POSIX Threads has three main primitives for synchronization:
  - Mutex
  - Condition variable
  - Barriers

Avoid Synchronization!

- Ideally a parallel program needs no synchronization.
- Synchronization and therefore data communication between threads/caches take time.
- Some problems can be divided into suitable tasks statically.
- However, a common problem if T tasks are statically assigned to P threads is that some tasks take more time and therefore there becomes an inbalance in the work load, i.e. some threads take much longer time than the others.
- We will next look at a program with dynamic task assignment.

Mutex

- A POSIX Threads **mutex** is a lock with a sleep queue — and not a spin lock.
- The type is `pthread_mutex_t` and the most important functions related to it are:
- `pthread_mutex_init`
- `pthread_mutex_destroy`
- `pthread_mutex_lock`
- `pthread_mutex_trylock`
- `pthread_mutex_unlock`
- All five take a pointer to a `pthread_mutex_t`, and `pthread_mutex_init` also takes a pointer to attributes, which may be `NULL`. We will look at the attributes below.

A Worklist

- Assume we have a number of tasks to be processed.
- We put the tasks in a list and create threads which take tasks from the list and process them.
- Concurrently adding or removing of items in the list means the list must be protected, otherwise the program suffers data races and a likely disaster.
- Two alternatives:
  1. Put a mutex lock in each list head, i.e. protect the data.
  2. Use a common mutex for all lists, i.e. protect the code.
- Which is best depends on the application. There can be more concurrency if each list head has its own lock, at the cost of memory...

Condition Variables

- A condition variable lets a thread wait for something to happen in the future, and another thread to inform it that it has happened.
- For example: a worker thread can wait for a task being inserted in the list and the master can signal any waiting worker thread that it just has inserted a new task.
- The POSIX Thread condition variable type is: `pthread_cond_t`.
- In addition to initialization and destruction functions the main functions are:
  - `pthread_cond_wait` — causes calling thread to wait
  - `pthread_cond_signal` — wakes up one waiting thread
  - `pthread_cond_broadcast` — wakes up all waiting threads

Pthread Mutex and Condition Variable

- Normally a thread has taken a mutex and then inspects the state of a data structure.
- If the thread must wait for a change in the state by some other thread it needs to call `pthread_cond_wait`.
- It must also unlock the mutex, otherwise no other thread can change the state.
- Suppose it wrongly would first unlock the mutex and then wait. Then by Murphy's Law (if something bad can happen, it will) the other thread signals the condition just before our thread calls `pthread_cond_wait` and it will therefore never wake up.
- On the other hand, it cannot first go to sleep and unlock the mutex...
- The solution is to do both operations in the call to `pthread_cond_wait` and therefore a pointer to the mutex is also passed as an argument to `pthread_cond_wait`.
- When the thread wakes up again, it will have the mutex locked.

More Details

- One mutex may be used for multiple condition variables, such as a mutex for protecting a buffer with the condition variables to signal to a consumer thread that a buffer is no longer empty or to a producer that it is no longer full.
- Two threads wanting to wait for the same condition variable **must** use the same mutex.
- It is legal to signal a condition variable without having locked the corresponding mutex, but not so common.

Predicate, Condition Variable, and Mutex

- The logic expression in the C code which decides whether a thread should wait on the condition variable is called the **predicate** associated with the condition variable.
- The predicate is computed from shared data which different threads can modify, and therefore that data must be protected using a mutex.
- For example, the predicate may be computed by a boolean function:

```
bool empty(buffer_t* buffer);
```

- The predicate should **always** be tested in a loop and **not** in an if-statement.
- Strictly speaking, some programs for example some with only two threads, might work correctly without a loop but you will enjoy life more if you use loops. See next slide.

Why You Need A Loop

- You should write your code like this.

```
pthread_mutex_lock(&mutex);
while (!predicate())
       pthread_cond_wait(&cond, &mutex);
/* do something... */
pthread_mutex_unlock(&mutex);
```

- There are at least three reasons for doing so:
  1. **Intercepted wakeups:** Another thread might have locked the mutex before yours.
  2. **Loose predicates:** This is a kind of predicate that says "the predicate may be true (but check before relying on it)."
  3. **Spurious wakeups:** This is very uncommon and is essentially an error that a thread was woke up without any good reason.

Intercepted Wakeups

- Should you signal a condition variable before or after you unlock its associated mutex (in case you have it locked) ???
- If you signal first, then the woke up thread will immediately try to lock the mutex and find it locked and wait again, now instead on a mutex, causing unnecessary context switching and synchronization overhead, both in the form of instructions and cache misses.
- If you unlock first, another thread may take the lock before the thread you wake up. That is not wrong, of course.
- It means the predicate might not longer be true after the other thread has unlocked the lock and it is your turn.
- This is called an intercepted wakeup.
- Due to intercepted wakeups, you must check the condition in a loop.

Loose Predicates

- It may be more convenient and/or efficient to say "you might have something interesting to check out" rather promising something.
- The woke up thread then must itself determine if there really was something for it, or whether it should continue waiting.
- This is called a loose predicate.

Spurious Wakeups

- When you hit CTRL-C to terminate a program you send a so called UNIX signal to it. This use of the word signal has nothing to do with the signal function of condition variables.
- When a thread receives a UNIX signal and was in the UNIX kernel waiting for a system call to complete, that system call is terminated and returns with the error code `EINTR`.
- Some UNIX signals are sent to all threads of a running program (called a **process**) and a thread waiting on a condition variable, i.e. in a system call on UNIX will thus be interrupted to handle the signal.
- An interrupted system call is not resumed but the application proceeds after it has returned.
- In principle a system call used by `pthread_cond_wait` could be interrupted and result in a spurious wakeup, but that is not the behaviour on Linux which uses the **futex** system call described below.

Implementation of Linux Native Pthreads Library

- Should a mutex lock involve the Linux kernel?
- Preferably not!
- On Linux there is a low-level synchronization primitive called **futex** which is used to implement the Pthreads library.
- We will next look at the implementation of `pthread_mutex_lock`.

User Level Locking with Futex in Linux

- Originates from IBM Reseach and the IBM Linux Technology Center.
- Implemented in the GNU C Library and in the Linux kernel, since version 2.5.7.
- The lock variable is in user space in shared memory and there is a corresponding wait queue for a lock in the kernel.
- The fast case is when there is no contention for the lock and therefore the kernel needs not be involved.
- The lock is manipulated in user space with atomic instructions.

Initialization of a Pthread Mutex

```
pthread_mutex_t       mutex = PTHREAD_MUTEX_INITIALIZER;
```

- This initialises the mutex with default attributes.
- `PTHREAD_MUTEX_INITIALIZER` is a constant expression, meaning we can initialise a mutex like this at file scope (static storage, ie a `static` or global variable).
- The mutex starts out as unlocked (of course).
- If allocated by eg malloc, then `pthread_mutex_init()` should be called.
- After usage, `pthread_mutex_destroy` should be called for a mutex, and then its memory should be deallocated using `free`, if appropriate.

pthread_cond_timedwait

- To wait on a condition variable with a time out, use `pthread_cond_timedwait`.

```
int pthread_cond_timedwait(
        pthread_cond_t*,
        pthread_mutex_t*,
        struct time_spec*);
```

- The time is absolute time and to wait eg for at most 3 seconds, one can use:

```
timeout.tv_sec = time(NULL) + 3;
timeout.tv_nsec = 0;
```

- If there is a time out, the return value is `ETIMEDOUT`.

Initialisation in Sequential Programs

- The usual sequential way to initialise is to have code like this:

```
#include <stdbool.h>
void f(void)
{
        static bool initialised = false;
        if (!initialised) {
                init();
                initialised = true;
        }
        ...
}
```

- Might not work in a multithreaded program!

Initialisation in multithreadthreaded programs

```
#include <stdbool.h>
pthread_mutex_t  init_lock;
void f(void)
{
        static bool initialised = false;
        pthread_mutex_lock(&init_lock);
        if (!initialised) {
                init();
                initialised = true;
        }
        pthread_mutex_unlock(&init_lock);
}
```


Initialisation in multithreadthreaded programs

- Now we just need to initialise the mutex.... :-)
- If it can be done in main before any other threads are created, then fine.
- In the current (and future) Pthreads standard, as we have seen, we can do

```
pthread_mutex_t       init_lock = PTHREAD_MUTEX_INITIALIZER;
```

- In earlier versions of the standard, it was not guaranteed that the right hand side of above, `PTHREAD_MUTEX_INITIALIZER`, was a constant expression, and hence could not be an initialiser of data with "static storage duration" (ISO C terminology for static or global variable) before the program started.

pthread_once

- One can write `pthread_once_t once = PTHREAD_ONE_INIT;`.
- The `once` variable can have static storage duration.
- The function `int pthread_once(pthread_once_t*, void (*)(void));` is used to call a function once. It takes a pointer to a "once" variable and a function to execute for the initialisation.
- If another thread executes the same call after the first is done, nothing will happen.
- If it instead calls it during the first call, the second thread will wait until the first call is done.

Thread attributes

- Examples of attributes which can be set:
  - Whether another thread can join with a particular thread (portable).
  - Stack size (not portable)
  - Stack address (even less portable)
- A thread which is not joinable is "detached" which means the resource used by the thread are recycled immediately when the thread terminates.
- The joinable attribute can be set to one of
  - `PTHREAD_CREATE_JOINABLE`, or
  - `PTHREAD_CREATE_DETACHED`.
- An initially joinable thread can make itself detached but not vice versa.

Setting the detached and the stack size attributes

```
pthread_attr_t          attr;
pthread_t               thread;
void*                   work(void*);

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_attr_setstacksize(&attr, 2 * PTHREAD_STACK_MIN);
pthead_create(&thread, &attr, work, arg);
```


Threads in the new ISO C Standard from 2011

Contents of Lecture 6

- Atomic objects (recall, object in ISO C terminology = "data")
- Memory consistency model for non-atomic objects
- Synchronize with operation
- Dependency ordered memory accesses
- Happens before relation and data races
- Motivation from the Linux kernel for dependency order
- Implementation on POWER
- Threads API in C11 (very similar to Pthreads)

C11

- Current ISO C Standard is C11 from December 2011.
- See `http://www.open-std.org/jtc1/sc22/wg14`
- C11 contains various news but we will focus on three in this course:
  - Section 5.1.2.4 Multi-threaded executions and data races
  - `<threads.h>` — similar to Pthreads
  - `<stdatomic.h>` — types, operators, and functions for atomic objects

Atomic Objects

- Atomic objects are new in C11 and similar to Java volatile variables.
- Atomic objects can safely be accessed without locking:

```
_Atomic int counter = ATOMIC_VAR_INIT(0);

Thread 1                Thread 2
counter++;              counter++;
```

- Of course, you may need locks for other reasons to protect your data.
- Atomic objects have a global modification order and all threads see the same modification order.
- There is no total modification order of all atomic objects — different threads can see the stores to different atomic objects in different orders.

Syntax for Declaring Atomic Objects

- A new type qualifier: `_Atomic`.
- The other type qualifiers are: `const`, `volatile`, `restrict`.
- Examples:

```
_Atomic int                 a; // atomic int
int* _Atomic                b; // atomic pointer
_Atomic int*                c; // pointer to atomic int
_Atomic int* _Atomic        c; // atomic pointer to atomic int
_Atomic struct { int d; }   e; // atomic struct
```

- In C++ it's written as `atomic<int>` a.
- For easier porting C allows: `_Atomic (`*type-name*`)`

Using Atomic Objects

- Members of an atomic struct/union may not be accessed individually.
- The whole struct must first be copied to a non-atomic variable of compatible type.
- The ++, --, and compound assignment operators (e.g. +=) are atomic read-modify-write operations.
- The size of atomic and non-atomic compatible types is typically different as well as the alignment requirements.
- The memory ordering when using these operators is sequential consistency, which means costly memory fences are used.
- We will see functions for performing the same atomic operations which use relaxed memory ordering and may be preferable.
- Recall that alignment requirement refers to that for example a four byte `int` must be given an address that is a multiple of four etc.

Standard Atomic Data Types in <stdatomic.h>

- `<stdatomic.h>` defines basic atomic types including

`atomic_char`	`atomic_schar`	`atomic_uchar`
`atomic_short`	`atomic_sshort`	`atomic_ushort`
`atomic_int`	`atomic_sint`	`atomic_uint`
`atomic_long`	`atomic_slong`	`atomic_ulong`
`atomic_llong`	`atomic_sllong`	`atomic_ullong`
`atomic_wchar_t`	`atomic_intptr_t`	`atomic_uintptr_t`
`atomic_address`	`atomic_bool`	`atomic_flag`

- `atomic_flag` is a lock-free struct.
- The other types might be implemented with locks.
- An atomic flag can be initialized with `ATOMIC_FLAG_INIT`.

Lock Free Property

- To know whether the other basic atomic types are lock free, the following macros can be evaluated:

```
ATOMIC_CHAR_LOCK_FREE
ATOMIC_CHAR16_T_LOCK_FREE
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_WCHAR_T_LOCK_FREE
ATOMIC_SHORT_LOCK_FREE
ATOMIC_INT_LOCK_FREE
ATOMIC_LONG_LOCK_FREE
ATOMIC_LLONG_LOCK_FREE
ATOMIC_ADDRESS_LOCK_FREE
```

- If for example `ATOMIC_INT_LOCK_FREE` is true then both `_Atomic signed int` and `_Atomic unsigned int` are lock-free.

Initializing an Atomic Object

- The initialization itself is **not** atomic!
- Either use `ATOMIC_VAR_INIT(value)`, or
- `void atomic_init(volatile A* ptr, C value);`

Memory Order for Atomic Operations

- The enumerated type `memory_order` contains the enumerators

```
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
```

- They are used with the functions operating on atomic objects described next.
- For example:

```
x = atomic_load_explicit(&a, memory_order_relaxed);
y = atomic_load(&b);
```

- Without `_explicit`, `memory_order_seq_cst` is used.
- As we will see in detail the code becomes faster and more confusing with `memory_order_relaxed`. Crashing fast is better.

Question

- Consider the code where all variables initially are zero.

```
// Thread 1
x = atomic_load_explicit(&b, memory_order_relaxed);
atomic_store_explicit(&a, x, memory_order_relaxed);

// Thread 2
y = atomic_load_explicit(&a, memory_order_relaxed);
atomic_store_explicit(&b, 42, memory_order_relaxed);
```

- Can both `x` and `y` become 42?

Answer: Yes

- The code may execute in the following order:

```
atomic_store_explicit(&b, 42, memory_order_relaxed);          // Thread 2
x = atomic_load_explicit(&b, memory_order_relaxed);           // Thread 1
atomic_store_explicit(&a, x, memory_order_relaxed);           // Thread 1
y = atomic_load_explicit(&a, memory_order_relaxed);           // Thread 2
```

- With relaxed memory ordering and no dependency the store can be reordered and execute first.
- There is a dependency through the variable `x` between the accesses by Thread 1 — so they may not be reordered, luckily.
- We will see more about dependences below.

Atomic Exchange Macros

- These and the following atomic operations are called functions in the standard but are macros.
- *A* refers to an atomic type, e.g. `_Atomic int`.
- *C* refers to the corresponding non-atomic type, i.e. `int`.
- The atomic exchange function writes a new value and returns the old value pointed to by `ptr`.
- `C atomic_exchange_explicit(volatile A* ptr, C value, memory_order order);`
- `C atomic_exchange(volatile A* ptr, C value);`

Atomic Compare and Exchange

- There are two versions: strong and weak.
- The weak may fail (and let you know) and must therefore be used in a loop.
- The functions compare the value at the location pointed to by `ptr` with an expected value and if they are equal writes a new value.
- The result of the comparison is returned.
- If the values are not equal, the current value is copied to expected, or actually to where the pointer `expected` points.
- The strong functions behave as:

```
if (*ptr == *expected)
        *ptr = value;
else
        *expected = *ptr;
```


Atomic Compare and Exchange Function Prototypes

```
bool atomic_compare_exchange_strong_explicit(
        volatile A*     ptr,
        C*              expected,
        C               value,
        memory_order    success,
        memory_order    failure);

bool atomic_compare_exchange_weak_explicit(
        volatile A*     ptr,
        C*              expected,
        C               value,
        memory_order    success,
        memory_order    failure);

bool atomic_compare_exchange_strong(
        volatile A*     ptr,
        C*              expected,
        C               value);

bool atomic_compare_exchange_weak(
        volatile A*     ptr,
        C*              expected,
        C               value);
```

- For the explicit functions, memory is affected according to parameters `success` and `failure`, depending on the result of the comparison.

Weak Compare and Exchange Functions

- The weak compare and exchange functions may fail spuriously, which means they fail to perform the compare and exchange and "give up".
- If so, the return value is guaranteed to be false, and the current value is **not** copied to what `ptr` points.
- The weak forms allow faster implementation on machines with load-locked/load-linked/load-and-reserve and store conditional instructions — instead of atomic compare and exchange.
- The load-locked type of instructions are described below but the ide and ta is to split the atomic operation into two instructions.
- A processor *P* first performs a load-locked and then a store conditional.
- If a different processor *Q* performs a store between the load-locked and store conditional made by *P*, then the store conditional made by *P* fails.

Using the Weak Compare and Exchange Functions

```
_Atomic int     a;
int             exp;
int             value;

exp = atomic_load(&a);
do
        value = f(exp);
while (!atomic_compare_exchange_weak(&a, &exp, value));
```


Atomic Fetch and Modify Functions

- These functions atomically read-compute-modify an atomic object.

computation	C operator	function
addition	+	`atomic_fetch_add_explicit`
subtraction	-	`atomic_fetch_sub_explicit`
or	|	`atomic_fetch_or_explicit`
xor	^	`atomic_fetch_xor_explicit`
and	&	`atomic_fetch_and_explicit`

- Plus the usual non-explicit functions, for example: `C atomic_fetch_add(volatile A* ptr, M value);` adds `value` to what `A` points to.
- If `A` is arithmetic then `M` is `C` and if it's `atomic_address` then `M` is `ptr_diff_t`.
- The value of `*ptr` before the operation is returned.

Test and Set Atomic Flag

- The type `atomic_flag` is the atomic type for the classic test-and-set operation.

```
bool atomic_flag_test_and_set(
        volatile atomic_flag*   ptr);

bool atomic_flag_test_and_set_explicit(
        volatile atomic_flag*   ptr,
        memory_order            order);
```

- These functions set the flag if it was cleared.
- If it already was set, it may be written again but that would be a poor implementation due to cache effects if multiple processors are waiting for the flag to be cleared.
- The old value is returned.

Atomic Flag

- The atomic flag type and functions are the minimal hardware supported atomic operations.
- All others can be implemented using these.
- However, better performance can be achieved with more hardware support.

Clear Atomic Flag

```
bool atomic_flag_clear(
        volatile atomic_flag*   ptr);

bool atomic_flag_clear_explicit(
        volatile atomic_flag*   ptr,
        memory_order            order);
```

- These functions clear the flag.
- The `order` may neither be `memory_order_acquire` nor `memory_order_acq_rel`

The C11 Memory Consistency Model for Non-Atomic Objects

- Initially some in the standardization committee wanted to standardize on sequential consistency.
- Fortunately, C11 has standardized on a relaxed memory model for non-atomic objects.
- This is partly based on input from Linux kernel developers.
- Thus, to make an assignment visible to another thread requires synchronization between the writing and reading threads.
- There are three main kinds of synchronization and we will start with mutex unlock/lock.

Mutex Unlock/Lock

- A mutex in C11 is called `mtx_t`

```
int     a;

Thread 1                        Thread 2
mtx_lock(&m);
a = 1;
mtx_unlock(&m);

                                mtx_lock(&m);
                                printf("a = %d\n", a);
                                mtx_unlock(&m);
```

- The unlock by Thread 1 and lock by Thread 2 make the write of a visible to Thread 2.
- A part of the mutex unlock is to perform a **release operation** and of a mutex lock to perform an **acquire operation**.
- The write by Thread 1 is said to **happen before** the read by Thread 2.

