Lecture 7: Other Concurrency Models

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Two Aspects of Parallel Programming

Correctness
avoiding race conditions and deadlocks

Performance
efficient use of resources
Concurrent Platforms

- Libraries
  - .NET Thread pool class
  - Message Passing Libraries
    - MPI
  - Task-parallel Libraries
    - Intel TBB

- Data-Parallel Programming Languages
  - RapidMind → Intel Ct
  - NESL

- Functional Languages
  - Haskell
  - Erlang

- Parallel Language Extensions
  - OpenMP
  - Cilk/ Cilk++

Where is Pthreads btw??
Threads ... PThreads
Multithreading
Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs
A Thread

- **Definition**: sequence of related instructions executed independently of other instruction sequences
- A thread can create another thread
- Each thread maintains its current machine state

![Thread Levels]

- User-Level Threads
- Kernel-Level Threads
- Hardware Threads
User-Lever Threads (ULT)

- All thread management is done by the application.
- The kernel is not aware of the existence of threads.
User-Level Threads (ULTs)

Advantages

• Thread switch does not require kernel-mode.
• Scheduling (of threads) can be application specific.
• Can run on any OS.

Disadvantages

• A system-call by one thread can block all threads of that process.
• In pure ULT, multithreading cannot take advantage of multiprocessing.
Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- no thread management is done by the application
- Windows OS is an example of this approach
Kernel-Level Threads (KLTs)

**Advantages**

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines can be multithreaded

**Disadvantages**

- The transfer of control from one thread to another within the same process requires a switch to the kernel
Combined (Hybrid) Approach

- Thread creation is done completely in user space.
- Bulk of scheduling and synchronization of threads is by the application (i.e. user space).
- Multiple ULTs from a single application are mapped onto (smaller or equal) number of KLTs.
- Solaris is an example
Relationship Between ULTs & KLTs

• 1:1
  - user-level thread maps to kernel-level thread
  - e.g. win32, Linux (original C-library), windows 7, FreeBSD

• N:1 (user-level threads)
  - Kernel is not aware of the existence of threads
  - e.g. Early version of Java, Solaris Green Thread

• M:N
POSIX Threads

- **Portable Operating System Interface**
- Is an IEEE standard
- API
- Maintain compatibilities among OSes
- Pthreads $\rightarrow$ a POSIX standard for threads
POSIX Threads (Pthreads)

- Low-level threading libraries
- Native threading interface for Linux now
- Use kernel-level thread (1:1 model)
  - starting from kernel 2.6
- developed by the IEEE committees in charge of specifying a Portable Operating System Interface (POSIX)
- Shared memory
POSIX Threads (Pthreads)

- C language programming types and procedure calls
- implemented with a `pthread.h` header
- To compile with GNU compiler, 2 methods:
  - `gcc/g++ progrname -lpthread`
  - `gcc/g++ -pthread progrname`
- Capabilities like thread priority are not part of the core pthreads library.
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable: accessible to all threads */
int thread_count;

void *Hello(void* rank); /* Thread function */

int main(int argc, char* argv[]) {
    long thread; /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;

    /* Get number of threads from command line */
    thread_count = strtol(argv[1], NULL, 10);

    thread handles = malloc (thread_count*sizeof(pthread_t));
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);

    free(thread_handles);
    return 0;
} /* main */

void *Hello(void* rank) {
    long my_rank = (long) rank; /* Use long in case of 64-bit system */
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);

    return NULL;
} /* Hello */
Running a Pthreads program

./helloworld  <number of threads>

./helloworld  1

    Hello from the main thread
    Hello from thread 0 of 1

./helloworld  4

    Hello from the main thread
    Hello from thread 0 of 4
    Hello from thread 1 of 4
    Hello from thread 2 of 4
    Hello from thread 3 of 4
Starting the Threads

```c
int pthread_create ( 
    pthread_t * thread_p /* out */, 
    const pthread_attr_t* attr_p /* in */, 
    void* (*start_routine) ( void * ) /* in */, 
    void* arg_p /* in */ );
```
pthread_t objects

- **Opaque**
  - The actual data that they store is system-specific.
  - Their data members aren’t directly accessible to user code.