Release and Acquire, and Consume

- Recall, a release makes previous writes visible to other threads.
- A release orders memory accesses so that no preceding write may be moved to after the release by the compiler or hardware.
- An acquire makes writes by other threads that have made a release visible.
- An acquire orders memory accesses so that no subsequent read or write may be moved to before the acquire.
- A consume is similar to an acquire but it lets unrelated reads be moved by the compiler to before the consume. In addition it can be implemented faster on some machines including POWER.
- Release/acquire is different from unlock/lock but unlock/lock perform release/aqcuire in addition to keeping track of the lock value.

Conflicting Expression Evaluations

- Two expression evaluations **conflict** if they access the same memory location and at least one of them modifies that location.
- For example:

```
// Thread 1          // Thread 2
a = b + c;
                     d = a + 1;
```

- An assignment is an expression so the code above conflict.
- By evaluation is meant that the expressions are computed at runtime.
- Conflicting evaluations are necessary in parallel programs unless each thread can work exclusively on its own data.
- Conflicting evaluations become a big problem if they are not ordered through the happens before relation.

Happens Before and Data Races

- We will look at the details of the happens before relation and what it means in terms of executing C code at the machine instruction level below, after introducing happens before informally.
- It is the unlock/lock pair which guarantees that the write happens before the read and that the data becomes visible to the other thread.
- Since there is no synchronization in the previous slide there is a data race, obviously.
- An unlock on m **synchronizes with** a lock on m.
- Recall that mutex unlock/lock is one of three main kinds of synchronizations which orders two variable accesses.
- The other two are:
  - Release on an atomic object M followed by acquire or consume on M
  - Memory fences.
- There are additional important details for atomic objects and fences which we will see later.

Summary so far

- An essential property of parallel C programs is that we have ordered all memory accesses through the happens before relation which will be explained in detail soon.
- If you use mutexes to order the accesses, you will be safe.
- Atomic objects and fences are intended for use when:
  - Mutexes don't give sufficient performance.
  - You implement other high level synchronization primitives.
- We will next explain the happens before relation in detail.
- Then we will show concrete examples of using atomic objects including implementation details for POWER processors.

Sequenced Before

- The most common sequence point in C/C++ is semicolon.
- Others include:
  - Function call
  - Comma operator
  - After evaluating the left operand of `?:`, `&&` and `||`.
- Below at `L`, the assignment to and use of `v` are ***not*** sequenced.

```
int u = 1, v = 2, w;

L: w = (v = u + 3) + v * 4;
```

- It is legal C but the value of `w` can become either 12 or 20.
- It is due to it is unspecified which operand of `+` is evaluated first.
- In the following w becomes 16 since comma is a sequence point.

```
w = (v = u + 3), v * 4;
```


Memory Fences

- These are also called **memory barriers** and are used frequently in the Linux kernel.
- A memory fence is one of:
  - release fence
  - acquire fence
  - both release and acquire fence
- A fence has no memory location operand.
- Two threads T_1 and T_2 synchronize using a fence if T_1 writes to an atomic object M after the release fence which is read by T_2 before the acquire fence.
- Details on the following slides.

An Example

```
_Atomic int     m;
int             u;
// T1   T2
u = 1;
atomic_thread_fence(memory_order_release);                  // A
atomic_store_explicit(&m, 1, memory_order_relaxed);         // X

        while (atomic_load_explicit(&m, memory_order_relaxed) == 0)   // Y
                ;
atomic_thread_fence(memory_order_acquire);                  // B
printf("u = %d\n", u);
```

- Accesses to atomic objects do not produce data races.
- Simply running the two threads does not order them in any way.
- Only if Thread 2 reads `m` after the write then it knows that the value of `u` is correct.
- The modification of `u` happens before the read of `u`.

Synchronize-with using Fences

- A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y which operate on an atomic object M and
- X is sequenced after A,
- Y is sequenced before B,
- Y reads a value written by X or the hypothetical release sequence headed by X if it were a release.

```
u = 1
A: release fence
X: atomic store M relaxed
```

```
Y: atomic load M relaxed
B: acquire fence
print u
```

- One of A and B can be replaced with an atomic operation, which eliminates the need for either X or Y.
- A and X can be replaced with a release operation on M.
- Y and B can be replaced with an acquire operation on M.

Wrong Synchronize-with using Fences (1)

```
Thread 1                    Thread 2
u = 1
X: atomic store M relaxed
A: release fence

                            Y: atomic load M relaxed
                            B: acquire fence
                            print u
```

- Assignment to `u` and `X` can be reordered
- Thread 2 can read `M` and perform `B`
- Thread 2 can then read `u`
- Then the invalidation of `u` can reach Thread 2

Wrong Synchronize-with using Fences (2)

```
Thread 1                    Thread 2
u = 1
A: release fence
X: atomic store M relaxed

                            B: acquire fence
                            Y: atomic load M relaxed
                            print u
```

- In Thread 2 reading `u` and `Y` can be reordered
- Thread 2 cannot know it has actually read a 1 in `M`
- `B` can have been executed before `A`

Dependences

- Dependences are intra-thread (within one thread only).
- There are two kinds of dependences:
  - Data dependence — due to writing and reading a memory location.
  - Control dependence — due to branches such as if-statements.
- ***Only*** data dependences are considered in C11.
- As we will see, this can lead to unexpected results.
- When we talk about "dependences" from now on, we only mean data dependences.

Dependences between Intra-Thread Evaluations

- Consider two expression evaluations A and B made by one thread.
- When we say that A **carries a dependency to** B it means A must be evaluated before B:

```
v = u + 1;      // A
w = v * 2;      // B
```

- The order of A and B should not be changed!
- Both compilers and hardware must preserve this order — and they do.
- A is **sequenced** before B since there is a sequence point between A and B.

Definition of Dependency in C11

- A carries a dependency to B if
  - the value of A is used as an operand of B (except the left operand of `?:` , `&&` or `||`), or
  - A writes a scalar object (pointer or arithmetic variable) or a bitfield in memory location M and B reads from M the value written by A, and A is sequenced before B
  - For some evaluation X, A carries a dependence to X and X carries a dependency to B.
- Carries a dependency is intra-thread.

Dependency-Ordered

- A **release sequence** is a maximal sequence of modifications of M headed by a release operation on M and performed by that thread or by other threads performing atomic read-modify-write accesses on M.
- Recall, a consume operation is like an acquire except it allows more optimizations on some machines, e.g. on the POWER it becomes an ordinary load instruction.
- An evaluation A in one thread is **dependency ordered before** an evaluation B in another thread if:
  - A performs a release operation on an atomic object M and B performs a consume operation on M and reads a value written by any side effect of A in the release sequence of A, or
  - for some evaluation X, A is dependency ordered before X and X carries a dependency to B.

Inter-Thread Happens Before

- An evaluation A **inter-thread happens before** an evaluation B if:
  - A synchronizes with B (mutex unlock/lock), or
  - A is dependency ordered before B (release/consume), or
  - for some evaluation X:
    - A synchronizes with X and X is sequenced before B, or
    - A is sequenced before X and X synchronizes with B, or
    - A inter-thread happens before X and X inter-thread happens before B.
- An evaluation A **happens before** an evaluation B if:
  - A is sequenced before B (intra-thread), or
  - A is inter-thread happens before B.
- C11, Section 5.1.2.4 paragraph 25:
  The execution of a program contains a **data race** *if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.*

Motivation for Dependency Ordering

- Improved performance!
- In programs (such as the Linux kernel) with important data structures which are rarely modified and very frequently read there exist faster solutions than using full release/acquire synchronization on modern architectures.
- In this sense modern architectures include POWER, MIPS and ARM.
- For other architectures including x86, optimizing compilers can make better optimizations if dependency ordering is used rather than release/acquire, as we will see below.

Read-Copy Update

- RCU is an alternative to readers-writers locks with the following essential functionality:
  - Pointers to data structures are protected with RCU.
  - Readers use read-side critical sections marked by enter/exit calls.
  - When a reader is in such a section a writer may not modify the data structure but instead modifies a copy of it.
  - When the last reader has left, the original data structure is updated.
- RCU is heavily used in the Linux kernel.
- RCU was one part of the SCO vs IBM lawsuit.
- From linux-2.6.37.1/kernel/signal.c

```
/*
 * Protect access to @t credentials. This can go away when all
 * callers hold rcu read lock.
 */
rcu_read_lock();
user = get_uid(__task_cred(t)->user);
atomic_inc(&user->sigpending);
rcu_read_unlock();
```

From the Linux Kernel Documentation

```
rcu_dereference()

typeof(p) rcu_dereference(p);

Like rcu_assign_pointer(), rcu_dereference() must be implemented
as a macro.

The reader uses rcu_dereference() to fetch an RCU-protected
pointer, which returns a value that may then be safely
dereferenced.  Note that rcu_deference() does not actually
dereference the pointer, instead, it protects the pointer for
later dereferencing.  It also executes any needed memory-barrier
instructions for a given CPU architecture.  Currently, only Alpha
needs memory barriers within rcu_dereference() -- on other CPUs,
it compiles to nothing, not even a compiler directive.
```


A Simplified Example

- Consider the following example code (also originally from Linux)

```
list_t*         head;
int             b;

void insert(int a)
{
        list_t* p = kmalloc(sizeof *p, GFP_KERNEL);

        spin_lock(&m);
        p->a = 1;
        p->next = head;
        rcu_assign_pointer(head, p);
        spin_unlock(&m);
}

void first(void)
{
        list_t* q = rcu_dereference(head);

        return q->a + b;
}
```

- The use of `rcu_dereference()` must be done within an `rcu_read_lock()` / `rcu_read_unlock()` — not seen in this example.

A More Detailed Look

```
int             b;

void first(void)
{
        list_t* q = rcu_dereference(head);

        return q->a + b;
}
```

- Before the proposal for dependency ordering through release/consume, the then current draft would require the use of an acquire operation in the `rcu_dereference`.
- Doing so prevents the compiler from loading `b` before the execution of the `rcu_dereference`.
- A standard for a high performance language must permit extensive compiler optimization and efficient execution on modern machines.
- C11 succeeds with this — also for multicores.

More Details About Dependences

- Recall the intra-thread A carries a dependency to B.
- The compiler becomes responsible for not optimizing away such dependences.
- This might sound trivial but is not. See below.
- A dependency is started with a consume operation:

```
p = atomic_load(q, memory_order_consume);
a = *p + 1;
```

- A root of a dependency tree is created with the consume, and the memory read in `*p` becomes ordered, as we expect.
- The compiler or hardware is not allowed to move the `*p` to before the first line which would probably not make much sense, anyway.
- Behaviour is as we want and expect.

Constant Propagation Example 1(3)

- Consider the following code and assume the compiler has deduced `size` is one:

```
i = atomic_load(q, memory_order_consume);
a = b[i % size];
```

- What happens if the compiler transforms the code to the following?

```
i = atomic_load(q, memory_order_consume);
a = *b;
```


Constant Propagation Example 2(3)

- There is now **no dependence** between the two statements!

```
i = atomic_load(q, memory_order_consume);
a = *b;
```

- Without a dependence the reads are not ordered.
- This is not what the programmer expected!!!
- Therefore, optimizing C compilers must analyse all dependences **before** any code transformation and preserve them.
- What does "preserve" mean, then?

Constant Propagation Example 3(3)

- Preserving the data dependency means letting the hardware know about them.
- That can be done in different ways for different processors.
- One way is to insert instructions so that there will be a chain of dependences for the processor pipeline to see.
- Compilers thus must preserve such dependences in the complete program!
- Writing optimizing compilers C all of a sudden became twice as interesting.
- There is, however, a very simple first version implementation: treat all `memory_order_consume` as `memory_order_acquire` which automatically will order the memory accesses.

One More Data Dependency

```
a = atomic_load(p, memory_order_consume);
atomic_store(q, a, memory_order_relaxed);
atomic_store(r, b, memory_order_relaxed);
```

- Since there is no data dependency between the consume and the second store, they are not ordered.
- As programmers we need to be very careful about what we expect to be ordered!

More Details Next

- Implementation on a specific architecture: POWER.
- Illustrations of what actually is ordered for several situations.
- We will begin with the POWER synchronization instructions.
- Approximate clock counts below are not fixed but depends on what is happening in the machine at the moment.
- Numbers from Christoph von Praun: *"Deconstructing Redundant Memory Synchronization"* (while at IBM Research).

mnemonic	name	POWER4 cycles	POWER5 cycles
ldarx/stwxc.	load and reserve / store conditional	80	75
hwsync or sync	heavy weight sync	140	50
lwsync	light weight sync	110	25
isync	instruction sync	30	10
eieio	enforce inorder execution of I/O	not measured	not measured

- We want to use functions which use the less costly instructions!
- Such as the consume operation which is only an ordinary load instruction on POWER!

Load and Reserve/Store Conditional Purpose

- Some processors use atomic test-and-set instructions while others use pairs of special load and store instructions.
- Load and reserve is also called load-locked and load-linked.
- In addition to POWER, it's used by ARM and MIPS.
- Test-and-set, or compare-and-swap, is used by e.g. x86.
- The purpose with load-and-reserve/store conditional is to simplify the design of the pipeline.

Load and Reserve/Store Conditional Behaviour

- The load instruction fetches data from a memory location, and makes a reservation R in memory.
- If a different processor modifies the same memory location, the reservation R is lost.
- If/when the processor with a reservation makes a conditional store, the memory location is modified only if the reservation was not lost in between.
- Therefore it's an atomic read-modify-write.
- The `stwcx.` conditional store in POWER sets a condition code to indicate whether the store succeeded or not (the dot in the mnemonic indicates that the condition code is set by an operation on POWER).

POWER Memory Barrier Instructions

- The POWER memory barrier instructions create a memory barrier with two sets of instructions:
  - the A-set with instructions a_i preceding the barrier, and
  - the B-set with instructions b_j following the barrier.
- Depending on which barrier instruction is used, some b_j instructions may be reordered with some a_i instructions.

hwsync Memory Barrier for Sequential Consistency

- The A-set consists of all instructions preceding the `hwsync`.
- The B-set consists of memory access instructions following the `hwsync`.
- Except for the instruction `icbi` which invalidates an instruction cache block, no B-set instruction may be reordered with any A-set instruction.
- This is the most costly POWER synchronzation instruction.
- Not only for cached data but for any storage.
- It's used e.g. to implement sequentially consistent write on POWER:

```
stwx    r1,r2,r3
hwsync
```


lwsync Memory Barrier for Release Operation

- The A and B sets consist of all memory access instructions preceding and following the `lwsync`, respectively.
- Only the following pairs of a_i, b_j instructions are ordered:
  - A-side load → B-side load
  - A-side load → B-side store
  - A-side store → B-side store
- Thus, A-side store → B-side load are not ordered.
- The `dcbz` data cache block zero is counted as a store.
- It's used e.g. to implement a release:

```
stwx    r1,r2,r3        # modify shared data...
stwx    r4,r5,r6
...
stwx    r29,r30,r31     # last write in critical section
lwsync
stwx    r8,r9,r10       # set lock to free
```


isync Memory Barrier for Acquire Operation

- The A and B sets consist of all instructions preceding and following the `isync`, respectively.
- An instruction which cannot raise an exception in the pipeline can be allowed to complete and instructions following the `isync` therefore actually execute without order.
- Consider the sequence:

```
ldw     r1,r2,r3
isync
stw     r4,r5,r6
```

- The store may execute before the load if the load had e.g. a cache miss.
- This is not what we want and to overcome that problem we can exploit that POWER does not permit speculative execution of store instructions.

bc;isync Memory Barrier for Acquire Operation

- By inserting a conditional branch, `bc`, before the `isync` both the `isync` and `stw` become speculative until the branch outcome is known.
- We can use beq as follows:

```
ldw     r1,r2,r3 # r1 = MEMORY[r2+r3]
cmp.    r1,r1    # certainly true but the beq must
beq              # wait according to the specification
isync            # since no speculative stw is allowed.
stw     r4,r5,r6
```

- Since the store may not execute speculatively it must wait for the branch outcome.
- This memory barrier is the fastest.
- The previous two, however, can order instructions from different processors due to the `hwsync`/`lwsync` are **cumulative** — see below.

eieio Memory Barrier

- (unique five vowel instruction)
- Enforce in order execution of I/O.
- It only orders stores.

Cumulative Ordering

- If the memory accesses ordered by a memory barrier executed by one processor P_i also take into account memory accesses executed by other processors as described below the barrier is **cumulative**.
- By **applicable** storage accesses for a barrier is meant the storage access which are ordered by that barrier.
- Two rules:
  - The A-set also includes all applicable storage accesses made by other processors which have completed with respect to P_i before the barrier is created (by executing the barrier instruction).
  - The B-set also includes all applicable accesses made by any processor P_j after P_j has executed a load that returned a value stored by an instruction in the B-set.
- The B-set expands recursively.

Example of Cumulative Ordering

```
// int a = b = 0;
// Thread 1                Thread 2                Thread 3
1:a = 1;
2:lwsync

                           3:x = a; // a == 1
                           4:y = b;
                           5:lwsync
B = { 2,3,5,6 }            6:b = 2;
                                                   7:x = b; // b == 2
                                                   8:lwsync
                                                   9:y = a; // a == 1
A = { }                    A = { 2,3,4 }           A = { 2,4,7 }
B += { 7,8 }               B = { 6,7,9 }           B = { 9 }
```

- Due to cumulativity of `lwsync`, it's certain that **if** Thread 2 reads 1 in access 3 **and** Thread 3 reads 2 in access 7 **then** Thread 3 will read 1 in access 9.
- Using instead `bc;isync` Thread 3 may read zero in access 9.

Implementation on POWER

- The following is based on the ISO C/C++ standardization document ISO/EIC JTC1 SC22 WG21 N2745 written by Paul E McKenney and Raul Silvera from IBM.
- There are other implementations possible.

Load

Load relaxed	`ld`
Load consume	`ld`
Load acquire	`ld; cmp; bc; isync`
Load seq cst	`hwsync;ld; cmp; bc; isync`

1. Using sequential consistency on machines with relaxed memory models makes it easier to write correct parallel code which will be slow.
2. Even if your code is safety critical (when people can die due to bugs) and performance is not an issue, you should not use SC, use a single threaded C program instead.

Store

Store relaxed	`st`
Store release	`lwsync;st`
Store seq cst	`hwsync;st`

1. It's easier to program under sequential consistency!
2. Yes, but, why would you want to use synchronization instructions before every store in a critical section since nobody should look at the data before you have left the critical section anyway?
3. Write buffering permitted by store relaxed allows the machine to pipeline all the stores in the critical section.
4. At the end of the critical section you use store release and wait for the remaining preceding stores (now possibly waiting in a write buffer) to complete (i.e. invalidate the other copies if there are any left).

Memory Fence

Acquire fence	`lwsync`
Release fence	`lwsync`
Acq rel fence	`lwsync`
Seq cst fence	`hwsync`

Acquire and Release a Spin Lock

```
        # acquire
loop:   lwarx   r6,0,r3         # load lock and reserve
        cmpw    r4,r6           # r4 is a free lock's value
        bne-    loop            # restart if locked.
        stwcx.  r6,0,r3         # try to store
        bne     loop            # restart if store failed.
        isync                   # store succeeded.
        lwzx    r7,r8,r9        # first load of shared data

        # release
        stwx    r7,r8,r10       # last store of shared data
        lwsync                  # export shared data
        stw     r4,0,r3         # unlock the lock. same r4 as above.
```


<threads.h> Header File

- C11 threads are similar to but simpler than Pthreads.
- `thrd_t` — thread type
- `once_flag` — a type for performing initializations exactly one time
- `mtx_t` — mutex type
- `cnd_t` — condition variable type
- `tss_t` — thread specific storage (not the same as `_Thread_local`)

Initialization Function

- `void call_once(once_flag* flag, void (*func)(void));`
- The flag should be initialized with:

```
once_flag flag = ONCE_FLAG_INIT;
```

- If multiple threads invoke `call_once` with the same flag, the function will only be called one time, and the others will wait until the call to `func` returns.

Thread Enumeration Constant Error Codes

- `thrd_success` — indicates an operation succeeded.
- `thrd_error` — indicates an operation failed but not why.
- `thrd_busy` — an operation failed due to a resource was already in use.
- `thrd_nomem` — an operation failed due to memory allocation failed.
- `thrd_timeout` — a timed wait operation timed out.

Mutex Options for `mtx_init` Function

- `mtx_plain` — the mutex should support none of below options.
- `mtx_recursive` — set mutex to support recursive locking.
- `mtx_timed` — set mutex to support timed wait.
- `mtx_try` — set mutex to support test and return.