- **Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it’s associated.**
A closer look

int pthread_create (  
    pthread_t* thread_p /* out */ ,  
    const pthread_attr_t* attr_p /* in */ ,  
    void* (*start_routine) ( void * ) /* in */ ,  
    void* arg_p /* in */ ) ;

If you want to keep the default, just use NULL.

Allocate before calling.

Pointer to the argument that should be passed to the function.

The function that the thread is to run.
Function started by pthread_create

- Prototype:
  ```c
  void* thread_function ( void* args_p );
  ```

- Void* can be cast to any pointer type in C.

- So args_p can point to a list containing one or more values needed by thread_function.
- Similarly, the return value of thread_function can point to a list of one or more values.
Stopping the Threads

• We call the function `pthread_join` once for each thread.
• Blocking function.

```c
int pthread_join(  
    pthread_t    thread;   /* in */  
    void **     ret_val;   /* out */  
);  
```
### Matrix-Vector Multiplication in PThreads

The matrix-vector multiplication in PThreads can be represented as follows:

\[
\begin{align*}
\begin{array}{cccc}
    a_{00} & a_{01} & \cdots & a_{0,n-1} \\
    a_{10} & a_{11} & \cdots & a_{1,n-1} \\
    \vdots & \vdots & & \vdots \\
    a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
    \vdots & \vdots & & \vdots \\
    a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\end{array}
\end{align*}
\begin{align*}
\begin{array}{c}
y_0 \\
y_1 \\
\vdots \\
y_{m-1} \\
\end{array}
\end{align*}
\]

\[
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}
\]
Serial pseudo-code

/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j] * x[j];
}

\[ y_i = \sum_{j=0}^{n-1} a_{ij} x_j \]
Using 3 Pthreads

<table>
<thead>
<tr>
<th>Thread</th>
<th>Components of y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y[0], y[1]</td>
</tr>
<tr>
<td>1</td>
<td>y[2], y[3]</td>
</tr>
<tr>
<td>2</td>
<td>y[4], y[5]</td>
</tr>
</tbody>
</table>

**General Case**

\[
y[i] = 0.0; \\
\text{for } (j = 0; j < n; j++) \\
y[i] += A[i][j] \times x[j];
\]

**Thread 0**

\[
y[0] = 0.0; \\
\text{for } (j = 0; j < n; j++) \\
y[0] += A[0][j] \times x[j];
\]
Pthreads matrix-vector multiplication

void *Pth_mat_vect(void * rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i][j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */
That previous example was easy!

- $A$, $X$, and $Y$ can all be global.
- Threads update different elements of $Y$ simultaneously $\rightarrow$ no problem

Let’s see a not so easy one!
Estimating $\pi$

$$\pi = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right)$$

double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
A thread function for computing $\pi$

```c
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0) /* my_first_i is even */
        factor = 1.0;
    else /* my_first_i is odd */
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        sum += factor/(2*i+1);
    }

    return NULL;
} /* Thread_sum */
```
Using a dual core processor

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$10^5$</td>
</tr>
<tr>
<td>$\pi$</td>
<td>3.14159</td>
</tr>
<tr>
<td>1 Thread</td>
<td>3.14158</td>
</tr>
<tr>
<td>2 Threads</td>
<td>3.14158</td>
</tr>
</tbody>
</table>

Note that as we increase $n$, the estimate with one thread gets better and better!!

Reason: Race Condition in updating sum by more than one thread.
Possible race condition

Assume two threads are executing the following code, with \( x \) initially 0:

\[
\begin{align*}
y &= \text{computer}(\text{my\_rank}) \\
x &= x + y;
\end{align*}
\]