Specifying Waiting Time

- The struct `xtime` contains at least the following members.
- `time_t sec;`
- `long nsec;`
- They may be declared in any order in the struct.
- This means that code which compares two char-pointers (each pointing to one of them) should not assume one is before the other.
- In `struct { int a, b; } c; int* p = &c.a; int* q = &c.b;` we know that `p < q`, since `a` is declared before `b`.

Condition Variable Functions

```
int cnd_init(cnd_t* cond);
void cnd_destroy(cnd_t* cond);
int cnd_signal(cnd_t* cond);
int cnd_broadcast(cnd_t* cond);
int cnd_wait(cnd_t* cond, mtx_t* mtx);
int cnd_timedwait(cnd_t* cond, mtx_t* mtx, const xtime* xt);
```

- These functions are all similar to the corresponding Pthreads functions, except that Pthread condition variables can have certain attributes and be statically initialized.

```
int pthread_cond_init(
        pthread_cond_t* restrict                cond,
        const pthread_condattr_t* restrict      attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```


Mutex Functions

```
int mtx_init(mtx_t* mtx, int type);
void mtx_destroy(mtx_t* mtx);
int mtx_lock(mtx_t* mtx);
int mtx_timedlock(mtx_t* mtx, const xtime* xt);
int mtx_trylock(mtx_t* mtx);
int mtx_unlock(mtx_t* mtx);
```

- The `type` should be one of:

```
mtx_plain
mtx_timed
mtx_try
mtx_plain | mtx_recursive
mtx_timed | mtx_recursive
mtx_try | mtx_recursive
```

- Only `mtx_trylock` can return `thrd_busy`.
- A prior call to `mtx_unlock` synchronizes with a successful call to a `mtx_lock` function.

Thread Functions

```
int thrd_create(thrd_t* thr, int (*func)(void*), void* arg);
void thrd_exit(int res);
int thrd_join(thrd_t thr, int* res);
int thrd_detach(thrd_t thr);
thrd_t thrd_current(void);
int thrd_equal(thrd_t u, thrd_t v);
void thrd_sleep(const xtime* xt);
void thrd_yield(void);
```

Thread Specific Storage Functions

```
int tss_create(tss_t* key, void (*dtor)(void*));
void tss_delete(tss_t key);
void* tss_get(tss_t key);
int tss_set(tss_t key, void* value);
```


Low-level Parallel Programming: Explicit DMA Transfers on the Cell

Contents of Lecture 7

- An Introduction to the Cell Broadband Engine Architecture

Key Ideas of the Cell Processor

- Highest possible performance at a low power dissipation
- 8 SIMD vector processors added to a multithreaded PowerPC processor
- Cost: more complex programming
- Benefit for the right program: superior performance per watt.

History of the Cell Processor

1999 A partnership with IBM is proposed by Sony CEO Nobuyuki Idei.
2001 Sony, Toshiba and IBM form the STI Design Center in Austin.
2004 First Cell processor manufactured and runs faster than 4 GHz.
2005 PS3 is shown by Sony.
2006 PS3 is sold by Sony and IBM releases a Linux SDK for the PS3.
2008 IBM's Roadrunner with 12,960 Cell processors and Opterons becomes the fastest computer in the world.
2009 There were (probably false) rumours that the Cell was killed.
2010 IBM say they might integrate Cell technology in normal Power processors.

The Cell Broadband Engine Architecture

- A Cell processor consists of:
  - One multithreaded 64-bit PowerPC processor (currently two threads)
  - Eight SIMD vector SPU processors with 128 registers and 256 KB RAM
  - One programmable memory interface controller
  - Two input/output interfaces
- The PowerPC processor is optimized for general purpose computing and SIMD vector processing
- The SPUs (synergistic processor units) are optimized for SIMD vector processing and concurrent data transfers.
- The SPUs have no caches and both data and instructions must be explicitly transferred from system memory by software using DMA.
- If programmed cleverly, the SPUs can achieve **very high** performance.

The Synergistic Processor Unit

- The SPU contains two pipelines, the even and the odd pipelines.
  - Even: fixed point unit and floating point unit.
  - Odd: fixed point unit, load/store unit, and Channel and DMA unit.
- 128 128-bit registers.
- All ISO C integer data types and float and double.
- 256 KB local RAM memory for instructions and data.
- No hardware caches!
- The Channel and DMA unit is used for synchronization with other processors and to transfer data.

Filling the SPU Instruction Pipeline

- Firstly, the instruction must already be in the local store.
- Before an instruction can start doing work in a functional unit in one of the even or odd pipelines, it must go through the following:
  1. Fetch: 7 cycles.
  2. Decode: 3 cycles.
  3. Issue: 2 cycles.
- Instructions are fetched into one of ILB1 and ILB2, which can hold 32 instructions each.
- When an ILB is empty 32 new instructions are fetched from the local store. Every instruction is 32 bits.
- Instruction fetch has the lowest priority after DMA and load/store accesses.

Branch Prediction in the SPU

- There is no dynamic branch prediction in the SPU.
- Branches to lower addresses are predicted to be taken. The idea is that a loop's branch to the next iteration is backward and is usually taken.
- A mispredicted branch costs 18 cycles.
- There are also special **branch hint** instructions which software can use to tell SPU about future branches.

SIMD Programming on the Cell

- Both the PPU and SPU support SIMD instructions.
- On the PPU, it's the VMX or AltiVec instruction set.
- On the SPU, all instructions are SIMD.
- The two SIMD instruction sets are similar but not equal.
- Use the vector extensions in GCC to avoid having to use specific SIMD instructions.

EIB and MFC

- EIB stands for the **Element Interconnect Bus**
- MFC stands for the **Memory Flow Controller**
- These two perform the data transport for DMA transfers.
- There is one EIB which consists of four rings, each 16 bytes wide.
- Each SPU has its own MFC which send and receive messages on the EIB.
- Two rings send data in one direction and the other two in the other direction.
- Software doesn't specify which ring should be used — too messy and which is best to use is dependent on what is currently happening.
- Each ring can have up to three concurrent transfers — but they are not allowed to overlap, e.g. SPU 2 can send to SPU 5 and SPU 6 to SPU 7 concurrently but SPU 6 cannot concurrently send to SPU 4 on the same ring — but on another ring is OK.

DMA Transfers

- The size of a DMA transfer must be 1,2,4,8, or a multiple of 16 and at most 16 KB.
- Better performance for 128-byte aligned data.
- The start address of objects must be 16-byte aligned — allocate extra memory with malloc.
- The lower 4 bits of the effective address and the LS address must be equal.
- If above rules are not followed, the program will get a bus error and terminate.

DMA Parameters and Operation

- A DMA request needs the following parameters:
  - effective address
  - local store address
  - size
  - tag
  - operation
- The tag identifies the DMA request and is a value in the range 0..31.
- The operation is either `MFC_PUT_CMD` or `MFC_GET_CMD`.
- This looks like a normal memory load or store access, except for the tag and larger data size.
- What is so special that the Cell processor?

Comparison with Data Prefetching

- Recall that cache misses can be difficult to avoid on multiprocessors and that data prefetching is not a universal solution.
- For instance the prefetched data might be evicted before being used — even by other prefetches!
- The difference between normal multicores (multiprocessors) and the Cell is that you have complete control over the local store parameter in the DMA request.
- You can avoid overwriting useful data simply by using different local store addresses.
- More control in other words.
- The DMA operations are similar to data prefetches in that the processor continues with other work and the MFC performs the data transfer.

The Local Store is Not a Cache, Though!

- If prefetched data does not arrive in the cache before it's needed, the processor will wait until it does arrive.
- That will not happen automatically in the SPU.
- Recall that the tag DMA parameter is used to identify DMA request.
- You can tell the SPU to wait until a subset of all pending DMA operations have completed.
- This is done by setting a bit corresponding to the tag you want to wait for and then issue a special command.
- This will cause the SPU to wait until all DMA operations of the specified tags have completed.
- In fact, you can use one tag for multiple DMA operations so you can have more than 32 pending operations if you wish.
- With this architecture you control parallelism between SPU's as well as data transfer and computation within each SPU.

Mailboxes

- An alternative to DMA for small amounts of data is using mailboxes.
- Each SPU has three mailboxes:
  - An outgoing mailbox
  - An outgoing interrupt mailbox
  - An incoming mailbox
- Directions are from the view of an SPU:
  - out means out from the SPU, and
  - in means in to the SPU.
- The outgoing mailbox has a capacity of one 32-bit word.
- The incoming mailbox has a capacity of four 32-bit words.
- Writing to a full or reading from an empty mailbox blocks the thread.

Software Development for the Cell

- There are several Linux distributions ported to the Cell and we will use YDL.
- YDL = Yellow Dog Linux. The Y is the Y in the command `yum` (Yellowdog Update Manager) available at least on YDL and on Fedora.
- IBM distributes the Cell Software Development Kit for YDL.
- The SDK contains numerous libraries, documention, tutorials, example programs.
- A Cell application is written in multiple parts:
  - One part for the PPU as a normal C program.
  - One part for each program the SPU's should execute.
  - All of these have their own `main` function.
  - The different parts are compiled with one of:
    - `spu-gcc`, and
    - `ppu32-gcc`.

Connecting the PPU and SPU Programs

- Suppose your SPU executable is called `dataflow`.
- All executables for the SPU are statically linked.
- The SPU executable is made accessible to the PPU program through:
  1. `ppu-embedspu -m32 dataflow dataflow dataflow-embed.o`
     The first `dataflow` is the symbol name and the second is the input file name.
  2. Declare in the PPU program:
     `extern spe_program_handle_t dataflow;`
     This external symbol is loaded by the PPU program and then the SPU can be executed.
- The PPU thread which starts the SPU program is blocked until the SPU program exits.
- With Pthreads, we use a thread to run the SPU and wait for it to complete.

Executing an SPU Program from a PPU Program

```
extern spe_program_handle_t     dataflow;
spe_context_ptr_t               ctx;
unsigned int                    entry = SPE_DEFAULT_ENTRY;

ctx = spe_context_create (0, NULL);
spe_program_load (ctx, &dataflow);
spe_context_run(ctx, &entry, 0, arg, NULL, NULL);
```

- Create one context for each SPU.
- Call `spe_context_run` from a thread on the PPU.
- The `void* arg` argument becomes accessible to the SPU program through an `unsigned long long parm`:

```
/* SPU main. */
int main(unsigned long long id, unsigned long long parm);
```


SPU `makefile`

```
PROGRAM_spu     := dataflow
LIBRARY_embed   := lib_dataflow_spu.a

INCLUDE         := -I .. -I /opt/cell/sdk/src/lib/sync/spu_source

LDFLAGS         += -lsync -L/opt/cell/sdk/src/lib/sync/spu
CFLAGS          += -O4

include /opt/cell/sdk/buildutils/make.footer
```


PPU `makefile`

```
C               = 3
S               = 10000
V               = 1000
U               = 4
A               = 1000

DIRS            := spu
PROGRAM_ppu     := dataflow
IMPORTS         = spu/lib_dataflow_spu.a -lspe2 -lpthread

INSTALL_DIR     = $(EXP_SDKBIN)/tutorial
INSTALL_DIR     = .
INSTALL_FILES   = $(PROGRAM_ppu)

include /opt/cell/sdk/buildutils/make.footer

#CFLAGS         = -O4 -Wall -pedantic -std=c99 -DTEST
#CFLAGS         = -O4 -Wall -pedantic -std=c99
CFLAGS          = -maltivec -O4 -Wall -pedantic -std=c99 -Werror

OBJS    = driver.o error.o

all: $(OBJS)
        ./dataflow $(S) $(V) $(U) $(A) $(C)

clean:
        rm -f $(PROGRAM_ppu) $(OBJS)
        (cd spu; make clean)
```


Hints for Cell Dataflow Analysis

- Be careful to align data properly otherwise you will see bus errors.
- Change the data structures from the C program to fit the Cell better.
- Parallelize on three levels:
  1. Compute using multiple SPU's.
  2. Transfer data in both directions while computing on an SPU.
  3. Use `__vector` GCC extension to exploit SIMD on the SPU's.
- Premature optimization is the root of all evil.

More About Tags

- Recall, every DMA request uses a tag 0..31 to identify the **tag group** of the request.
- A new tag is allocated with:

```
tag = mfc_tag_reserve();
```

- Alternatively, additional DMA requests can be grouped to a previously allocated tag.
- Nothing stops you from using only one tag for your entire program but it will not be very efficient.
- Recall, the purpose of using different tags is to have multiple data transfers in progress concurrently with your computations.
- E.g. one tag group for iteration $i + 2$, one for $i + 1$, one for i which you must wait for, and then other groups for writing back data to system RAM.

Waiting for Tag Group Completion

- `mfc_read_tag_status_immediate` returns an int with bits set to one for the completed tag groups.

```
/* loop until tag 3 has completed: */
do {
        work();
        status = mfc_read_tag_status_immediate();
} while ((status & (1<<3)) == 0);
```

- With this function, we can actually do some more work while waiting...
- The next two functions block the SPU until some or all tag groups have completed.
- `mfc_read_tag_status_any` returns when one has completed.
- `mfc_read_tag_status_all` returns when all have completed.

Releasing a Tag

- A tag is released using the function
  `uint32_t mfc_tag_release(uint32_t tag);`
- The functions for reserving and releasing a tag are not necessary to use.
- We can, however, actually use any value in 0..31.
- If different parts of our application need a tag, these library functions can help us not using the same tag for different purposes at the same time.

High-level Parallel Programming: OpenMP

Contents of Lecture 8

- OpenMP

Parallel Execution of Software

- Ideally optimizing compilers would be able to parallelize existing C/C++ and FORTRAN codes.
- From our example of the dataflow program (and other programs), I think it's rather difficult to write such a compiler.
- Instead of writing new sequential programs we can use e.g.
  - C/Pthreads, C11 Threads, Java Threads
  - Scala Actors
- What about all existing codes?

OpenMP for C/C++ and FORTRAN

- Another option is tool support for manual parallelization:
  - Programmer annotates the source code and is guarantees the validity of parallelization of a loop.
  - Tool support: generating parallel code for a loop
- GCC supports the OpenMP standard for this approach.
- Include `<omp.h>` and annotate e.g. as:

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < n; ++i) {
        /* ... */
}
```

- Compile with `gcc -fopenmp`

The Main Advantage with OpenMP

- We don't want to rewrite millions of C/C++ and FORTRAN codes from scratch.
- Using a new and relatively untested language may be a big risk.
- Untested = less than 20 years of experience...
- With OpenMP we can parallelize our applications **incrementally**.
- We can focus on one for-loop at a time.

Origin of OpenMP

- All supercomputer companies had their own compiler directives to support this "semi-automatic" parallelization.
- When SGI and Cray (one of the three Cray companies) merged they needed to define a common set of compiler directives.
- Kuck and Associates, a compiler company, and the U.S. ASCI (Accelerated Strategic Computing Initiative) joined SGI and Cray.
- In 1997 IBM, DEC, HP and others were invited to join the group now called OpenMP.
- In 1997 the specification for OpenMP 1.0 for FORTRAN was released.
- Next year the specification for C/C++ was released.
- The current version is OpenMP 4.0 and was published 2013.
- GCC 4.4 supports OpenMP.

OpenMP Components

1. Compiler directives. `#pragma` in C/C++.
2. A runtime library
3. Environment variables, like `OMP_NUM_THREADS`.

Barriers

- A barrier is a synchronization primitive which makes all threads reaching the barrier wait for each other.
- When the last thread has reached the barrier, all threads can proceed and continue after the barrier.

`#pragma omp parallel`

- A **structured block of code** is either
  - a compound statement, i.e. a block enclosed in braces, or
  - a for-loop.
- The pragma `omp parallel` is used before a structured block of code and specifies all threads should execute all of that block.
- Note that this is typically not what we want in a for-loop, see below.
- A default number of threads is used, which can be changed with the environment variable `OMP_NUM_THREADS`, which can be larger than the number of processors in the machine.
- This pragma creates an implicit barrier after the structured block.

Example

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

int main(void)
{
        #pragma omp parallel
        {
                int     tid; // thread id.

                tid = omp_get_thread_num();
                printf("hello world from thread %d\n", tid);
        }

        return 0;
}
```

- Since tid is declared in the compound statement, it becomes private.
- `omp_get_thread_num()` returns an id starting with zero.

OpenMP and Pthreads

- The OpenMP runtime library creates the threads it needs using Pthreads on Linux.
- After a parallel block, the threads wait for the next their work and are not destroyed in between.
- This model of parallelism is called **fork-join** and only the master thread executes the sequential code.
- It's possible to nest parallel regions — but when I tested no additional threads where used (instead the master thread ran the code four times).

Two OpenMP Functions

- To specify in the program how many threads you want, use

```
omp_set_num_threads(nthread);
```

- To measure elapsed wall clock time in seconds, use

```
double  start, end;

start = omp_get_wtime();
/* work. */
end = omp_get_wtime();
```


Parallel For-Loops

- In addition to the `#pragma omp parallel` you must also specify `#pragma omp for` before the loop.
- Without the second pragma each thread will execute all iterations.
- Note that it's the programmer's responsibility to check that there are no data dependences between loop iterations.

For Loop Scheduling

- There are three ways to schedule for-loops:
- `schedule(static)`
  - The iterations are assigned statically in contiguous blocks of iterations.
  - Static scheduling has the least overhead, obviously, but may suffer from poor load imbalance, e.g. in an LU-decompsition.
- `schedule(dynamic)` or `schedule(dynamic, size)`
  - The default size is one iteration.
  - A thread is assigned size contigouos iterations at a time.
- `schedule(guided)` or `schedule(guided, size)`
  - The default size is one iteration.
  - With a size, a thread never (except possibly at the end) is assigned few than size contigouos iterations at a time.
  - The number of iterations assigned to a thread is proportional to the number of unassigned iterations and the number of threads.
- In addition, `runtime` can be specified which uses the environment variable `OMP_SCHEDULE` which must be one of above three but without the size.

An Example

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

#define N (1024)

float a[N][N];
float b[N][N];
float c[N][N];

int main(void)
{
        int     i;
        int     j;
        int     k;

        #pragma omp parallel private(i,j,k)
        #pragma omp for schedule(static, N/omp_get_num_procs())
        for (i = 0; i < N; ++i)
                for (k = 0; k < N; ++k)
                        for (j = 0; j < N; ++j)
                                a[i][j] += b[i][k] * c[k][j];

        return 0;
}
```

- We need private for j and k since they are declare before the pragma.
- If a function is called in a parallel region, all its local variables become private.

Parallel Tasks

```
#pragma omp sections
{
        #pragma omp section
        {
                work_a();
        }

        #pragma omp section
        {
                work_b();
        }

}
```

- Each section is executed in parallel.

Reductions

- By a **reduction** is meant computing a scalar value from an array such as a sum.
- The loop has a data dependence on the `sum` variable.
- How can we parallelize it anyway?

```
float   a[N];
float   sum;
int     i;

for (sum = i = 0; i < N; ++i)
        sum += a[i];
```


OpenMP Reductions

- By introducing a sum variable private to each thread, and letting each thread compute a partial sum, we can parallelize the reduction:

```
float   a[N];
float   sum;
int     i;

#pragma omp parallel
#pragma omp for
#pragma omp reduction(+:sum)
for (sum = i = 0; i < N; ++i)
        sum += a[i];
```

- We can write the pragmas on one line if we wish:

```
#pragma omp parallel for reduction(+:sum)
for (sum = i = 0; i < N; ++i)
        sum += a[i];
```

- There are reductions for: `+ - * & | ^ && ||` with suitable start values such as `1` for `*` and `~0` for `&`.

Critical Sections

- A critical sections is created as in:

```
#pragma omp critical
{
        point->x += dx;
        point->y += dy;
}
```


Atomic Update

- When one variable should be updated atomically, we can use:

```
#pragma omp atomic
count += 1;
```


Explicit Barriers

- Recall there is an implicit barrier at the end of a parallel region.
- To create a barrier explicitly, we can use:

```
#pragma omp barrier
```


Work for One Thread

- Recall only the master executes the sequential code between parallel regions.
- If we wish only the master should execute some code in a parallel region, we can use

```
#pragma omp master
```

- If it doesn't matter which thread performs the work, we can instead use

```
#pragma omp single
```

- There is a difference between the two above constructs: an implicit barrier is created after a `single` directive.