---

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Started by main thread</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Call \text{Compute}()</td>
<td>Started by main thread</td>
</tr>
<tr>
<td>3</td>
<td>Assign ( y = 1 )</td>
<td>Call \text{Compute}()</td>
</tr>
<tr>
<td>4</td>
<td>Put ( x=0 ) and ( y=1 ) into registers</td>
<td>Assign ( y = 2 )</td>
</tr>
<tr>
<td>5</td>
<td>Add 0 and 1</td>
<td>Put ( x=0 ) and ( y=2 ) into registers</td>
</tr>
<tr>
<td>6</td>
<td>Store 1 in memory location ( x )</td>
<td>Add 0 and 2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Store 2 in memory location ( x )</td>
</tr>
</tbody>
</table>
Solution 1: Busy Waiting

flag initialized to 0 by main thread

```c
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

• This will ensure no race condition ... But:
  – Busy doing nothing
  – serialization
  – optimizing compilers can mess with it!
Pthreads global sum with busy-waiting

```c
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        while (flag != my_rank);
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */
```

In dual core with n = $10^8$
serial $\rightarrow$ 2.8s
2 threads $\rightarrow$ 19.5s!
Optimization: Global sum function with critical section after loop

```c
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
        my_sum += factor/(2*i+1);

    while (flag != my_rank);
    sum += my_sum;
    flag = (flag+1) % thread_count;

    return NULL;
} /* Thread_sum */
```

In dual core
with n = \(10^8\)
serial ➔ 2.8s
2 threads ➔ 1.5s!

But still the thread
is using the CPU doing
nothing ➔ busy waiting
is not the ideal solution
for mutual exclusion
Solution 2: Mutexes

• A thread that is busy-waiting may continually use the CPU accomplishing nothing.

• Mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.
Mutexes

• The Pthreads standard includes a special type for mutexes: `pthread_mutex_t`.

```c
int pthread_mutex_init(
    pthread_mutex_t* mutex_p, /* out */
    const pthread_mutexattr_t* attr_p /* in */);
```
 Mutexes

• When a Pthreads program finishes using a mutex, it should call

```c
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```

• In order to gain access to a critical section a thread calls

```c
int pthread_mutex_lock(pthread_mutex_t* mutex_p /* in/out */);
```

• When a thread is finished executing the code in a critical section, it should call

```c
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```
Global sum function that uses a mutex

```c
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;
    double my_sum = 0.0;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        my_sum += factor/(2*i+1);
    }
    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);

    return NULL;
} /* Thread_sum */
```
Run-times (in seconds) of $\pi$ programs using $n = 10^8$ terms on a system with two four-core processors.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Busy-Wait</th>
<th>Mutex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.90</td>
<td>2.90</td>
</tr>
<tr>
<td>2</td>
<td>1.45</td>
<td>1.45</td>
</tr>
<tr>
<td>4</td>
<td>0.73</td>
<td>0.73</td>
</tr>
<tr>
<td>8</td>
<td>0.38</td>
<td>0.38</td>
</tr>
<tr>
<td>16</td>
<td>0.50</td>
<td>0.38</td>
</tr>
<tr>
<td>32</td>
<td>0.80</td>
<td>0.40</td>
</tr>
<tr>
<td>64</td>
<td>3.56</td>
<td>0.38</td>
</tr>
</tbody>
</table>
There are applications where:

- We need to control the order threads access the critical section.
- We may need to have more than one thread in the critical section.
Syntax of the various semaphore functions

```c
#include <semaphore.h>

int sem_init(
    sem_t* semaphore_p /* out */,
    int shared /* in */,
    unsigned initial_val /* in */);