Locks

- OpenMP supports two kinds of locks: plain locks and recursive locks.
- Recall a thread can lock a recursive lock it already owns without blocking for ever.
- Recursive locks are called nested locks in OpenMP.
- The lock functions are `omp_init_lock`, `omp_set_lock`, `omp_unset_lock`, `omp_test_lock` and `omp_destroy_lock`, and `omp_nest_init_lock`, `omp_nest_set_lock`, `omp_nest_unset_lock`, `omp_nest_test_lock` and `omp_nest_destroy_lock`

OpenMP Memory Consistency Model

- Weak ordering is the consistency model for OpenMP.
- The required synchronization instructions are inserted implicitly with the above introduded directives.
- A for loop can be created without an implicit barrier using `nowait` and in that case `#pragma omp flush` makes caches consistent.
- A list of variables to write back can be specified: `#pragma omp flush(a,b,c)`

Compiler Support for OpenMP

- The following compilers support OpenMP
- GNU, IBM XL, Oracle, Intel, Portland Group, Pathscale, Absoft, Fujitsu, Microsoft, HP, Cray.

Parallelizing Sequential C Code Automatically

Contents of Lecture 9

- Reflections on explicit parallelism and OpenMP.
- Application classes needing automatic parallelization
- Loop-level parallelization using unimodular transformations
- The SUIF compiler
- Thread level speculative execution — a study of SPEC CPU2000 at Stanford

The Need for Automatic Parallelization

- There are huge amounts of source code which is sequential.
- We almost always want faster programs.
- Which classes of sequential software can be automatically parallelized?
- Why should we not always parallelize by hand?
- More precise question: how can we get sufficient performance...
  - ...at acceptable development cost and time?
  - ...for acceptable hardware costs?
- For the last 50 years or so, there has been a quest for automatically parallelizing sequential software.
- Should we write multi-threaded code or hope for automatic parallelization???

No Simple Answer

- Different projects have radically different needs!
- Different source codes are more or less easy to parallelize!
- There is no simple answer to the question whether automatic parallelization is worthwhile.
- If it solves your problem under various constraints, then it is worthwhile, as any other optimization.

Some Reflections on OpenMP

- Of the codes we have seen so far, for which can we realistically build tools which automatically can extract lots of parallelism?
- Automatically parallelizing sequential Java or C codes with complex recursive data structures (e.g. a graph)? Not likely!?
- Automatically rewriting sequential Scala programs and create actors? Not likely!?
- OpenMP loops? Yes, since such loops don't have dependences, at least there exist numerical loops which can be executed in parallel!!
- Since such loops exist — why do we need OpenMP??
- Because no compiler exists which is as reliable as a programmer inserting `#pragma parallel for` directives.
- However, with a sufficiently good parallelizing C compiler for numerical codes (i.e. matrix computations) we would not need OpenMP.
- In this lecture we will learn how close we are to such C compilers.

Safety of Parallelization

- The criterion to determine whether it's safe for a compiler to generate parallel code for a loop is, of course, whether the parallel code may produce different output.
- If different loop iterations access the same array element and at least one of the accesses is a store, then the loop must be executed sequentially.

From Simple to Hard Parallelization Problems

- As we will see, matrix computation codes where the loop bounds and array references are linear functions of the loop index variables are relatively easy to analyse.
- This analysis will then conclude either that the loop is sequential, or a certain number of outer loops can run in parallel.
- At the other end are codes which for example contain dynamically allocated recursive data structures, e.g. lists or trees — it's very difficult or impossible to determine for a compiler wether it is safe to execute parts of such codes in parallel.
- In between are codes which sometimes can be analysed using symbolic analysis, i.e. mathematical reasoning with symbols (as opposed to only known constants) which can prove that a loop can execute in parallel.

Runtime Testing

- For some codes, even if it's impossible to determine at compile time whether it's safe to parallelize a loop, that information might be available at runtime.
- Therefore it's common to perform testing at runtime to determine whether a loop should execute sequentially or in parallel.
- In addition to determining the safety of parallel execution, this technique can also be used to determine whether parallel execution will make the loop faster.

Speculative Execution

- Instead of giving up on parallelization of difficult sequential codes, some researchers argue for **speculative execution** at the thread level.
- All superscalar processors speculate on the outcome of branches which is very useful.
- Thread-level speculation might be a good idea but the costs of miss-speculation may be too high.

Back to ILLIAC IV

- Recall ILLIAC IV built from 1965-1975 at the University of Illinois
- As mentioned earlier that project developed many code transformations for vectorization and parallelization.
- The theoretical foundation was made by Utpal Banerjee in his master's thesis.
- He is at Intel and has written a series of very readable books on data dependence analysis and parallelization.

Inner vs Outer Loop Parallelization

- In the course optimizing compilers in LP1 (every even year) we learn about inner loop parallelization which is used e.g. for automatic SIMD vectorization and software pipelining.
- Here the focus instead is on automatic parallelization for multicores, i.e. outer loop parallelization.
- The foundations for inner and outer loop parallelization are similar, since they both rely on data dependence analysis.

Introduction to Data Dependence Analysis

- There are **data dependences** in the following code:

```
S1: x = a + b;
S2: y = x + 1;
S3: x = b * c;
```

- The value written to x in S_1 is read in S_2.
- This is called a **true dependence** and is written $S_1 \delta^t S_2$.
- This notation means that S_1 must execute before S_2 in a transformed program.
- In a true data dependence between two statements both statements access the same memory location, and the first statements writes a value which the other statements reads.

Data Dependences at Different Levels

- Data dependences can be at several different levels, including:
  - Instructions
  - Statements
  - Loop iterations
  - Functions
  - Threads
- Parallelizing compilers usually find parallelism between different loop iterations of a loop.
- If the compiler can determine that there are no dependences between loop iterations then it can either:
  - Produce parallel machine code, or
  - Produce source code with OpenMP `#pragma parallel for` directives.
- If there are dependences, it may still be possible to execute the loop in parallel since perhaps the loop iterations are not totally ordered.

Total vs Partial Order and Loop Iteration Iterations

- Integers are totally ordered since we can determine which of a and b is greater if $a \neq b$.
- Consider a directed acyclic graph. In topological sorting you can process any vertex u if all predecessors of u already have been processed.
- Obviously, we should not execute a loop iteration before its input data has been computed.
- In executing a loop in parallel we perform a topological sort of the loop iterations.
- There is no search at runtime to determine which iterations can be executed, though.
- The topological sorting is the major work in parallelization.
- That is, to find an order of the iterations which maximizes parallelism.

Different Types of Data Dependences

- When we write "loop iteration" below our sentence is also valid for the other levels (instruction, statement, etc).
- Below iteration I_1 always executes before iteration I_2.
- Recall, in a **true dependence**, written $I_1 \delta^t I_2$, I_1 produces a value consumed by I_2.
- In an **anti dependence**, written $I_1 \delta^a I_2$, I_1 reads a memory location later overwritten by I_2.
- In an **output dependence**, written $I_1 \delta^o I_2$, I_1 writes a memory location later overwritten by I_2.
- In an **input dependence**, written $I_1 \delta^i I_2$, both I_1 and I_2 read the same memory location.
- The first three types of dependences create partial orderings among all iterations, which parallelizing compilers exploit by ordering iterations to improve performance.

Dependences in the Example Code

- Let us classify all dependences in the code:

```
S1: x = a + b;
S2: y = x + 1;
S3: x = b * c;
```

- There is a true dependence from S1 to S2 due to x.
- There is an anti dependence from S2 to S3 due to x.
- There is an output dependence from S1 to S3 due to x.

Loop Level Data Dependences

- In the loop

```
for (i = 3; i < 100; i += 1)
        a[i] = a[i-3] + x;
```

- There is a true dependence from iteration i to iteration $i + 3$.
- Iteration $i = 3$ writes to a_3 which is read in iteration $i = 6$.
- A loop level true dependence means one iteration writes to a memory location which a later reads.
- Typically this means one iteration I writes to an array element a_x which a **later** iteration reads.

Perfect Loop Nests

- A **perfect loop nest L** is a nest of m nested **for** loops $L_1, L_2, ...L_m$ such that the body of L_i, $i < m$, consists of L_{i+1} and the body of L_m consists of a sequence of assignment statements.
- For $1 < r \leq m$ p_r and q_r are linear functions of $I_1, ..., I_{r-1}$.

for $(I_1 = p_1;\ I_1 <= q_1;\ I_1 += 1)$ {
  for $(I_2 = p_2;\ I_2 <= q_2;\ I_2 += 1)$ {
    ⋮
    for $(I_m = p_m;\ I_m <= q_m;\ I_m += 1)$ {
      $h(I_1, I_2, ..., I_m)$;
    }
  }
}

Example Perfect Loop Nest

- All assignments, **except** to the loop index variables are in the innermost loop.
- There may be any number of assignment statements in the innermost loop.

```
for (i = 0; i < 100; i += 1) {
        for (j = 3 + i; j <  2 * i + 10; j += 1) {
                for (k = i - j; k < j - i; k += 1) {
                        a[i][j][k] += b[k][j][i];
                }
        }
}
```


Loop Bounds

- The lower bound for I_1 is $p_{10} \leq I_1$.
- The lower bound for I_2 is

$$\begin{array}{rcl} I_2 & \geq & p_{20} + p_{21} I_1 \\ p_{20} & \leq & I_2 - p_{21} I_1 \\ p_{20} & \leq & -p_{21} I_1 + I_2 \end{array} \quad (1)$$

- The lower bound for I_3 is

$$\begin{array}{rcl} I_3 & \geq & p_{30} + p_{31} I_1 + p_{32} I_2 \\ p_{30} & \leq & I_3 - p_{31} I_1 - p_{32} I_2 \\ p_{30} & \leq & -p_{31} I_1 - p_{32} I_2 + I_3 \end{array} \quad (2)$$

and so forth. We represent this on matrix form as $\mathbf{p_0} \leq \mathbf{IP}$, or... see next slide.

Loop Bounds on Matrix Form

- $\mathbf{P} = \begin{pmatrix} 1 & -p_{21} & -p_{31} & \dots & -p_{m1} \\ 0 & 1 & -p_{32} & \dots & -p_{m2} \\ 0 & 0 & 1 & \dots & -p_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$ and $\mathbf{p_0} = (p_{10}, p_{20}, ..., p_{m0})$.
- Similarly, the upper bounds are represented as $\mathbf{IQ} \leq \mathbf{q_0}$.
- The loop bounds, thus, are represented by the system:

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{IP} \\ & & \mathbf{IQ} \leq \mathbf{q_0} \end{array}\right\}$$

Example Non-Perfect Loop Nest

- The assignment to c_{ij} before the innermost loop makes it a non-perfect loop nest.
- Sometimes non-perfect loop nest can be split up, or **distributed** into perfect loop nests.
- See next slide.

```
for (i = 0; i < 100; i += 1) {
        for (j = 0; j < 100; j += 1) {
                c[i][j] = 0;
                for (k = 0; k < 100; k += 1) {
                        c[i][j] += a[i][k] * b[k][j];
                }
        }
}
```


Loop Distribution

- Result of loop distribution.

```
for (i = 0; i < 100; i += 1)
        for (j = 0; j < 100; j += 1)
                c[i][j] = 0;
for (i = 0; i < 100; i += 1)
        for (j = 0; j < 100; j += 1)
                for (k = 0; k < 100; k += 1)
                        c[i][j] += a[i][k] * b[k][j];
```


Some Terminology

- The index vector $\mathbf{I} = (I_1, I_2, ..., I_m)$ is the vector of index variables.
- The index values of **L** are the values of $(I_1, I_2, ..., I_m)$.
- The index space of **L** is the subspace of Z^m consisting of all the index values.
- An **affine array reference** is an array reference in which all subscripts are linear functions of the loop index variables.

Symbolic Analysis

- Data dependence analysis is normally restricted to affine array references.
- In practice, however, subscripts often contain **symbolic constants** as shown below which is test s171 in the C version of the Argonne Test Suite for Vectorising Compilers.
- There is no dependence between the iterations in this test.

```
for (i=0; i<n; i++)
    a[i*n] = a[i*n] + b[i];
```


Problematic Non-Affine Index Functions

- In the loop

```
scanf("%d", &x);

for (i = 3; i < 100; i += 1) {
S1:     a[i]   = a[x] + 1;
S2:     b[i]   = b[c[i-1]] + 2;
S3:     d[i]   = d[i * i] + 3;
}
```

- Some compilers try to do runtime dependence testing to take care of S_1 but it may cause too much overhead if many variables must be checked.
- While S_3 is not difficult, almost all parallelizing compilers focus on index expressions which are linear functions of the loop variables.

Representing Array References

- Let X be an n-dimensional array. Then an affine reference has the form:
- $X[a_{11}i_1 + a_{21}i_2...a_{m1}i_m + a_{01}]...[a_{1n}i_1 + a_{2n}i_2...a_{mn}i_m + a_{0n}]$
- This is conveniently represented as a matrix and a vector $X[\mathbf{IA} + \mathbf{a_0}]$, where
- $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$ and $\mathbf{a_0} = (a_{10}, a_{20}, ..., a_{n0})$.
- We will refer to **A** and $\mathbf{a_0}$ as the **coefficient matrix** and the **constant term**, respectively.

The Data Dependence Equation

- For two references $X[\mathbf{IA} + \mathbf{a_0}]$ and $X[\mathbf{IB} + \mathbf{b_0}]$ to refer to the same array element there must be two index values, **i** and **j** such that $\mathbf{iA} + \mathbf{a_0} = \mathbf{jB} + \mathbf{b_0}$ which we can write as $\mathbf{iA} - \mathbf{jB} = \mathbf{b_0} - \mathbf{a_0}$.
- This system of Diophantine equations has n (the dimension of the array X) scalar equations and $2m$ variables, where m is the nesting depth of the loop.
- It can also be written in the following form:

$$(\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b_0} - \mathbf{a_0}. \quad (3)$$

- We solve the system of linear Diophantine equations in (3) using a method presented shortly.

An Example

```
for (i = 0; i < 100; i += 1)
        for (j = 2*i + 4; j < i + 40; j += 1)
                a[2i-3j-1][2i+j-3] = f(a[-3i+4j+1][-i+2j+7]);
```

- The above loop nest has the following two array reference representations:
  $\mathbf{A} = \begin{pmatrix} 2 & 2 \\ -3 & 1 \end{pmatrix}$ and $\mathbf{a_0} = (-1, -3)$.
  $\mathbf{B} = \begin{pmatrix} -3 & -1 \\ 4 & 2 \end{pmatrix}$ and $\mathbf{b_0} = (1, 7)$.

Dependence Distances

- Let $\prec_\ell$ be a relation in $\mathbf{Z}^m$ such that $\mathbf{i} \prec \mathbf{j}$ if $i_1 = j_1$, $i_2 = j_2$, ..., $i_{l-1} = j_{l-1}$, and $i_l < j_l$.
- For example: $(1, 3, 4) \prec_3 (1, 3, 9)$.
- The lexicographic order $\prec$ in $\mathbf{Z}^m$ is the union of all the relations $\prec_\ell$: $\mathbf{i} \prec \mathbf{j}$ iff $\mathbf{i} \prec_\ell \mathbf{j}$ for some ℓ in $1 \leq \ell \leq m$.
- The sequential execution of the iterations of a loop nest follows the lexicographic order.
- Assume that $(\mathbf{i}; \mathbf{j})$ is a solution to (3), and that $\mathbf{i} \prec \mathbf{j}$. Then $\mathbf{d} = \mathbf{j} - \mathbf{i}$ is the **dependence distance** of the dependence.

Uniform Dependence Distance

- If a dependence distance **d** is a constant vector then the dependence is said to be uniform.
- The dependence distance $\mathbf{d} = (1, 2)$ is uniform, while the dependence distance $\mathbf{d} = (1, t_2)$ is nonuniform.
- Uniform distance vectors are *very* desirable since loops with only uniform distance vectors can be optimized with unimodular transformations, described below.
- For this, the set of all distance vectors $\mathbf{d_i}$ of a loop nest **L** are arranged in a matrix with n rows and m columns where n is the number of dependences in **L** and m is the number of index variables in **L**.
- Note that a zero distance between references within the same statement does not cause an loop-level dependence, e.g. `a[i] = a[i] + x` but still an instruction-level dependence.

Loop Independent and Loop Carried Dependences

- A loop independent dependence is a dependence such that $\mathbf{d} = \mathbf{j} - \mathbf{i} = (0, ..., 0)$.
- A loop independent dependence does not prevent concurrent execution of different iterations of a loop. Rather, it constrains the scheduling of instructions in the loop body.
- A loop carried dependence is a dependence which is not loop independent, or, in other words, the dependence is between two different iterations of a loop nest.
- A dependence has level ℓ if in $\mathbf{d} = \mathbf{j} - \mathbf{i}$, $\mathbf{d}_1 = 0, \mathbf{d}_2 = 0, ..., \mathbf{d}_{l-1} = 0$, and $\mathbf{d}_l > 0$.
- Only a loop carried dependence has a level, and it is only the loop at that level which needs to be executed sequentially.

The GCD Test

- The GCD test was invented at Texas Instruments and first described 1973.
- Consider the loop

```
for (i = lb; i <= ub; ++i)
        x[ a1 * i + c1] = x[a2 * i + c2] + y;
```

- To prove independence, we must show that the Diophantine equation

$$a_1 i_1 - a_2 i_2 = c_2 - c_1 \quad (4)$$

has no solutions.
- We compute the gcd of a_1 and a_2 and check whether it divides $c_2 - c_1$, and if it does not, there is no solution and we have proved independence, otherwise we must use another test.

Weaknesses of The GCD Test

- There are two weaknesses of the GCD test:
  1. It does not exploit knowledge about the loop bounds.
  2. Most often the gcd is one.
- The first weakness means the GCD Test might be unable to prove independence despite the solution to (4) actually lies outside the index space of the loop.
- The second weakness means independence usually cannot be proved.

GCD Test for Nested Loops and Multdimensional Arrays

- The GCD Test can be extended to cover nested loops and multidimensional arrays.
- The solution is then a vector and it usually contains unknowns.
- The Fourier-Motzkin Test described shortly takes the solution vector from this GCD Test and checks whether the solution lies within the loop bounds.
- Next we will look at unimodular matrices and Fourier-Motzkin elimination used by the Fourier-Motzkin Test.

Unimodular Matrices

- An integer square matrix $\mathbf{A}$ is unimodular if its determinant $det(\mathbf{A}) = \pm 1$.
- If $\mathbf{A}$ and $\mathbf{B}$ are unimodular, then $\mathbf{A}^{-1}$ exists and is itself unimodular, and $\mathbf{A} \times \mathbf{B}$ is unimodular.
- $\mathcal{I}$ is the $m \times m$ identity matrix.

Elementary Row Operations

- The operations
  - *reversal*: multiply a row by -1,
  - *interchange*: interchange two rows, and
  - *skewing*: add an integer multiple of one row to another row,

  are called the elementary row operations.
- With each elementary row operation, there is a corresponding *elementary matrix*.

Performing Elementary Row Operations

- To perform an elementary row operation on a matrix $\mathbf{A}$, we can premultiply it with the corresponding elementary matrix.
- Assume we wish to interchange rows 1 and 3 in a 3×3 matrix $\mathbf{A}$. The resulting matrix is formed by

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \times \mathbf{A}.$$

- The elementary matrices are all unimodular.

3×3 Reversal Matrices

- $$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

3×3 Interchange Matrices

- $$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

3×3 Upper Skewing Matrices

- $$\begin{pmatrix} 1 & z & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & z \\ 0 & 0 & 1 \end{pmatrix}.$$

3×3 Lower Skewing Matrices

- $$\begin{pmatrix} 1 & 0 & 0 \\ z & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ z & 0 & 1 \end{pmatrix},$$

  and
- $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & z & 1 \end{pmatrix}.$$

Echelon Matrices

- Let l_i denote the column number of the first nonzero element of matrix row i.
- A given $m \times n$ matrix $\mathbf{A}$, is an *echelon matrix* if the following are satisfied for some integer ρ in $0 \leq \rho \leq m$:
  - rows 1 through ρ are nonzero rows,
  - rows $\rho + 1$ through m are zero rows,
  - for $1 \leq i \leq \rho$, each element in column l_i below row i is zero, and
  - $l_1 < l_2 < \ldots < l_\rho$.
- The following are examples of echelon matrices:

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{pmatrix}$$

Echelon Reduction

- Given an $m \times n$ matrix $\mathbf{A}$, Echelon reduction finds two matrices $\mathbf{U}$ and $\mathbf{S}$ such that $\mathbf{U} \times \mathbf{A} = \mathbf{S}$, where $\mathbf{U}$ is unimodular and $\mathbf{S}$ is echelon.
- $\mathbf{U}$ remains unimodular since we only apply elementary row operations.

```
function echelon_reduce(A)
    U ← I_m
    S ← A
    i_0 ← 0
    for (j ← 1; j ≤ n; j ← j + 1) {
        if (there is a nonzero s_ij with i_0 < i ≤ m) {
            i_0 ← i_0 + 1
            i ← m
            while (i ≥ i_0 + 1) {
                while (s_ij ≠ 0) {
                    σ ← sign(s_(i−1)j × s_ij)
                    z ← ⌊|s_(i−1)j| / |s_ij|⌋
                    subtract σz(row i) from (row i − 1) in (U; S)
                    interchange rows i and i − 1 in (U; S)
                }
                i ← i − 1
            }
        }
    }
    return U and S
end
```


Example Echelon Reduction

- We will now show how one can echelon reduce the following matrix:

$$\mathbf{A} = \begin{pmatrix} 2 & 2 \\ -3 & 1 \\ 3 & 1 \\ -4 & -2 \end{pmatrix}.$$

- We start with with $\mathbf{U} = \mathbf{I_4}$ and $\mathbf{S} = \mathbf{A}$ which we write as:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 0 & 3 & 1 \\ 0 & 0 & 0 & 1 & -4 & -2 \end{array}\right).$$

- Then we will eliminate the nonzero elements in $\mathbf{S}$ starting with $s_{41}, s_{31}, s_{21}, s_{42}$ and so on.

Example Echelon Reduction

- $\mathbf{j = 1, i_0 = 1, i = 4}$. We always wish to eliminate s_{ij}, which currently means s_{41}.
- $\sigma \leftarrow -1$ and $z \leftarrow 0$. Nothing is subtracted from row 3.
- Then rows 3 and 4 are interchanged in $(\mathbf{U}; \mathbf{S})$, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 0 & 1 & -4 & -2 \\ 0 & 0 & 1 & 0 & 3 & 1 \end{array}\right).$$

Example Echelon Reduction

- We continue the inner while loop and find that $\sigma \leftarrow -1$ and $z \leftarrow 1$. Then $-1\times$ row 4 is subtracted from row 3, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 1 & 0 & 3 & 1 \end{array}\right).$$

- Then rows 3 and 4 are interchanged, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 0 & 3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right).$$

Example Echelon Reduction

- s_{41} is still zero, and the inner while loop is continued and $\sigma \leftarrow -1$ and $z \leftarrow 3$. Then $-3\times$ row 4 is subtracted from row 3:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 0 & 0 & 1 & 1 & -1 & -1 \end{array}\right).$$

- Then rows 3 and 4 are interchanged, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & 0 & 0 & -3 & 1 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Now the first $_{ij}$ has become zero and i is decremented.

Example Echelon Reduction

- $\mathbf{j = 1, i_0 = 1, i = 3}$. We now wish to eliminate s_{31}. $\sigma \leftarrow +1$ and $z \leftarrow 3$. Then $3\times$ row 3 is subtracted from row 2:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Then rows 2 and 3 are interchanged, resulting in:

$$(\mathbf{U}; \mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

Example Echelon Reduction

- $\mathbf{j} = \mathbf{1}, \mathbf{i_0} = \mathbf{1}, \mathbf{i} = \mathbf{2}$. We now wish to eliminate s_{21}. $\sigma \leftarrow -1$ and $z \leftarrow 2$. Then $-2\times$ row 2 is subtracted from row 1:

$$(\mathbf{U};\mathbf{S}) = \left(\begin{array}{cccc|cc} 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Interchanging rows 2 and 1 results in:

$$(\mathbf{U};\mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & -3 & -3 & 0 & 4 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

Example Echelon Reduction

- $\mathbf{j} = \mathbf{2}, \mathbf{i_0} = \mathbf{2}, \mathbf{i} = \mathbf{4}$. We now wish to eliminate s_{42}. $\sigma \leftarrow -1$ and $z \leftarrow 2$. $-2\times$ row 4 is subtracted from row 3:

$$(\mathbf{U};\mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & -2 \end{array}\right).$$

- Interchanging rows 4 and 3 results in:

$$(\mathbf{U};\mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

Example Echelon Reduction

- $\mathbf{j} = \mathbf{2}, \mathbf{i_0} = \mathbf{2}, \mathbf{i} = \mathbf{3}$. We now wish to eliminate s_{32}. $\sigma \leftarrow 0$ and $z \leftarrow 0$. Nothing is subtracted from row 2 but rows 3 and 2 are interchanged:

$$(\mathbf{U};\mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

At this point $\mathbf{S}$ is an echelon matrix and the algorithm stops (the outer while loop since $i = i_0$). As will turn out to be convenient later, we prefer positive values of s_{11} and therefore multiply with -1 finally resulting in:

$$(\mathbf{U};\mathbf{S}) = \left(\begin{array}{cccc|cc} 0 & 0 & -1 & -1 & 1 & 1 \\ 0 & 0 & 4 & 3 & 0 & -2 \\ 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 5 & 3 & 0 & 0 \end{array}\right).$$

GCD of multiple integers

- Let $a_1, a_2, ..., a_m$ denote a list of integers, not all zero,
- $\mathbf{U}$ an $m \times m$ unimodular matrix,
- $\mathbf{S} = (s_{11}, 0, ...0)^T$ an $m \times 1$ echelon matrix, such that $\mathbf{UA} = \mathbf{S}$ where $\mathbf{A}$ is the $m \times 1$ matrix $(a_1, a_2, ..., a_m)^T$,
- then $\gcd(a_1, a_2, ..., a_m) = |s_{11}|$.

Linear Diophantine Equations

- To perform data dependence analysis for multidimensional arrays we need to consider a system of n linear diophantine equations in m variables.
- m is twice the loop nesting and n the number of dimensions in an array.

$$\mathbf{xA} = \mathbf{c} \tag{5}$$

Here $\mathbf{x}$ is an $1 \times m$ integer matrix, $\mathbf{A}$ is an $m \times n$ integer matrix, and $\mathbf{c}$ is an $1 \times n$ integer matrix.

- (5) is easy to solve if $\mathbf{A}$ is an echelon matrix.
- With echelon reduction we find $\mathbf{U}$ and $\mathbf{S}$ such that $\mathbf{UA} = \mathbf{S}$.
- We will check if there is an integer solution to $\mathbf{tS} = \mathbf{c}$ instead.

Linear Diophantine Equations

Theorem

- *Let $\mathbf{A}$ be a given $m \times n$ integer matrix and $\mathbf{c}$ a given integer n vector.*
- *Let $\mathbf{U}$ denote an $m \times m$ integer matrix an $\mathbf{S}$ an $m \times n$ integer echelon matrix, such that $\mathbf{UA} = \mathbf{S}$.*

The system of equations

$$\mathbf{xA} = \mathbf{c} \tag{6}$$

has a solution iff there exists an integer m-vector $\mathbf{t}$ such that $\mathbf{tS} = \mathbf{c}$. When a solution exists, the set of all solutions is given by the formula

$$\mathbf{x} = \mathbf{tU} \tag{7}$$

where $\mathbf{t}$ is the integer vector which satisfies $\mathbf{tS} = \mathbf{c}$.

Linear Diophantine Equations

Proof.

- An integer m-vector $\mathbf{x} = \mathbf{tU}$ will be a solution to 7 iff

$$\mathbf{c} = \mathbf{xA} = \mathbf{tUA} = \mathbf{tS} \tag{8}$$

- If there is no such integer vector $\mathbf{t}$ such that $\mathbf{tS} = \mathbf{c}$, then there is no integer solution to $\mathbf{xA} = \mathbf{c}$ either.
- If there is such a $\mathbf{t}$, then all solutions have the form $\mathbf{x} = \mathbf{tU}$, where $\mathbf{t}$ is integral and $\mathbf{tS} = \mathbf{c}$.

□

Linear Diophantine Equations

- To illustrate how equations of the form $\mathbf{xA} = \mathbf{c}$ can be solved using the techniques introduced above, let us solve

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \end{pmatrix} \begin{pmatrix} 2 & 2 \\ -3 & 1 \\ 3 & 1 \\ -4 & -2 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \tag{9}$$

- Firstly we use echelon reduction to find the matrices $\mathbf{U}$ and $\mathbf{S}$.
- Then we formulate the equation $\mathbf{tS} = \mathbf{c}$:

$$\begin{pmatrix} t_1 & t_2 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & -2 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 4 \end{pmatrix} \tag{10}$$

It is trivially solved and we find that $\mathbf{t} = (2, -1, t_3, t_4)$, where t_3 and t_4 are arbitrary integers.

Linear Diophantine Equations

- We then find $\mathbf{x}$:

$$\mathbf{x} = \mathbf{tU} = \begin{pmatrix} 2 & -1 & t_3 & t_4 \end{pmatrix} \begin{pmatrix} 0 & 0 & -1 & -1 \\ 0 & 0 & 4 & 3 \\ 1 & 0 & 2 & 2 \\ 0 & 1 & 5 & 3 \end{pmatrix} = \tag{11}$$

$$(t_3, t_4, 2t_3 + 5t_4 - 7, 2t_3 + 3t_4 - 5) \tag{12}$$

Fourier-Motzkin Elimination

- Suppose we find, during data dependence analysis, an integer vector $\mathbf{x}$ which is a solution to $\mathbf{xA} = \mathbf{c}$.
- Then what we can conclude is that there exist index variables such that the two array references being tested can reference the same memory location.
- If no solution can be found then we know there is no dependence. If there is a solution, then there may be a dependence.
- If the solution $\mathbf{x}$ represents index variables which are outside the loop bounds, then $\mathbf{x}$ does not prove that a data dependence exists. So, we need also solve a system of linear inequalities when the solution $\mathbf{x}$ exists.
- An additional constraint is, of course, that the solution is integer. Unfortunately, the problem of solving a linear integer inequality is NP-complete.

Fourier-Motzkin Elimination

- In 1827 Fourier published a method for solving linear inequalities in the real case. This method is known as Fourier-Motzkin elimination and is used in compilers as an approximation.
- If Fourier-Motkin elimination finds that there is no real solution, then there certainly is no integer either. But if there is a real solution, there may or may not be an integer solution.
- Fourier-Motzkin elimination is regarded as a time-consuming algorithm and to apply it so perhaps thousands of data dependence tests may make the compiler too slow. Therefore, it is used as a backup tests when other faster tests fail to prove independence.

Fourier-Motzkin Elimination

- An interesting question is how frequently Fourier-Motzkin elimination finds a real solution when there is no integer solution. Some special cases can be exploited.
- For instance, if a variable x_i must satisfy $2.2 \leq x_i \leq 2.8$ then there is no integer solution.
  Otherwise, if we find eg that $2.2 \leq x_i \leq 4.8$ then we may try the two cases of setting $x_i = 3$ and $x_i = 4$, and see if there still is a real solution.
- It is easiest to understand Fourier-Motzkin elimination if we first look at an example.

Fourier-Motzkin Elimination

- Assume we wish to solve the following system of linear inequalities.

$$\begin{array}{rcrcl} 2x_1 & - & 11x_2 & \leq & 3 \\ -3x_1 & + & 2x_2 & \leq & -5 \\ x_1 & + & 3x_2 & \leq & 4 \\ -2x_1 & & & \leq & -3 \end{array} \tag{13}$$

- We will first eliminate x_2 from the system, and then check whether the remaining inequalities can be satisfied. To eliminate x_2, we start out with sorting the rows with respect to the coefficients of x_2:

$$\begin{array}{rcrcl} -3x_1 & + & 2x_2 & \leq & -5 \\ x_1 & + & 3x_2 & \leq & 4 \\ 2x_1 & - & 11x_2 & \leq & 3 \\ -2x_1 & & & \leq & -3 \end{array} \tag{14}$$

Fourier-Motzkin Elimination

- First we want to have rows with positive coefficients of x_2, then negative, and lastly zero coefficients.
- Next we divide each row by its coefficient (if it is nonzero) of x_2:

$$\begin{array}{rcrcl} \frac{-3}{2}x_1 & + & x_2 & \leq & \frac{-5}{2} \\ \frac{1}{3}x_1 & + & x_2 & \leq & \frac{4}{3} \\ \frac{2}{11}x_1 & - & x_2 & \geq & \frac{3}{11} \end{array} \tag{15}$$

Of course, the $\leq$ becomes $\geq$ when dividing with a negative coefficient. We can now rearrange the system to isolate x_2:

$$\begin{array}{rcl} x_2 & \leq & \frac{3}{2}x_1 - \frac{5}{2} \\ x_2 & \leq & -\frac{1}{3}x_1 + \frac{4}{3} \\ \frac{2}{11}x_1 - \frac{3}{11} & \leq & x_2 \end{array} \tag{16}$$

Fourier-Motzkin Elimination

- At this point, we make a record of the minimum and maximum values that x_2 can have, expressed as functions of x_1. We have:

$$b_2(x_1) \leq x_2 \leq B_2(x_1) \tag{17}$$

where

$$\begin{array}{rcl} b_2(x_1) & = & \frac{2}{11}x_1 \\ B_2(x_1) & = & \min(\frac{3}{2}x_1 - \frac{5}{2}, -\frac{1}{3}x_1 + \frac{4}{3}) \end{array} \tag{18}$$

Fourier-Motzkin Elimination

- To eliminate x_2 from the system, we simply combine the inequalities which had positive coefficients of x_2 with those which had negative coefficients (ie, one with positive coefficient is combined with one with negative coefficient):

$$\begin{array}{rcrcl} \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & \frac{3}{2}x_1 - \frac{5}{2} \\ \frac{2}{11}x_1 & - & \frac{3}{11} & \leq & -\frac{1}{3}x_1 + \frac{4}{3} \end{array} \tag{19}$$

- These are simplified and the inequality with the zero coefficient of x_2 is brought back:

$$\begin{array}{rcl} -\frac{29}{22}x_1 & \leq & -\frac{49}{22} \\ -\frac{17}{33}x_1 & \leq & \frac{53}{33} \\ -2x_1 & \leq & -3 \end{array} \tag{20}$$

Fourier-Motzkin Elimination

- We can now repeat parts of the procedure above:

$$\begin{array}{rcl} x_1 & \leq & \frac{53}{17} \\ x_1 & \geq & \frac{49}{29} \\ x_1 & \geq & \frac{3}{2} \end{array} \tag{21}$$

- We find that

$$\begin{array}{rcl} b_1() & = & \max(49/29, 3/2) = 49/29 \\ B_1() & = & 53/17 \end{array} \tag{22}$$

The solution to the system is $\frac{49}{29} \leq x_1 \leq \frac{53}{17}$ and $b_2(x_1) \leq B_2(x_1)$ for each value of x_1.

Fourier-Motzkin Elimination

```
procedure fourier_motzkin_elimination(x, A, c)
    r ← m,    s ← n,    T ← A,    q ← c
    while (1) {
        n1 ← number of inqualities with positive t_ij
        n2 ← n1 + number of inqualities with negative t_ij
        Sort the inequalities so that the n1 with t_ij > 0 come first,
            then the n2 − n1 with t_ij < 0 come next,
            and the ones with t_ij = 0 come last.
        for (i = 1; i ≤ r − 1; i ← i + 1)
            for (j = 1; i ≤ n2; j ← j + 1)
                t_ij ← t_ij/t_ij
        for (j = 1; i ≤ n2; j ← j + 1)
            q_j ← q_j/t_ij
        if (n2 > n1)
            b_r(x_1, x_2, ..., x_{r−1}) = max_{n1+1≤j≤n2}(− Σ_{i=1}^{r−1} t_ij x_i + q_i)
        else
            b_r ← −∞
        if (n1 > 0)
            j_r(x_1, x_2, ..., x_{r−1}) = min_{n1+1≤j≤n2}(− Σ_{i=1}^{r−1} t_ij x_i + q_i)
        else
            B_r ← ∞
        if (r = 1)
            return make_solution()
```


Fourier-Motzkin Elimination

```
        /* We will now eliminate x_r. */
        s' ← s − n2 + n1(n2 − n1)
        if (s' ≥ 0) {
            /* We have not discovered any inconsistency and */
            /* we have no more inequalities to check. */
            /* The system has a solution. */
            The solution set consists of all real vectors (x_1, x_2, ..., x_m),
            where x_{r−1}, x_{r−2}, ..., x_1 are chosen arbitrarily, and
            x_m, x_{m−1}, ..., x_r must satisfy
            b_i(x_1, x_2, ..., x_{i−1}) ≤ x_i ≤ B_i(x_1, x_2, ..., x_{i−1}) for r ≤ i ≤ m.
            return solution set.
        }
        /* There are now s' inequalities in r − 1 variables. */
        The new system of inequalities is made of two parts:
        Σ_{i=1}^{r−1}(t_ik − t_ij)x_i ≤ q_k − q_j for 1 ≤ k ≤ n1, n1 + 1 ≤ j ≤ n2
        Σ_{i=1}^{r−1} t_ij x_i ≤ q_j for n2 + 1 ≤ j ≤ s
        and becomes by setting r ← r − 1 and s ← s':
        Σ_{i=1}^{r} t_ij x_i ≤ q_j for 1 ≤ j ≤ s
    } end

function make_solution()
    /* We have come to the last variable x_1. */
    if (b_1 > B_1  or (there is a q_j < 0 for n2 + 1 ≤ j ≤ s))
        return there is no solution
    The solution set consists of all real vectors (x_1, x_2, ..., x_m),
        such that b_i(x_1, x_2, ..., x_m) ≤ x_i ≤ B_i(x_1, x_2, ..., x_m) for 1 ≤ i ≤ m.
    return solution set.
end
```


Summary

- In the case of a loop nest of height m and an n-dimensional array, we use the matrix representation of the references $\mathbf{iA} + \mathbf{a_0} = \mathbf{jB} + \mathbf{b_0}$, or equivalently:

$$(\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b_0} - \mathbf{a_0}, \tag{23}$$

where the $\mathbf{A}$ and $\mathbf{B}$ have m rows and n columns.
- We find a $2m \times 2m$ unimodular matrix $\mathbf{U}$ and a $2m \times n$ echelon matrix $\mathbf{S}$ such that

$$\mathbf{U} \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{S}. \tag{24}$$

- If there is a $2m$ vector $\mathbf{t}$ which satisfies $\mathbf{tS} = \mathbf{b_0} - \mathbf{a_0}$ then the GCD test cannot exclude dependence, and if so...
- ..., the computed $\mathbf{t}$ will be input to the Fourier-Motzkin Test.

The Fourier-Motzkin Test

- If the GCD Test found a solution vector $\mathbf{t}$ to $\mathbf{tS} = \mathbf{c}$, these solutions will be tested to see if they are within the loop bounds.
- Recall we wrote

$$\mathbf{x} = (\mathbf{i}; \mathbf{j}) \begin{pmatrix} \mathbf{A} \\ -\mathbf{B} \end{pmatrix} = \mathbf{b_0} - \mathbf{a_0}. \tag{25}$$

- We find $\mathbf{x}$ from:

$$\mathbf{x} = (\mathbf{i}; \mathbf{j}) = \mathbf{tU} \tag{26}$$

- With $\mathbf{U_1}$ being the left half of $\mathbf{U}$ and $\mathbf{U_2}$ the right half we have:

$$\mathbf{i} = \mathbf{tU_1} \tag{27}$$
$$\mathbf{j} = \mathbf{tU_2} \tag{28}$$

- These should be inserted to loop bounds constraints.