int sem_destroy(sem_t* semaphore_p /* in/out */);
int sem_post(sem_t* semaphore_p /* in/out */);
int sem_wait(sem_t* semaphore_p /* in/out */);
```

Semaphores are not part of Pthreads; you need to add this.

Non-zero means semaphore shared among processes.
sem_post and sem_wait

- **sem_post**: increment the semaphore
- **sem_wait**: check semaphore value
  - if 0 then block
  - else: decrement and proceed
Barriers

• **What?** Synchronizing the threads to make sure that they all are at the same point in a program

• **Why?**
  – For correctness
  – For debugging
  – For timing

• **Small challenge:** Many implementations of Pthreads do not provide barriers.
Steps

1. Declare a barrier variable (global):
   ```
   pthread_barrier_t barr;
   ```

2. Initialize the variable (by one thread):
   ```
   pthread_barrier_init(&barr, NULL, num_threads)
   ```

3. Use it (by all threads):
   ```
   pthread_barrier_wait(&barr);
   ```

4. Destroy it (by one thread):
   ```
   pthread_barrier_destroy(&barr)
   ```
There are many more APIs but you got the idea.
The Problem With Threads

- Paper by Edward Lee, 2006
- The author argues:
  - “From a fundamental perspective, threads are seriously flawed as a computation model”
  - “Achieving reliability and predictability using threads is essentially impossible for many applications”
- The main points:
  - Our abstraction for concurrency does not even vaguely resemble the physical world.
  - Threads are dominating but not the best approach in every situation
  - Yet threads are suitable for embarrassingly parallel applications
The Problem With Threads

• The logic of the paper:
  – Threads are nondeterministic
  – Why shall we use nondeterministic mechanisms to achieve deterministic aims?
  – The job of the programmer is to prune this nondeterminism.
  – This leads to poor results

Do you agree or disagree with the author?
Haskell

http://www.haskell.org
What is Functional Programming?

• Functional programming is style of programming in which the basic method of computation is the application of functions to arguments.

```java
    total = 0;
    for (i = 1; i <= 10; ++i)
        total = total+i;
```

- Summing up number from 1 to 10
- Summing up number from 1 to 10
- Imperative programming
- Mainly variable assignment
What does this have to do with multicore programming?

• One of the hardest things in parallel programming is locking

• Functional programming does not use mutable data → data is separate from the logic → no side effects → order does not matter → parallelism is easier to find

• We will look at one excellent example of functional programming: HASKELL
Haskell Is:

- Memory managed
- The #1 on Language Shootout for threading
- Pure functional programming
- Simplest Haskell program:
  \[
  \text{main} = \text{return}() 
  \]
Values and Expressions

• A *value* is a piece of data.

  2, 4, 3.14159, ”John”,

• An *expression* computes a value.

  2 + 2, 2*pi*r

• Expressions combine values using *functions* and *operators*. 
Functions

The solution of a quadratic equation:

\[-b + \sqrt{b^2 - 4ac} \div 2a\]
A *definition* gives a name to a value.

```plaintext
area :: Int
area = 41*37
```

- Names start with a small letter, and are made up of letters and digits.
- Types specify what kind of value this is.
- An expression says how the value is computed.
In Haskell, \( f :: A \rightarrow B \) means for every \( x \in A \),

\[
f(x) = \begin{cases} 
    \text{some element } y = f(x) \in B \\
    \text{run forever}
\end{cases}
\]

In words, “if \( f(x) \) terminates, then \( f(x) \in B \).”
A function definition specifies how the result is computed from the arguments.

\[
\text{area} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{area} ~ l \ b = l * b
\]

- Function types specify the type of the arguments and the result.
- The body specifies how the result is computed.
- The arguments are given names, after the function name.
Function Notation

Function arguments need **not** be enclosed in brackets!

Example: average :: Float -> Float -> Float

average x y = (x + y) / 2

Calls:

- average 2 3 → 2.5
- average (2+2) (3*3) → 6.5
Functional Programming

• A functional program consists mostly of function definitions.

• Simple functions are used to define more complex ones, which are used to define still more complex ones, and so on.

• Finally, we define a function to compute the output of the entire program from its inputs.
Compiling Your Haskell Program

• to get the interactive system:
  $ ghci

• To generate an executable:
  $ ghc -o hello hello.hs

• Check:
  http://www.haskell.org/haskellwiki/Haskell_in_5_steps
Transactional Memories
What Are We Talking About?

• Incorporating transactions in parallel programming → computations wrapped in transactions
• A alternative way to coordinate concurrent threads
• Characteristics of a transaction: ACI
  – Atomicity
  – Consistency
  – Isolation
Databases!!

• Database systems have successfully exploited parallel hardware for decades.

• Databases achieve good performance by executing many queries simultaneously and by running queries on multiple processors when possible.

• The author of an individual query need not worry about this parallelism!
Databases!!

- DB programming model → transactions
- Computation executes as if it was the only computation accessing the DB.
- Results indistinguishable from the situation in which the transactions run one after the other → serializability
- Transactions allow concurrent operations to access a common DB and still produce predictable, reproducible results.
Why Don’t we Learn from DB?

Transactional Memory

In multicore, the main data storage during execution is typically the memory.
A transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer.

**Atomicity**
All constituent actions in a transaction complete successfully, or none of these actions appear to start executing.

**Consistency**
Application dependent

**Isolation**
Transactions do not interfere with each other while they are running, regardless of whether or not they are executing in parallel.

**Durability**
Once a transaction commits, its result is permanent.

ACID Properties
Example

Thread 1

```
begin_xaction
A = A - 20
B = B + 20
A = A - B
C = C + 20
end_xaction
```

Thread 2

```
begin_xaction
C = C - 30
A = A + 30
end_xaction
```

Thread 1’s accesses and updates to A, B, C are atomic.

Thread 2 sees either “all” or “none” of Thread 1’s updates.
Another Example

```c
int x = 0; int y = 0;

T1

atomic {
    x = 42;
    y = 42;
}

T2

atomic {
    int tmp1 = x;
    int tmp2 = y;
}
```

What values does T2 see?
Yet Another Example

- Transactions appear to execute in commit order
  - Flow (RAW) dependency cause transaction violation and restart

```
ld 0xdddd
...
st 0xbeef
```

```
ld 0xdddd
...
ld 0xbeef
```

```
ld 0xbeef
```

```
ld 0xbbbb
```

Violation!

Re-execute with new data
Who Uses TM?

• Programmer
• Compiler designer to implement some high-level language features
• **Important**: Beside using TM for parallel programming, it can also be used in:
  – error recovery
  – real-time programming
  – multitasking
Basic TM

// Transaction management
void StartTx();
bool CommitTx();
void AbortTx();

// Data access
T ReadTx(T *addr);
void WriteTx(T *addr, T v);

Important: Different implementations of TM may have different names for functions.
void PushLeft(DQueue *q, int val) {
    QNode *qn = malloc(sizeof(QNode));
    qn->val = val;
    QNode *leftSentinel = q->left;
    QNode *oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
}

void PushLeft(DQueue *q, int val) {
    QNode *qn = malloc(sizeof(QNode));
    qn->val = val;
    do {
        StartTx();
        QNode *leftSentinel = ReadTx(&q->left);
        QNode *oldLeftNode = ReadTx(&(leftSentinel->right));
        WriteTx(&(qn->left), leftSentinel);
        WriteTx(&(qn->right), oldLeftNode);
        WriteTx(&(leftSentinel->right), qn);
        WriteTx(&((oldLeftNode->left), qn));
    } while (!CommitTx());
}
Concurrency Control: Conflict-Detection-Resolution

• A conflict occurs when two transactions perform conflicting operations on the same piece (2 writes, or a read and a write)

• The conflict is detected when the underlying TM system determines that the conflict has occurred.

• The conflict is resolved when the underlying system or code in a transaction takes some action to avoid the conflict—e.g., by delaying or aborting one of the conflicting transactions.
Simply Speaking ...

- TM makes private copy of shared data.
- The transaction updates the private copy.
- If shared data is unchanged
  - Update shared data with private copy
- Else conflict has occurred
  - Discard private copy and repeat transaction
Some TM implementations use Pessimistic control, others use Optimistic control.
<table>
<thead>
<tr>
<th>Read</th>
<th>Optimistic</th>
<th>Pessimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>Optimistic</td>
<td>TCC</td>
<td>Intel C++ STM</td>
</tr>
<tr>
<td></td>
<td>TL2</td>
<td>Intel Java STM</td>
</tr>
<tr>
<td></td>
<td>SigTM</td>
<td>HASTM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Microsoft OSTM</td>
</tr>
<tr>
<td>Pessimistic</td>
<td>LogTM</td>
<td>Intel C++ STM</td>
</tr>
</tbody>
</table>
Version Management

• What to do about writes before a transaction commits?
• Eager version management
  – The transaction directly modifies the data in memory
  – Keeps an undo-log holding overwritten data
  – Requires pessimistic concurrency control
• Lazy version management
  – Updates are delayed until a transaction commits
  – Transaction maintains its tentative writes in a transaction-private redo-log
Conflict Detection

• With pessimistic approach it is easy \( \rightarrow \) locks!
• With optimistic approach, there are several issues:
  – Granularity of conflict (cache line, objects, …)
  – The time at which detection occurs:
    • When transaction declares its intend to access the data \( \rightarrow \) eager conflict detection
    • On validation: can occur several times during transaction lifetime
    • On commit \( \rightarrow \) lazy conflict detection
  – Which kind of access is treated as conflicts?
    • Among concurrent transactions
    • Between active and committed transactions
Can We Make Things Simpler?

- Things look very verbose hence error prone
- Instead of WriteTx and ReadTx can we have something simpler?
- **Atomic block** of statements
  - getting rid of WriteTx and ReadTx
  - Implemented for many languages
How to Provide the *Illusion* of Transactions?

- **Software Transactional Memory (STM)**
- **Hardware Transactional Memory (HTM)**
Software Transactional Memory (STM)

Components:

- **transaction descriptor**: is the per-transaction data structure that keeps track of the state of the transaction
- **Undo-log or Redo-log**
- **read-set or write-set**: tracks the memory locations that the transaction has read from or written to
STM

- Compiler instruments code with transaction prolog, epilog, and read/write function.
- Runtime tracks memory accesses, detects conflicts, and commits/aborts execution.

```
atomic {
    r = x;
    y = r + 1;
}
```

```plaintext
td = getXnDesc();
txnBegin(td);
r = txnReadInt(td, &x);
txnWriteInt(td, &y, r+1);
txnEnd(td);
```
Hardware Transactional Memory (HTM)

• Three flavors
  – Full implementation of TM in hardware
  – Allowing hardware transactions to coexist with software transactions
  – hardware extension to provide speed-up to parts of software TM
HTM

- HTM must perform the following functions
  - identify memory locations for transactional accesses
  - manage the read-sets and write-sets of the transactions
  - detect and resolve data conflicts,
  - manage architectural register state
  - commit or abort transactions
# Requirements for Supporting Transactions

<table>
<thead>
<tr>
<th>(Buffering)</th>
<th>Transactional cache</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conflict detection</strong></td>
<td>Cache coherence protocol</td>
</tr>
<tr>
<td><strong>Abort/Recovery</strong></td>
<td>Invalidate transactional cache line</td>
</tr>
<tr>
<td><strong>Commit</strong></td>
<td>Validate transactional cache line</td>
</tr>
</tbody>
</table>
HTM

- Extensions to the instruction set
- Tracking read-sets and buffering write-sets is done using caches and buffers
- Coherence messages trigger conflict detection
- Nearly all conventional HTM proposals perform eager conflict detection
Hardware Support for Performance

Core 1

begin_xaction
A.withdraw(20)
B.deposit(20)
end_xaction

A.sum = 100
B.sum = 200

Architectural Memory state
A.sum = 80
B.sum = 220

1. Record recovery state
2. Buffer updates/track accesses
3. Commit if no external access (discard all updates if conflict)

Core 2

begin_xaction
Sum = A.sum + B.sum
end_xaction

A.sum = 100
B.sum = 200

Coherence protocol for conflicts

Core 2 sees $300 – never $280 or $320

Source: Konrad Lai (Intel) slides “Transactional Memories”
STM vs HTM

• Software is more flexible than hardware and permits the implementation of a wider variety of more sophisticated algorithms.
• Software is easier to modify and evolve than hardware.
• STMs can integrate more easily with existing systems and language features, such as garbage collection.
• STMs have fewer intrinsic limitations imposed by fixed-size hardware structures, such as caches.
**HTM vs STM**

- HTM systems can typically execute applications with lower overheads than STM systems.
- Less reliant than STM systems on compiler optimizations to achieve performance.
- HTM systems can have better power and energy profiles.
- Treat all memory accesses within a transaction as implicitly transactional.
- HTM systems can provide strong isolation without requiring changes to non-transactional memory accesses.
- HTM systems are well suited for systems languages such as C/C++ that operate without dynamic compilation, garbage collection, and so on.
Conclusions

Open question: What is the best parallel programming model?