The Fourier Motzkin Test

- Recall the original loop bounds are:

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{IP} \\ & & \mathbf{IQ} \leq \mathbf{q_0} \end{array}\right\}$$

- The solution vector $\mathbf{t}$ must satisfy:

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{tU_1P} \\ & & \mathbf{tU_1Q} \leq \mathbf{q_0} \\ \mathbf{p_0} & \leq & \mathbf{tU_2P} \\ & & \mathbf{tU_2Q} \leq \mathbf{q_0} \end{array}\right\} \tag{29}$$

- If there is no integer solution to this system, there is no dependence.
- Recall, however, the system is solved with real or rational numbers so the Fourier-Motzkin Test may fail to exclude independence.

After Data Dependence Analysis

- When we have performed data dependence analysis of all pairs of references to the same arrays, we have a **dependence matrix**, denoted $\mathbf{D}$.
- Some rows will be due to some array and other rows due to some other arrays.
- It's the dependence matrix that determines which transformations we can do.
- As mentioned, in the optimizing compilers course inner loop transformations are studied for SIMD vectorization and software pipelining.
- We will look at outer loop parallelization.

Unimodular Transformations

- A **unimodular transformation** is a loop transformation completely expressed as a unimodular matrix $\mathbf{U}$.
- A loop nest $\mathbf{L}$ is changed to a new loop nest $\mathbf{L_U}$ with loop index variables:

$$\mathbf{K} = \mathbf{IU}$$
$$\mathbf{I} = \mathbf{KU^{-1}}$$

- The same iterations are executed but in a different order.
- A new iteration order might make parallel execution possible.
- Before generating code for the new loop, the loop bounds for $\mathbf{K}$ must be computed from the original bounds:

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{IP} \\ & & \mathbf{IQ} \leq \mathbf{q_0} \end{array}\right\}$$

Computing the New Index Variables

- With

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{IP} \\ & & \mathbf{IQ} \leq \mathbf{q_0} \end{array}\right\} \tag{30}$$
$$\mathbf{I} = \mathbf{KU^{-1}} \tag{31}$$

We use Fourier-Motzkin elimination to find the loop bounds from

$$\left.\begin{array}{rcl} \mathbf{p_0} & \leq & \mathbf{KU^{-1}P} \\ & & \mathbf{KU^{-1}Q} \leq \mathbf{q_0} \end{array}\right\} \tag{32}$$

- The bounds are found starting with k_1, k_2 etc.
- This is the reason why we want to have an invertible transformation matrix.

New Array References

- All array references are rewritten to use the new index variables.
- Conceptually we could calculate, at the beginning of each loop iteration,

$$\mathbf{I} = \mathbf{KU^{-1}}$$

and then use this vector $\mathbf{I}$ in the original references, on the form:

$$x[\mathbf{IA} + \mathbf{a_0}]$$

- We don't do that of course and instead replace each reference with

$$x[\mathbf{KU^{-1}A} + \mathbf{a_0}]$$

- Here $\mathbf{KU^{-1}A} + \mathbf{a_0}$ can be calculated at compile-time.

The Distance Matrix

- The set of all vectors of dependence distances is represented by the **distance matrix** $\mathbf{D}$.
- We are free to swap the rows of $\mathbf{D}$ since it really is a set of dependences.
- Unimodular transformations require that all dependences are uniform, i.e. with known constants.
- Consider a uniform dependence vector $\mathbf{d} = \mathbf{j} - \mathbf{i}$.
- With the $\mathbf{K}$ index variables we have $\mathbf{d_U} = \mathbf{jU} - \mathbf{iU} = \mathbf{dU}$.
- Therefore, given a dependence matrix $\mathbf{D}$ and a unimodular transformation $\mathbf{U}$, the dependences in the new loop $\mathbf{L_U}$ become:

$$\mathbf{D_U} = \mathbf{DU}$$

Valid Distance Matrices

- The sign, **lexicographically**, of a vector is the sign of the first nonzero element.
- A distance vector can never be lexicographically negative since it would mean that some iteration would depend on a future iteration.
- Therefore no row in the new distance matrix $\mathbf{D_U} = \mathbf{DU}$ may be lexicographically negative.
- If we would discover a lexicographically negative row in $\mathbf{D_U}$, that loop transformation is invalid, such as the second row of the following $\mathbf{D_U}$:

$$D_U = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}$$

Outer Loop Parallelization

- By **outer loops** is meant all loops starting with the outermost loop.
- While we always can find a unimodular matrix through which we can parallelize the inner loops, this is not the case for outer loops.
- To parallelize the inner loops, we need to assure that all loop carried dependences are carried at the outermost loop.
- In other words, the leftmost column of the distance matrix $\mathbf{D_U}$ simply should consist only of positive numbers!
- For outer loop parallelization, $\mathbf{D_U}$ instead should have leading zero columns.

Rank of a Matrix

- A column of a matrix is linearly independent if it cannot be expressed as a linear combination of the other columns.
- The rank of a matrix is the number of linearly independent columns.
- For instance, an identity matrix $\mathbf{I_m}$ with m columns has $\text{rank}(\mathbf{I_m}) = m$.
- Any unimodular $m \times m$-matrix $\mathbf{U}$ has $\text{rank}(\mathbf{U}) = m$.
- A matrix with zero columns must have a rank less than the number of columns.
- So, since $\mathbf{D_U} = \mathbf{DU}$, if $\mathbf{D_U}$ should have a rank less than m, it must be $\mathbf{D}$ which contributes with that.

Outer Loop Parallelization Example

- Assume we have the distance matrix $\mathbf{D}$ defined as:

$$\mathbf{D} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix}$$

- With this distance matrix, only the innermost loop can be executed in parallel.
- We want a $\mathbf{D_U}$ with positive rows and zero columns to the left.
- For example:

$$\mathbf{D_U} = \begin{pmatrix} 0 & ? & ? \\ 0 & ? & ? \\ 0 & ? & ? \end{pmatrix} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \mathbf{U}$$

- If $\mathit{rank}(\mathbf{D}) = 3$ then such a $\mathbf{U}$ cannot exist.

Steps towards Finding $\mathbf{U}$

- We start with transposing $\mathbf{D}$:

$$\mathbf{D^t} = \begin{pmatrix} 6 & 0 & 1 \\ 4 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix}$$

- Using the Echelon reduction algorithm, we compute:
  - a unimodular matrix $\mathbf{V}$
  - an echelon matrix $\mathbf{S}$
- Such that $\mathbf{VD^t} = \mathbf{S}$, e.g.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -2 \\ 1 & -1 & -1 \end{pmatrix} \mathbf{D^t} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & 0 \end{pmatrix}$$

More Steps towards Finding $\mathbf{U}$

- We have $\mathbf{VD^t} = \mathbf{S}$:
$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -2 \\ 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} 6 & 0 & 1 \\ 4 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & 0 \end{pmatrix}$$
- Assume we wish to find $n = 1$ parallel outer loops.
- Then we find an $m \times (n+1)$ matrix $\mathbf{A}$ such that $\mathbf{DA}$ has n zero columns and then a column with elements greater than zero.
- This $\mathbf{A}$ will be used to find $\mathbf{U}$.
- How can we find $\mathbf{A}$?
- Multiplying the last row of $\mathbf{V}$ with the columns of $\mathbf{D^t}$ produces the zero row in $\mathbf{S}$.
- Thus, the first column of $\mathbf{A}$ should be the last row of $\mathbf{V}$, i.e.
$$\mathbf{DA} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & ? \\ -1 & ? \\ -1 & ? \end{pmatrix} = \begin{pmatrix} 0 & ? \\ 0 & ? \\ 0 & ? \end{pmatrix}$$

Finding the Rest of $\mathbf{A}$

- Finding the last column of $\mathbf{A}$ is easy. Denote it $\mathbf{u}$.
$$\mathbf{DA} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & u_1 \\ -1 & u_2 \\ -1 & u_3 \end{pmatrix} = \begin{pmatrix} 0 & \geq 1 \\ 0 & \geq 1 \\ 0 & \geq 1 \end{pmatrix}$$
- Multiplying each row of $\mathbf{D}$ with $\mathbf{u}$ should produce a positive number:
$$\begin{aligned} 6u_1 + 4u_2 + 2u_3 &\geq 1 \\ u_2 - u_3 &\geq 1 \\ u_1 + u_3 &\geq 1 \end{aligned}$$
- We find $\mathbf{u}$ to be e.g. $\mathbf{u} = (1, 1, 0)$.
$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Computing $\mathbf{U}$

- Given a matrix $\mathbf{A}$, using a variant of the algorithm for echelon reduction, we can find a unimodular matrix $\mathbf{U}$ such that $\mathbf{A} = \mathbf{UT}$
- i.e.
$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} = \mathbf{UT} = \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Computing $\mathbf{L_U}$

- With this loop transformation matrix $\mathbf{U}$, we get the following new dependence matrix $\mathbf{D_U}$:
$\mathbf{D_U} = \mathbf{DU}$
- i.e.
$$\mathbf{D_U} = \begin{pmatrix} 0 & 10 & 6 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \mathbf{DU} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$
- We don't actually need to compute $\mathbf{D_U}$ if we trust our compiler.
- The new loop $\mathbf{L_U}$ is constructed as explained before:
- A loop nest $\mathbf{L}$ is changed to a new loop nest $\mathbf{L_U}$ with loop index variables:
$$\mathbf{K} = \mathbf{IU}$$
- New array references and new loop bounds must be computed.
- We have already seen both of these two, but repeat them for convenience on the next two slides.

Recall: Computing the New Index Variables

- With

$$\left.\begin{aligned} \mathbf{p_0} &\leq \mathbf{IP} \\ \mathbf{IQ} &\leq \mathbf{q_0} \end{aligned}\right\} \quad (33)$$

$$\mathbf{I} = \mathbf{KU^{-1}} \quad (34)$$

We use Fourier-Motzkin elimination to find the loop bounds from

$$\left.\begin{aligned} \mathbf{p_0} &\leq \mathbf{KU^{-1}P} \\ \mathbf{KU^{-1}Q} &\leq \mathbf{q_0} \end{aligned}\right\} \quad (35)$$

- The bounds are found starting with k_1, k_2 etc.

Recall: New Array References

- All array references are rewritten to use the new index variables.
- Conceptually we could calculate, at the beginning of each loop iteration,
$$\mathbf{I} = \mathbf{KU^{-1}}$$
and then use this vector $\mathbf{I}$ in the original references, on the form:
$$x[\mathbf{IA} + \mathbf{a_0}]$$
- We don't do that of course and instead replace each reference with
$$x[\mathbf{KU^{-1}A} + \mathbf{a_0}]$$
- Here $\mathbf{KU^{-1}A} + \mathbf{a_0}$ can be calculated at compile-time.

The SUIF Compiler

- The SUIF compiler was developed at Stanford University.
- The project was lead by Monica Lam — who also invented modulo scheduling for software pipelining.
- Her project demonstrated that good parallelizing compilers not only must identify parallel loops, but at the same time perform cache optimizations to reduce communication between processors.

Parallelizing Compiler Challenges

- Due to synchronization and communication overhead, large computations must be identified that can execute in parallel.
- However, only identifying (or creating through transformations) parallelism is not sufficient.
- Early parallelizing compilers were not very good at optimizing both for:
  - parallelism, and
  - locality

  at the same time.
- This limited their success significantly.
- Due to shared memory and multiple levels of caches, multiprocessors make this complex.

Automatic Parallelization: Possible Portable Speedup

- As the SUIF compiler project showed, both parallelism and locality must be optimized.
- Without such compilers which can manage both, programmers must do it manually, which is expensive and time consuming.
- Furthermore, optimizing manually usually means tuning for a particular machine with its cache parameters.
- While performance can be good for a specific machine, there is a high risk the performance is not portable.
- The tuning needs to be repeated for other machines.
- That is another reason for using the best available parallelizing compiler.

Finding Coarse-Grain Parallelism

- Scalar analyses
- Array analyses
- Interprocedural analysis framework

Scalar Analyses

- Privatization of scalar variables
  - each thread gets its own variable
  - in addition to loop index variables, typically also "temporary" variables used in loops which are defined and used in only one loop iteration
- Scalar reduction recognition (such as `sum += a[i]`)
- Constant propagation.
- Induction variable recognition and elimination.
- Motion of loop-invariant expressions out of loops.

Array Analyses

- Performs data dependence tests as explained earlier.
- The Fourier-Motzkin test is the most expensive and is performed last if the others fail to prove independence.
- SUIF can also privatize arrays.
- Recognition of reductions on array elements, i.e. parallel reductions.
- Some such reductions are even recognized on the form `a[b[i]]`.
- The latter is useful for sparse computations.

Interprocedural Analysis Framework

- Instead of the simpler solution of inlining is interprocedural dataflow analysis performed.
- Inlining does not scale to large programs and should not be used instead of proper interprocedural dataflow analysis.
- The dataflow information is expressed as a (mathematical) function of the (source code) function's arguments.
- When the calling contexts are sufficiently different, the framework selectively clones the called procedure, i.e. makes a copy which is tuned to the specific calling context.
- This approach gathers as much specific information as does inlining but consumes much less memory — both in the compiler and compiled executable.
- In the FORTRAN 77 application TURB3D from SPEC CPU95, loops with up to nine functions in 42 calls, and (if inlined) 86,000 lines were parallelized.
- The complete TURB3D benchmark is slightly more than 2000 lines.

Memory Optimization Issues

- Communication — true sharing misses
- Limited capacity — numerical applications often access huge amounts of data before reusing the same data which results in poor temporal locality.
- Limited associativity — if the data is mapped to the same cache locations there can be conflict misses
- Large cache block size — risks of false sharing misses
- The SUIF compiler tries to avoid these issues as explained shortly, and it attempts to hide the latency of the remaining misses through data prefetching.

Transformations

- The SUIF compiler analyses which parts of a large array each processor will access.
- For example, a 2D array where a processor accesses every nth row can be transformed into a 3D array where all accessed rows are contiguous in memory.
- Arrays can be transposed when that increases locality.
- By making a processor's data contiguous in memory, both true and false sharing misses are reduced.
- Loop tiling is also used for the resulting loops.

Performance

- The SUIF compiler set a world record for SPEC CPUfp95 using a machine with eight Alpha processors.
- Two of the benchmarks resulted in speedups of approximately 10 and 15 times. How can that happen?
- Of 18 applications, interprocedural analysis was essential for seven, and locality optimizations for three.

Cache Aware Programming on Multicores

Contents of Lecture 10

- Cache Misses
- Reduce Communication
- Impove Locality
- Data Prefetching

Cache Misses

- Cold miss: first time a variable is accessed.
- Capacity miss: miss due to cache size regardless of associativity.
- Conflict miss: miss due to limited associativity.
- True sharing miss: essential miss since it communicates data
- False sharing miss: non-essential miss. We could have ignored the invalidation

Reduce Communication

- Ideally, each processor should work on its own data and no other should be involved. No communication.
- With communication, try to balance amount of tasks and communication.
- Small tasks gives better load balancing but usually more communication.

Exploiting a single consumer

- Suppose only one thread should be informed about an event. Instead of having one spin lock that all threads wait on, let each waiting thread have a unique spin lock so only one needs to know about the event.
- The "producer" needs to know which "consumer" to wake up of course.
- This technique can be used e.g. with spin locks in large machines.
- Instead of one spin lock we use an array of spin-locks and at an unlock the next element in the array is unlocked.
- Through this only the next thread in line will know about it.

False Sharing Miss

- Assume a cache block size of two words.

Access	*Processor 1*	*Processor 2*	*Comment*
1	`Load 0`		Cold miss
2		`Load 1`	Cold miss
3		`Store 1`	Invalidation
4	`Load 0`		False sharing miss

Effects of larger cache block size:

- Increased benefit from spatial locality (prefetching within block)
- The larger risk of suffering from false sharing.

True Sharing Miss

Access	*Processor 1*	*Processor 2*	*Comment*
1	`Load 0`		Cold miss
2		`Load 1`	Cold miss
3		`Store 1`	Invalidation
4	`Load 0`		True sharing miss
5	`Load 1`		Reads a new value

- While we *cannot* know it at the time of Access 4, that miss is a true sharing miss (which we realize at Access 5).

Reducing False Sharing

- Make sure each thread uses its own cache block.
- Bad idea to put multiple spin locks in an array.
- Put pointers to spin locks in the array instead.
- Cache block size usually not fixed for an architecture.
- Some Power processors use 32 bytes while e.g. 970MP uses 128 bytes.

Improve Locality

- Loop reordering and blocking as in the uniprocessor case
- Small data structures.
- Be aware of how they are accessed.
- With large structs, put the fields used at the same time near each other to make them be fetched in the same cache miss.

Data Prefetching

- The purpose is to fetch data so that it is available in the cache when it's needed.
- Compilers and hardware can do this for matrix codes.
- This is very difficult on recursive data structures such as lists or trees.
- Suppose we have a loop which traverses a list or tree.
- To prefetch a node needed e.g. three iterations ahead, we need to dereference multiple pointers where each dereference can result in a cache miss.
- In a superscalar processor with out-of-order execution of load instructions (i.e. a relaxed memory consistency model), this can possibly be useful.
- In a processor with a blocking cache, the pipeline will halt at the first cache miss and make the prefetching almost useless.

An Approach to Prefetching Nodes

- The problem with lists and trees is that we usually do not know the address of a node needed in the future.
- This is true if we allocate memory with standard methods such as `malloc` while an arena is more predictable.
- However, assume the size of a data structure is fixed for some time.
- Then we can put pointers to the nodes in an array in the expected order of traversal, and then we may be able to prefetch nodes sufficiently in advance.
- This can be useful if we will traverse a data structure multiple times.
- An example: the control flow graph in the course project, or the dominator tree of a control flow graph in an optimizing compiler.

More difficulties

- For shared data we intend to modify, it can be useful to prefetch it in exclusive mode, meaning that we request ownership of the cache block.
- The effect of this is:
  - Reduced write penalty in a sequentially consistent machine.
  - Reduced write traffic in all machines.
- However, with the ownership requests, there is a risk that we introduce additional cache misses!
- Measurements are needed, but note they are dependent both on the
  - Input data
  - Machine parameters such as number of processors, cache sizes, and latency.

Hardware-based Data Prefetching

- Several processors, including 970MP, do prefetching of array references.
- They work by discovering a constant stride and then predict which blocks will be required.
- Power also supports software programmable prefetch engines.

Cache-Miss Initiated Software Controlled Prefetch Engines

- A paper by Michel Dubois and myself evaluated the following:
  - Treat L2 cache misses as leight weight exceptions — there soon will not be much to do for the processor to do anyway.
  - Such exceptions do not involve the OS kernel but simply jump to a special place in the program.
  - For certain references in certain loops, the compiler has created an exception handler which will program a prefetch engine.
- The exception handler is a part of the function's control flow graph so it has access to all local variables which are register allocated both for the function and the exception handler.
- Therefore the exception handler can compute what to prefetch while the L2 cache miss is being serviced.
- The instruction overhead of always prefetching is removed.
- Knowing whether to insert prefetch instructions or not can be impossible, e.g. for `memcpy`.

Software Controlled Stream Prefetch on Power

- Four data streams can be prefetched concurrently
- The basic instruction is `dst` — data stream touch
- One of the instruction fields is a two bit stream selector
- Other parameters:
  - Prefetch unit size S in 16-byte blocks: value is 0 .. 31 where 0 means 32.
  - N Number of units to prefetch
  - Distance D in bytes between two units (i.e. stride)
- Use as follows:
  - `#include <altivec.h> vec_dst(addr`

Research trends

Contents of Lecture 11

- Thread level speculation based on programmer annotations in Stanford Hydra (Kunle Olukotun)
- Newer SPARC processors (Kunle Olukotun)
- Transactional Memory

Motivation for Chip Multiprocessors

- Extracting more instruction-level parallelism from sequential code running on superscalar processors is becoming increasingly difficult.
- The logic required to issue n instructions per clock grows quadratically.
- Increasing clock frequency of superscalar processors is also becoming increasingly difficult.
- Not only gate delay is significant but also wire delay, which means that a large complicated processor can be limited by long wires.
- Smaller independent processors can reduce this problem.

Thread Level Parallelism

- With a parallel program and multiple processors per chip, the transistors can be used much more efficiently. **But you need a parallel program...**
- A chip multiprocessor might have four processors, four L1 write-through data caches and one shared L2 cache.
- Communication between different caches on the same chip is of course much easier than on different nodes in a multiprocessor; so data layout is less of a problem.
- Just one problem: current parallelizing compilers are not good enough.
- Normally only numerical codes can be automatically parallelized.

An Attempt at Hardware Parallelised Code

- Since there are so many important sequential programs which need performance improvements, new approaches are possibly required.
- One proposed solution is to let the hardware guess cleverly where an independent thread might exist in the stream of instructions, run it speculatively and terminate it when an error is detected.
- The shared cache is the ideal "police" here: it can see which thread uses which data and in which order.
- There is one non-speculative thread and a number of speculative threads running ahead.
- If a speculative thread reads something which the original thread writes later, then a violation error is detected.
- Similar violations can happen for write/read and write/write.

Compare Speculative Threads with a Multithreaded Program

- In a multithreaded program (or compiler-parallelized code) the software must have explicit synchronization primitives.
- In a hardware-based thread speculation system, the hardware runs new threads and if there is no data dependence violation, everything was fine.
- So one obvious question is which is fastest?
- It depends, of course, on the amount of dependence violations.
- Either can be fastest.
- Programmer directives can tell the hardware where speculation is most promising — typically derived from a profiler.

Finding Threads

- Two issues: memory dependence violations and the same for registers.
- In Stanford Hydra, the hardware only considers memory dependences.
- This means every thread must access memory for a variable at least once so a violation can be detected.
- Two reasonable places for starting a thread: loop iteration and function call.
- In a call, the original thread does the call and a speculative thread continues after the call in the calling function.
- Hydra schedules in hardware the four least speculative threads (including the non-speculative thread).
- Eventually, a speculative thread either is killed or becomes the oldest.

Requirements on the Memory System in Hydra

- Forward data between parallel threads.
- Detect when a read occurs too early.
- Safely discard speculative state after violations.
- Retire speculative threads in the correct order.
- Provide memory renaming.

We will next see what each means.

Forward Data Between Parallel Threads

- A write by a thread invalidates the copies in future thread's caches (so that they can safely fetch the data from the shared cache if needed).
- A write does not invalidate earlier threads' caches but a certain bit is set there anyway to deal with the case when a completely new thread will run on the earlier thread's processor. That new thread should see the invalidation.
- The shared cache contains the permanent state and each thread has a write buffer at the shared cache, where speculative writes are stored.
- When a cache block is fetched from the shared cache, data in earlier threads' write buffers are merged with the data in the shared cache.

Detect when Reads Occur Too Early

- Recall that when a write occurs, future threads' cache copies are invalidated.
- We want to detect that a future thread has read something in the wrong order.
- By noting in future thread's cache which words have been read, then when an invalidation comes, it can be checked whether the future thread has violated a data dependence. If so, the future thread is restarted.

Safely Discard Speculative State After Violations

- The permanent state is in the shared cache.
- A future thread is not allowed to update the shared cache before it becomes non-speculative.
- Any thread writes to its L1 data cache.
- Therefore, while the Hydra uses write-through L1 caches, future threads write to a buffer at the shared cache.

Retire Speculative Writes in the Correct Order

- Recall all speculative writes are done to buffers.
- Simply transferring data from the buffers to the shared cache in the correct order, solves the problem.

The Pre-Invalidate Bit

- A processor can only read data written by itself or earlier threads.
- A write from a future thread should not invalidate a cache line in a L1 data cache.
- But, when the current thread running on a processor terminates and an even newer thread is run there instead, then that new thread should see the invalidations.
- Therefore, the pre-invalidate bit mentioned before is needed to hold information about an invalidation from a future thread and delay the invalidation.

Performance of Stanford Hydra

- Some programs see speedups of 2-3 times.
- With feedback information, the programmer could fix some problems and many programs saw speedups above 1.5.
- They use special syntax e.g. `pwhile` to make a parallel while loop when the programmer should control the parallelization — and not the hardware.
- Such marked loops will cause the compiler to make certain variables private to each thread.

TM: Transactional Memory

- This is sometimes called transactional coherence and consistency, TCC.
- The idea is essentially the same as speculative threads.
- All code is executed in transactions. A transaction is a unit of work which
  - modifies shared data atomically, and
  - either succeeds or is restarted.
- TM uses hardware which speculatively runs transactions in parallel, and restarts them if they fail.
- When a transaction completes, the hardware commits all its writes to shared memory as an atomic unit.

Benefits of using Transactional Memory

- The programmer "only" has to divide a program into transactions.
- No need for programmer orchestration using locks etc.
- Performance tuning is based on feedback and results in changing the transactions — which will affect only performance and not correctness.

TM Hardware

- Software defines the transactions (e.g. one loop iteration).
- Hardware buffers all writes locally.
- When a transaction has completed, hardware arbitrates for doing commit.
- At commit, all local writes are transferred to the shared memory (cache).
- Other processors snoop (listen) to the commit and can detect that its transaction has violated a dependency and needs to be restarted.
- There is also software TM which has more overheads.

Performance Programming for TM

- Minimize violations: don't write transactions that access shared variables too much for too long.
- On the other hand, making transactions too small introduces overhead.
- Avoid buffer overflows. If a processor's own buffer overflows, the transactions must "swap" the local writes to memory (of course not making the data visible to others — just in order to get more storage).

Making Transactions

- Divide software into transactions
  - The basic work is the same as normal parallelization: finding parallel regions.
  - But, the programmer does not have to guarantee that no data dependences exist.
  - If there are violations, then hardware will fix it by restarting the transaction.
- Specify order. Default order is program order, but the programmer can specify a more relaxed order which will improve performance.
- Performance tuning. Rewrite code based on feedback on which transactions had to be restarted; this is possibly more useful than cache miss ratios.

Privatization in TM

- All shared data must be accessed only in transactions.
- It's a bug to let both non-transactions and transactions access the same variable.
- Sometimes, however, there is shared data that was accessed in transactions but after a certain point it will only be accessed by one thread.
- Such data should be **privatized** in order to:
  - Avoid overhead of accesses in transactions.
  - Avoid prohibition of non-reversible actions in transactions — e.g. I/O

Transactional Memory vs Locking

- No deadlocks for TM but poor performance if there are many conflicts.
- Data must be partitioned for good performance:
  - avoiding lock contention,
  - avoiding transaction conflicts
- Critical section performance
  - full speed with locking — contention at synchronization only
  - contention can degrade performance anywhere during transaction
- Debugging:
  - naturally with locks
  - more difficult to set break points in the middle of a transaction
- Privatization:
  - trivial with locks
  - TM needs hardware support or performance penalty

The IBM Blue Gene/P Supercomputer

The IBM Blue Gene/P Supercomputer

- The IBM Blue Gene/P supercomputer was the first commercial machine that implements transactional memory in hardware.
- TM is implemented partly by having a version of each memory block.
- Recall that superscalar processors don't allow speculative instructions to modify memory.
- In Blue Gene/P they are allowed to write to the cache by also writing a version tag.
- With the version tag, aborted transactions can be rolled back.
- Sun's Rock also implemented TM (and other fancy things such as hardware scouts which are threads that can do prefetching) but was cancelled by Oracle when it bought Sun.
- Other implementations were done in software.

Power 2.07 Supports Transactional Memory

- The most recent version of the Power architecture, version 2.07, published May 10, 2013, supports TM.
- Every memory access is either transactional or non-transactional.
- New instructions include (the `.` suffix means they set the condition codes in `CR0`):
  - `tbegin.`
  - `tend.`
  - `tabort.`
  - `tsuspend.`
  - `tresume.`
  - `treclaim.`
  - `trechkpt.`
- Memory accesses executed between the `tbegin.` and `tend.` are transactional and all other accesses are non-transactional.

Overview

- The bit `TDOOMED` is set to 0 by `tbegin.` and to 1 at a failure.
- Most registers (but not `CR0` obviously) are saved at a `tbegin.` and are restored at a transaction failure.
- A failure handler is run at a transaction failure.
- A transaction can fail either due to itself or due to another transaction.
- It fails by itself (called self-induced) if:
  - It executes `tabort.` or `treclaim.`.
  - It has a too deep transaction nesting level at a new `tbegin.`.
  - It has a too large footprint (written too much data).
  - It executes a disallowed instruction — such as `doze`, `sleep` and `dcbi`.
- The failure handler should either retry the transaction or do the operation without a transaction (i.e. with locks).
- There is no guarantee of any forward progress or fairness by the hardware.

Nested Transactions

- Transactions can be nested.
- A `tend.` with field `A=1` ends all transactions of the thread and with `A=0` only ends the most recently started.
- A failure of a nested transaction terminates all transactions!

Conflicts

- A transaction conflicts with another transaction or a non-transactional access if they access the same cache block (i.e. memory block) and at least one is a store.
- At least one of two conflicting transactions fail, i.e. are aborted.
- Note the cache block granularity: since the cache block size is not defined by the architecture, software must be written accordingly.
- The reason for transaction failure is provided in a register.

Rollback-Only Transactions

- ROT transactions are speculative but not atomic.
- ROT's are started by a version of `tbegin.`.
- A ROT transaction does not conflict if it performs a load and the store was non-transactional.

Transaction Execution States

- There are three states:
  - Non-transactional: the normal state before any transaction is started.
  - Transactional: execution between a `tbegin.` and a `tend.`.
  - Suspended: execution by the same thread but as a temporary escape of the transactional state. This is execution between a `tsuspend.` and a `tresume.`.
- The purpose of the suspended state is for instance
  - Inter-thread communication: cannot be rolled back!
  - Other stores which should not be rolled back such as for debugging.
  - Accesses to non-cachable memory.
- Code in suspended state should be careful when accessing transactionally modified data since if the transaction is aborted at the same time the values may be either the transactionally stored or the rolled back values!

Problems with Normal Multiprocessors as Web Servers

- Data centers running databases and web servers have two big problems:
  - To achieve very high throughput so that no requests are blocked, and
  - the power consumption of the machines (multiprocessors).
- Throughput is perhaps more difficult. What is needed in conventional machines for throughput is high performance for each parallel thread.
- Since many requests are independent of each other, it is often very easy to run the server on a multiprocessor with good speedups.
- The problems of ever-more complex superscalar processor and longer-latency cache misses do not solve the throughput problem.

An Alternative to Normal Multiprocessors

- Normal multiprocessors are ideal for high-performance computations.
- Web servers typically don't need extremely high performance **single** threads but rather high throughput.
- As has been discussed in industry and academia for decades, one interesting architecture is multithreaded processors.
- While one thread in the pipeline is waiting for memory, hardware can schedule another thread immediately (e.g. the next clock cycle) to do useful work.
- In fact, web servers and multithreaded processors are a very good match.

Afari Websystems Inc

- Olukotun co-founded a start-up company, Afari Websystems Inc.
- The goal was to build server machines powered by multithreaded chip multiprocessors based on the SPARC architecture.
- After 9/11 the investor climate was not so good — in fact Olukotun was going to a business meeting in WTC that morning but late enough to avoid being murdered.
- Sun bought the start-up.
- Kunle Olukotun then spent time at Sun to develop the Niagara processor there.

Niagara Goals versus the Competition

- High throughput using multiple hardware threads and a crossbar to get very high bandwidth in the connect between the processors and a shared L2 cache.
- Lower power requirements than today's machines: e.g. Google need 400-700 W/sq foot while typical data centers support 70-150 W/sq. foot.
- Commercial server applications often have little instruction level parallelism and low cache hit rates, which means the power consumption made by superscalar processors typically is not worthwhile.

Niagara Overview

- The Niagara processor has 8 pipelines where each pipeline is shared by 4 threads.
- The shared cache is 3 MB and 12-way set associative.
- The crossbar provides 200 GB/s bandwidth.
- Each thread has its own store buffers and registers.
- Each pipeline is ***single-issue*** and has a Thread select in addition to the classic Fetch, Decode, Execute, Memory, and Write Back.
- Reminds us of early 1980s pipelines — except for thread select...
- Energy consumption by a Niagara processor is 74 W at 1.4 GHz.
- Compare it though with a complete PS3 which consumes about 95 W at 3.2 GHz (measured with a tool from Kjell & Co).

More Pipeline Details

- A long latency instruction, e.g. mul or div, causes a thread switch.
- Structural hazards (e.g. two threads that both need the divider) cause one of them to wait.
- Default is to select the least recently selected thread each cycle.
- A thread which wants to issue a load (i.e. something which might miss) is given lower priority than a thread with e.g. an add.
- The register file has three read ports and two write ports (for normal write and for e.g. divide that completes).

Further Developments

- The Niagara product name was UltraSPARC T1.
- The SPARC T2 has 8 cores and each core had 8 hardware threads.
- The SPARC T3 has 16 cores and each core has 8 hardware threads (now it is Oracle and not Sun).
- The SPARC T4 has 8 cores and each hardware thread has higher performance, and was released 2011.
- The SPARC T5 has 16 cores each with 8 threads, and the same higher performance hardware threads as SPARC T5.
- The current SPARC, M5, has 6 cores and 8 threads per core.
- The current Power, Power8, has 12 cores and 8 threads per core and uses simulatenous multithreading: 96 concurrent threads per chip.
- Intel Core i7 has 4 cores and 2 threads per core.

OpenCL

Contents of Lecture 12

- GPUs
- OpenCL
- About the exam

GPUs

- The term GPU was coined by Nvidia in 1999 when their GeForce 256 was launched
- It was a one chip processor for transformations, lighting, and triangle setup and clipping
- Performance was 10 million polygons per second
- Their competitor ATI launched Radeon 9700 in 2002.
- These early GPUs where not programmable but had APIs used by the host computer
- Modern GPUs are programmable e.g. in OpenCL — based on C

Differences between CPUs and GPUs

- CPUs have higher clock frequency compared with GPUs
- GPUs have many many more cores (e.g. hundreds or thousands)
- GPUs are good at SIMD processing
- GPUs cannot run OSes — no interrupt mechanism
- CPUs are better at control (such as if-then-else)
- CPU instruction sets are published while those of GPUs' are not
- GPU instructions can change between chip generations
- Intel makes integrated GPUs — on the same chip as the CPUs
- In Intel Core M, the GPU use about 60 % of the chip area
- CPUs and GPUs can complement each other

OpenCL

- OpenCL originates from Apple and is an Apple trademark
- Can e.g. be used with a C/C++ program on a host computer
- The purpose of OpenCL is to
  - Let software exploit all parallel hardware on a machine: CPUs, GPUs, DSPs etc
  - Provide a language that vendors agree on
  - Virtual memory works in compute devices such as GPUs — with OpenCl 2.0
  - Make software portable! (impossible with this hardware support?)
- How can this be achieved?

Portability despite different ISAs

- Due to different instruction set architectures, OpenCL programs are compiled at runtime!
- This costs some execution time (obviously) but achieves portability
- So each vendor provides an OpenCL-compiler and a runtime library.
- OpenCL is developed through an industry consortia called the Khronos Group

OpenCL

Based on C99

- No pointers to functions
- No bit-fields
- No variable-length arrays
- No structs
- New keyword `__kernel`
- A kernel is a function which should be executed by some compute engine

OpenCL Hardware Model

- Host — CPU
- Compute device — e.g. a GPU
- A compute unit — part of a compute device
- A processing element — part of a compute unit

An OpenCL Application

- Host code + device code
- Host tells device to transfer data to/from memory
- Host tells device to execute code
- An application consists of serial (only host) and parallel parts (run at devices)

Work items

- An N-dimensional domain is defined
- A ***kernel*** is executed on each point in the domain
- A kernel is a short function

```
__kernel void mul(
        __global const float* a,
        __global const float* b,
        __global float* c)
{
        int     id;

        id = get_global_id(0);
        c[id] = a[id] * b[id];
}
```


Execution Model

- A work-item is executed by a kernel function
- A ***context*** is the environment where a work-item is executed
  - device
  - device memory
  - command queue
- A ***command queue*** is used by the host to submit work to a device

Memory

- Private memory — per work item
- Local memory — shared within a work group
- Global/Constant memory — visible to all work groups
- Host memory — system RAM
- Programmer is responsible for transferring data

OpenCL language features

- Scalar types from C
- `image2d_t`, `image3d_t`, `sampler_t`
- Various vector types
- Memory fence — as in C11
- Memory barrier — all wait here

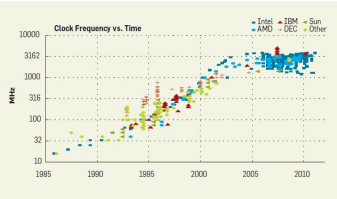
Kernel compilation

- A `cl_program` contains source code of kernels and the most recent successful build
- `clBuildProgram` performs the compilation
- Kernel source can be simply a string
- After setting up arguments, the code is sent to a device queue for execution

About the exam

- See reading advice on course page — I will update it with page numbers for Amazon printing
- There can be questions related to the labs and lecture notes (obviously)
- I will not about details such as language syntax
- There will be no questions where you will write code
- The questions are about "principles" and some "facts"
- What is meant by release and acquire belongs to "principles"
- I will not ask you to do dependence analysis but may ask about how a certain loop might be executed in parallel...
- Max 60 p, grade = $\lfloor score/10 \rfloor$
- There will be one simple question about history (2p).
- The group winning the competition will get coffee mugs at the exam.

The Rise of Multicore



CACM,
CPU DB: Recording Microprocessor History

The Rise of Multicore

Clock speeds stagnated around 3 GHz in 2005.

Performance gains now come from parallelism.

Current generation CPUs

- AMD Ryzen Threadripper: 8-16 cores
- Intel Core i9: 10-18 cores

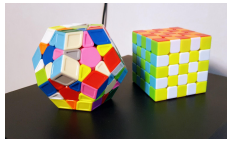
Parallelization

Parallel algorithms are

- complicated,
- hard to develop, and,
- very difficult to ensure correct.

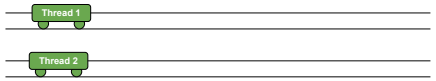
I like problems.

Hopefully you do, too!



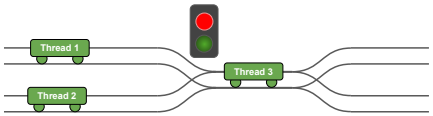
Parallel Programming

We can view threads as trains:



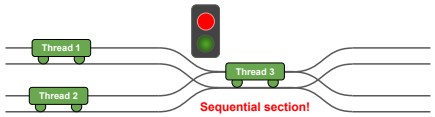
The computation they perform is the length of track covered.

Parallel Programming



Synchronization happens at intersections.

Parallel Programming



Locks force sequential execution.

Intermission: Amdahl's Law

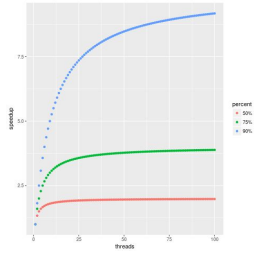
The potential speedup of parallelizing depends on the fraction of the work that can be parallelized.

Amdahl's law:

Speedup S from n-way parallelization:

S = 1 / (1 - p + p/n)

Where p is the fraction of work that can be executed in parallel.



Amdahl's Law

If 1/10th of the work is sequential, speedup limit is 10x.

Lock-Freedom

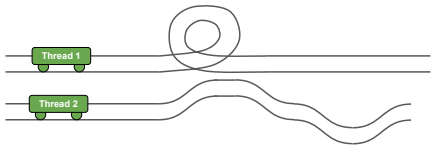
The goal of lock-free programming is to remove sequential bottlenecks.

☞ Absence of locks is not absence of synchronization/ordering.

Lock-free algorithms:
Can use retry mechanisms.
Can get stuck indefinitely due to unfair scheduling.

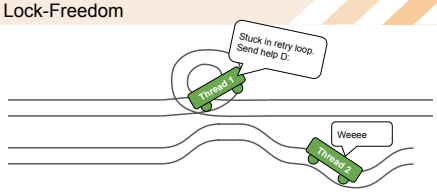
Wait-free algorithms:
Don't use unlimited retries.

Lock-Freedom



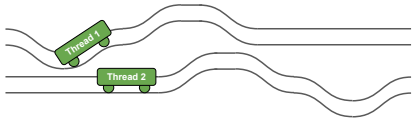
Lock-free: at least one thread guaranteed to complete.

Lock-Freedom



Lock-free: at least one thread guaranteed to complete.

Wait-Freedom



Wait-free: all threads guaranteed to complete!

Parallel Programming

How do we deal with parallelization in our code?

- High-level model of computation.
- High-level synchronization abstractions.

Reasoning about how hardware works is useful, but we use abstractions.
The exact details are too complicated, and change between hardware.

I will talk about the Java Memory Model.
All details do not apply to other languages!

Low- vs. High-Level

What you see is not what you get:

```
a = 3;
b = a * 2;
```

This can be optimized to

```
b = 6;
```

(This is a very obvious example.
Much less obvious reorderings are possible!)

Reordering

Instruction reordering is done by **compiler** and **processor**.

```
X = 3;
Y = 4;
```

Writes to memory can be reordered,
observed in the wrong order in different thread!

Reordering

Which states can T2 observe?

Thread 1:	**Thread 2:**
X = 3;	print(Y);
Y = 4;	print(X);

Reordering - Case 1

Which states can T2 observe?

Thread 1:		Thread 2:
X = 3;	✘	print(Y);
Y = 4;	✘	print(X);

Output:

0
0

Reordering - Case 2

Which states can T2 observe?

Thread 1:		Thread 2:
X = 3;	✔	print(Y);
Y = 4;	✘	print(X);

Output:

0
3

Reordering - Case 3

Which states can T2 observe?

Thread 1:		Thread 2:
X = 3;	✔	print(Y);
Y = 4;	✔	print(X);

Output:

4
3

Reordering - Case 4

Which states can T2 observe?

Thread 1:		Thread 2:
X = 3;	✘	print(Y);
Y = 4;	✔	print(X);

Output:

4
0

Reordering - Case 4

Which states can T2 observe?

Thread 1:		Thread 2:
X = 3;	✘	print(Y);
Y = 4;	✔	print(X);

Output:

4
0

Many other surprising cases here:
https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

Reordering - Case 4

Which states can T2 observe?

Thread 1:		Thread 2:
X = 3;	✘	print(Y);
Y = 4;	✔	print(X);

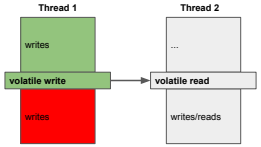
Output:

4
0

If **Y** is **volatile**,
this case is not possible!

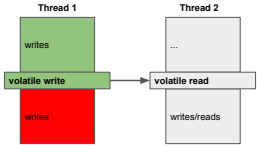
Volatile

1. Reading a volatile observes some previous write.
2. All previous writes from the observed write are visible.



Volatile

We can think of volatile read as lock acquisition,
…
and volatile write as lock release.



Volatile

Some problems can be solved with **volatile** flags:

```
volatile bool done = false;
int value;

int compute() {
    if (done) {
        return value;
    }
    value = 123;
    done = true;
    return value;
}
```

Volatile

Some problems can be solved with **volatile** flags:

```
volatile bool done = false;
int value;

int compute() {
    if (done) {
        return value;
    }
    value = 123;  // Multiple threads can enter here!
    done = true;
    return value;
}
```

Volatile

Volatile is a useful building block for ensuring safe publication!

Volatile is not strong enough to solve test-and-act problems that need mutual exclusion.

Other synchronization primitives like Compare-And-Set are used instead.

Fixpoint Functions

A fixpoint function **f(x)** is a function that has a fixed point **x**:

$$x = f(x)$$

Typically f(x) is defined in terms of f(x). It is **recursive**. Can be mutually recursive with other function(s).

Fixpoint Functions

A fixpoint function **f(x)** is a function that has a fixed point **x**:

$$x = f(x)$$

How can we find the fixed point **x**?

1. Start with some value.
2. Repeatedly apply **f** until the value is fixed.

Successive Approximation

```
int x = 0, old;
while (true) {
    old = x;
    x = f(x);
    if (x == old) break;
}
```

Initial value

Recompute

Change test

Fixpoint Functions

Fixpoint functions can be computed using

- **Change flag**: run the loop as long as a change is detected.
- **Worklist**: add nodes to worklist to propagate changes.

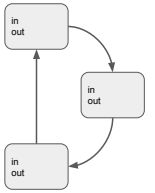
Liveness

Computing liveness is a fixpoint problem.

$$in = (out - def) \cup use$$
$$out = \bigcup_{s = succ} s.in$$

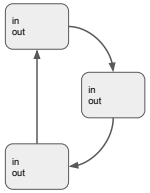
These equations are circularly defined!

Lock-based implementation could deadlock!



Liveness

The liveness algorithm used in this course is worklist based.



Sequential Liveness

Main loop:

```
while (!worklist.isEmpty()) {
    Vertex v = worklist.pop();
    v.compute(worklist);
}
```

```
class Vertex {
  BitSet in, out, use, def;

  void compute(List<Vertex> worklist) {}
}
```

Sequential Liveness

```
for (Vertex v : succ)
    out.or(v.in);
old = in;
in = new BitSet();
in.or(out);
in.andNot(def);
in.or(use);
if (!in.equals(old)) {
    for (Vertex v : pred)
        worklist.addLast(v);
}
```

Sequential Liveness

```
for (Vertex v : succ)
    out.or(v.in);
old = in;
in = new BitSet();
in.or(out);
in.andNot(def);
in.or(use);
if (!in.equals(old)) {
    for (Vertex v : pred)
        worklist.addLast(v);
}
```

Compute out set

Compute in set

Change test

Propagate change

Parallel Liveness

Parallelizing this sequential algorithm is not easy.

Most of your lab solutions are probably faulty - they worked by chance.

Lock-based Liveness

```
synchronized (this) {
    for (Vertex v : succ) {
        synchronized (v) {
            out.or(v.in);
        }
    }
    in = ...;
    if (!in.equals(old)) ...
}
```

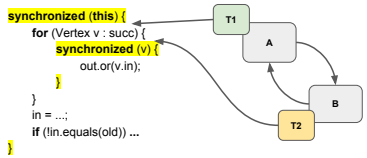
Lock-based Liveness

```
synchronized (this) {
    for (Vertex v : succ) {
        synchronized (v) {
            out.or(v.in);
        }
    }
    in = ...;
    if (!in.equals(old)) ...
}
```

Warning: circular dependency!

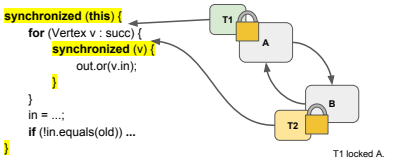
Lock-based Liveness

```
synchronized (this) {
    for (Vertex v : succ) {
        synchronized (v) {
            out.or(v.in);
        }
    }
    in = ...;
    if (!in.equals(old)) ...
}
```



Lock-based Liveness

```
synchronized (this) {
    for (Vertex v : succ) {
        synchronized (v) {
            out.or(v.in);
        }
    }
    in = ...;
    if (!in.equals(old)) ...
}
```



Deadlock!

T1 locked A.
T2 locked B and tries to lock A.

Lock-based Liveness (2)

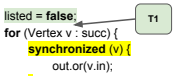
```
listed = false;
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
...
```

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```

Lock-based Liveness (2)

```
listed = false;
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
...
```

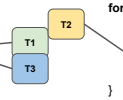
```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```



Lock-based Liveness (2)

```
listed = false;
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
...
```

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```



Data race!

T1 and T3 try to modify the same out set. Bad things happen.

Lock-based Liveness (3)

```
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
listed = false;
...
```

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```

Lock-based Liveness (3)

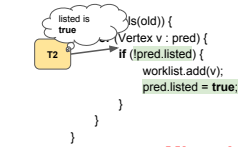
```
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
listed = false;
...
```

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```

Lock-based Liveness (3)

```
for (Vertex v : succ) {
    synchronized (v) {
        out.or(v.in);
    }
}
synchronized (this) {
    in = ...;
}
listed = false;
...
```

```
if (!in.equals(old)) {
    for (Vertex v : pred) {
        if (!pred.listed) {
            worklist.add(v);
            pred.listed = true;
        }
    }
}
```



T1 updated the vertex based on old state.

Missed update!

How to catch concurrency errors?

It can work fine on your machine even if you have a broken algorithm.

Running multiple times helps, but guarantees nothing!

Valgrind gives no guarantees.

It does not help to run more threads!

More threads can reduce the failure rate.

How to catch concurrency errors?

Think about your invariants and try to disprove them!

This leads to insight into possible flaws.

Invariants

Invariants for the liveness algorithm:

- A node must be updated it is in, or will be added to, the worklist.
- Only one thread works on a single node at a time.
- A thread should not hold multiple locks at the same time.

Try to disprove these to gain insight!

Try to remove invariants if possible.

Can you redesign the algorithm to remove an invariant,
or make it easier to ensure that it holds?

Compare And Set

Atomically test-and-set a variable:

```
AtomicReference r = new AtomicReference(null);
r.compareAndSet(null, foo);
```

Only succeeds if the value of **r** is **null**.

Atomically updates **r** to the **foo** reference.

Lock-Free Liveness

Main idea:

Take a snapshot of the state of the current vertex.

Only commit state update if no other thread changed the state.

Lock-Free Liveness

```
AtomicReference<BitSet> in;
while (true) {
    BitSet oldIn = in.get();
    ...
    if (!out.compareAndSet(oldOut, newOut)) {
        // Someone else changed the out set.
        continue;
    }
    ...
}
```

Take snapshot

Compute

Try update

Restart

Lock-Free Liveness

Rinse and repeat for the in set!

Attribute Grammars

Attributes are an abstraction used in compilers.

Compilation is split up into **attributes** - essentially, small functions.

Attributes are declarative and pure (no side effects).

Attribute Grammars

Attributes are automatically scheduled by an attribute evaluator.

Attribute evaluation can be freely reordered.

Attribute Grammars

JastAdd is a metacompilation tool:

A tool for building compilers.

jastadd

I recently implemented concurrent attribute evaluation in JastAdd.

The project was funded by a 2015 Google Faculty Research Award.

Concurrent attributes have been used to parallelize a Java compiler, with a resulting 2x speedup!

Attribute Liveness

Circular attributes are used to compute fixpoint functions!

For example, liveness can be computed using attributes!

It's the most compact way, counting code size.

Not the most efficient, but a nice demonstration of the abstraction power of attributes!

Attribute Liveness

```
syn BitSet Vertex.out() {
  BitSet out = new BitSet();
  for (Vertex s : getSucc())
    out.or(s.in());
  return out;
}
```

```
syn BitSet Vertex.in() circular [new BitSet()] {
  BitSet in = new BitSet();
  in.or(out());
  in.andNot(getDef());
  in.or(getUse());
  return in;
}
```

Attribute Liveness

```
syn BitSet Vertex.out() {
  BitSet out = new BitSet();
  for (Vertex s : getSucc())
    out.or(s.in());
  return out;
}
```

Fixpoint function.

```
syn BitSet Vertex.in() circular [new BitSet()] {
  BitSet in = new BitSet();
  in.or(out());
  in.andNot(getDef());
  in.or(getUse());
  return in;
}
```

Initial value.

Attribute Liveness

Compute liveness for all vertices:

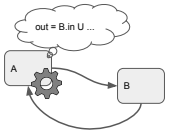
for (Vertex v : vertices) v.in();

This computes all in/out sets because **in** is recursively defined, and uses **out**.

Attribute Evaluation

How does it work?

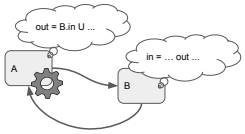
Recursive dynamic computation of the attribute function:



Attribute Evaluation

How does it work?

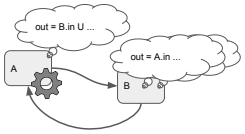
Recursive dynamic computation of the attribute function:



Attribute Evaluation

How does it work?

Recursive dynamic computation of the attribute function:



Circular Attribute Evaluator

Global state:

CYCLE - Keeps track of the current iteration in the fixpoint loop.
CHANGE - Global change flag. Fixpoint iteration is done when it remains false.

Each vertex:

in_value - Current approximation.
done - Flag indicating if the attribute is fully computed.
visited - Tracks the last iteration the attribute value was computed in.
Used as a visit flag.

Circular Attribute Evaluator

```
BitSet Vertex.in() {
    if (done) return in_value;        // Cache check.

    if (CYCLE == 0) {
        // Case 1: Start fixpoint evaluation!
    } else if (visited != CYCLE) {
        // Case 2: Compute value.
    } else {
        // Case 3: Reuse current value.
    }
    return in_value;
}
```

Circular Attribute Evaluator

```
// Case 1: Start fixpoint evaluation!
do {
    CYCLE += 1;
    CHANGE = false;
    visited = CYCLE;
    next = in_compute();
    if (!next.equals(in_value)) {
        in_value = next;
        CHANGE = true;
    }
} while (CHANGE);
done = true;
```

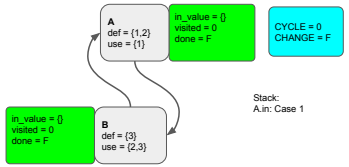
Circular Attribute Evaluator

```
// Case 2: Compute value.
visited = CYCLE;
next = in_compute();
if (!next.equals(in_value)) {
    in_value = next;
    CHANGE = true;
}
```

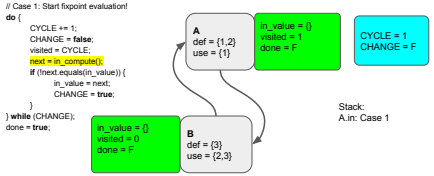
Circular Attribute Evaluator

```
// Case 3: Reuse current value.
return in_value;
```

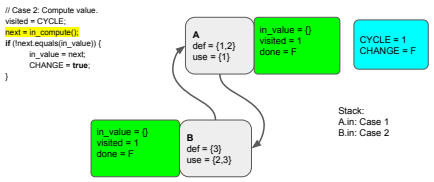
Attribute Evaluation



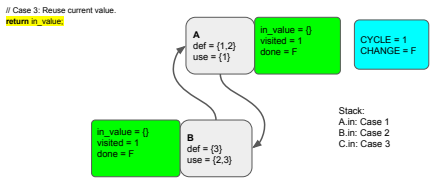
Attribute Evaluation



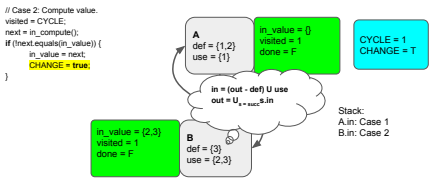
Attribute Evaluation



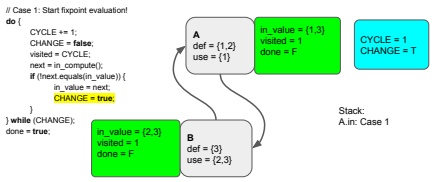
Attribute Evaluation



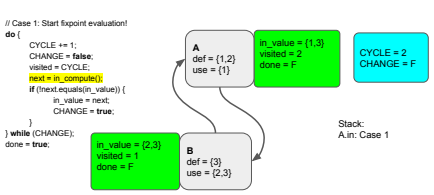
Attribute Evaluation



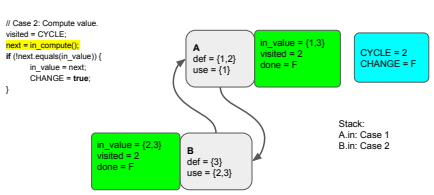
Attribute Evaluation



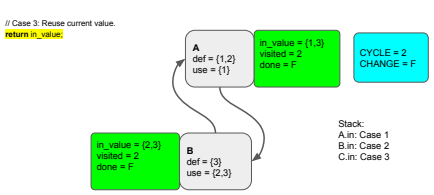
Attribute Evaluation



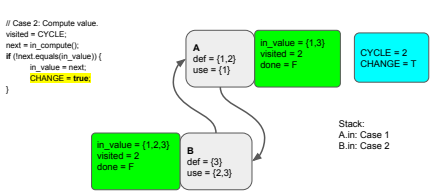
Attribute Evaluation



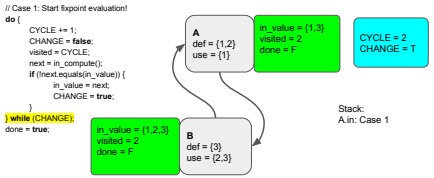
Attribute Evaluation



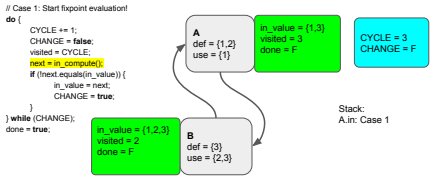
Attribute Evaluation



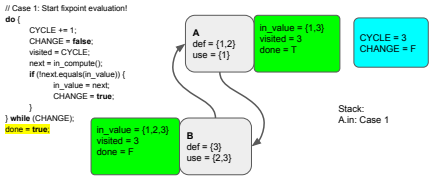
Attribute Evaluation



Attribute Evaluation



Attribute Evaluation



Concurrent Evaluation

Challenges in making the circular attribute evaluator concurrent:

Attribute approximations need to be safely shared between threads.

Can't use locks - would lead to deadlock in in circular eval!

Concurrent Evaluation

Changes in the concurrent algorithm:

Each thread has their own thread-local CHANGE and CYCLE states.

The per-attribute visit information is stored in a thread-local map.

Attribute value and done flags are combined into a tuple and stored using AtomicReference.

Thread-Local State

In Java, you can use ThreadLocal to store persistent thread-local data.

Thread-local state of the concurrent evaluator:

- **tls.iter** - maps attributes to iteration indices
- **tls.cycle** - current iteration index
- **tls.change** - change flag

Global State

Each vertex has the following global state:

```
CircularAttributeValue in_value;

class CircularAttributeValue {
    volatile bool done = false;
    AtomicReference value = new AtomicReference(NIL);
}
```

NIL is an invalid attribute value.

Concurrent Cache Check

First test if it was already computed:

```
if (in_value.done) {
    // done is volatile, ensuring safe publication:
    return in_value.get();
}
```

If not, compute the attribute. Does not matter if another thread just finished computing the attribute, we'll just get the same value!

Concurrent Compute

To compute the attribute value, each thread first reads the current value:

```
prev = in_value.value.get();

next = in_compute();

if (!next.equals(prev)) {
    in_value.value.compareAndSet(prev, next);
    tls.change = true;
}
```

Take snapshot

Compute

Change test

Try update

Lock-Freedom

This concurrent algorithm is lock-free and wait-free!

If a thread fails to update an attribute value, it will just the other thread's result going forward. (No unlimited retries)

The concurrent algorithm works for any fixpoint function, not just this liveness example.

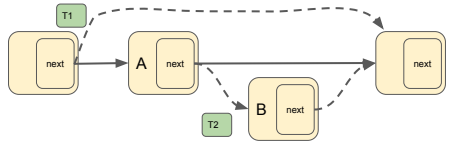
Other Topics

Other topics to discuss if there is time:

- Lock-free linked lists
- Skiplists
- Universal wait-free construction

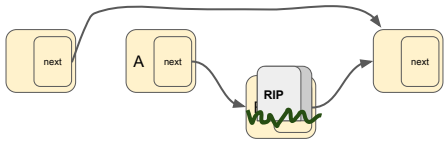
Concurrent Linked List

Thread 1 deletes node A, thread 2 deletes node B:



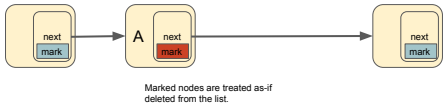
Concurrent Linked List

Result: node B was lost!



Lock-Free Linked List

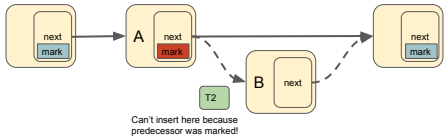
Solution: add a mark field to logically delete nodes.



Marked nodes are treated as-if deleted from the list.

Lock-Free Linked List

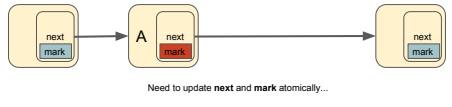
Solution: add a mark field to logically delete nodes.



Can't insert here because predecessor was marked!

Lock-Free Linked List

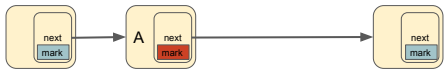
Physical deletion is done separately.



Need to update **next** and **mark** atomically...

Lock-Free Linked List

Physical deletion is done separately.



Need to update **next** and **mark** atomically…

Use tuple or **AtomicMarkedReference**

Then we can use CAS to atomically update the next pointer and the mark.

Linearizability

Overlapping linearizable functions calls work as if they took effect in a sequential order: