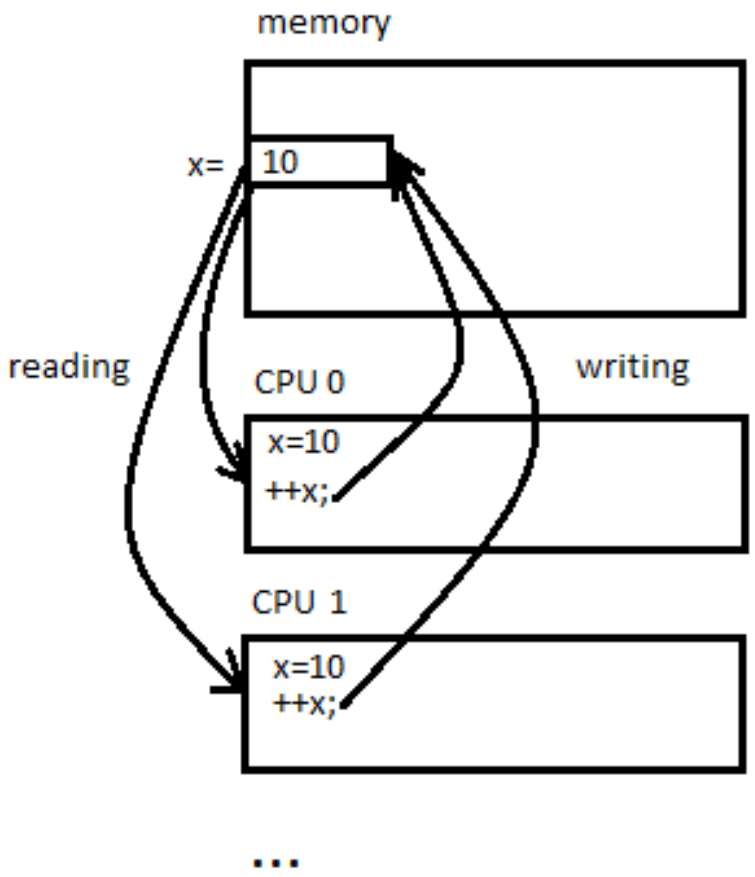
Parallel programming and thread safe data structures

Audrius Pakalniškis
Vilnius University
Lithuania

Tasks

- Goal: testing the potential of transactional memory
  - Getting acquainted with thread based parallelism, using TBB as abstraction.
  - Development of lock based parallel safe data structures.
  - Development of software transactional model based parallel-safe data structures.
  - Performance comparison.

Problem



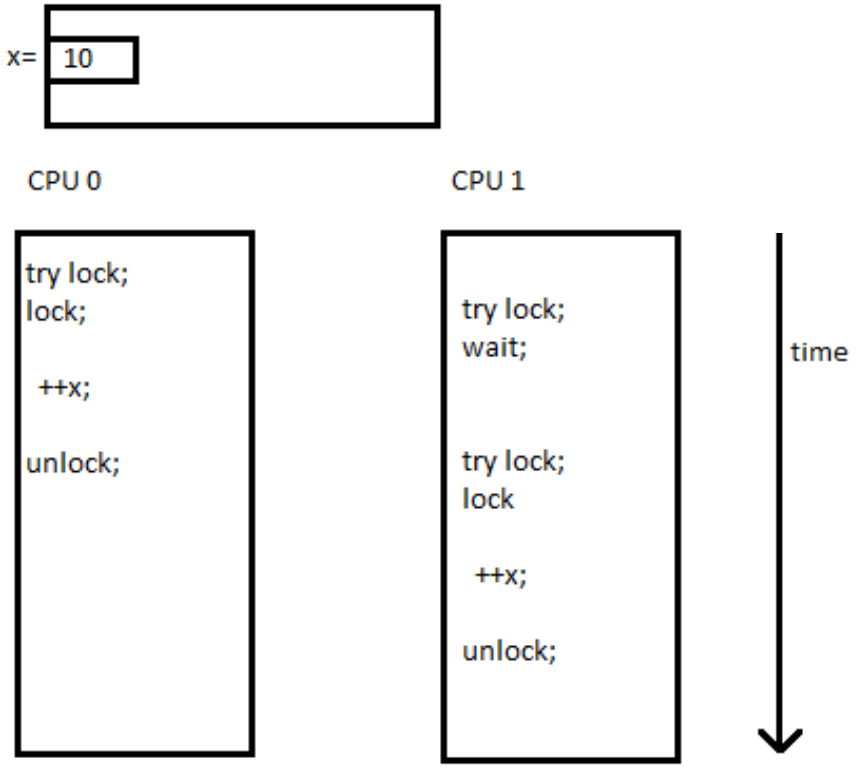
Serial result: x=12;

Parallel result: x=11;

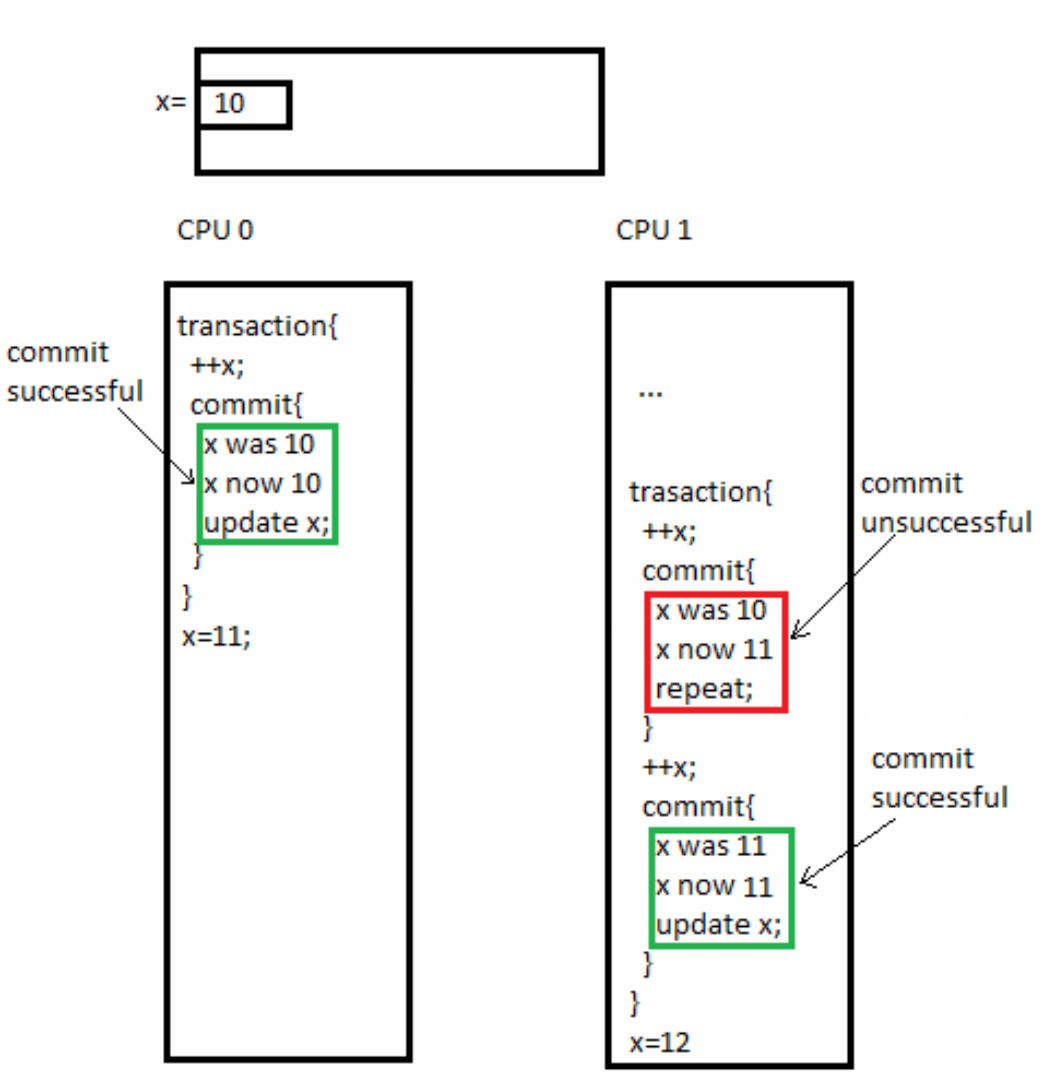
Mutex

- Mutex (mutual exclusion) lock some part of code; only one thread can access it.

```
mutex m;
int i=1;
void f(){
      m.lock();
      ++i;
      m.unlock;
}
```

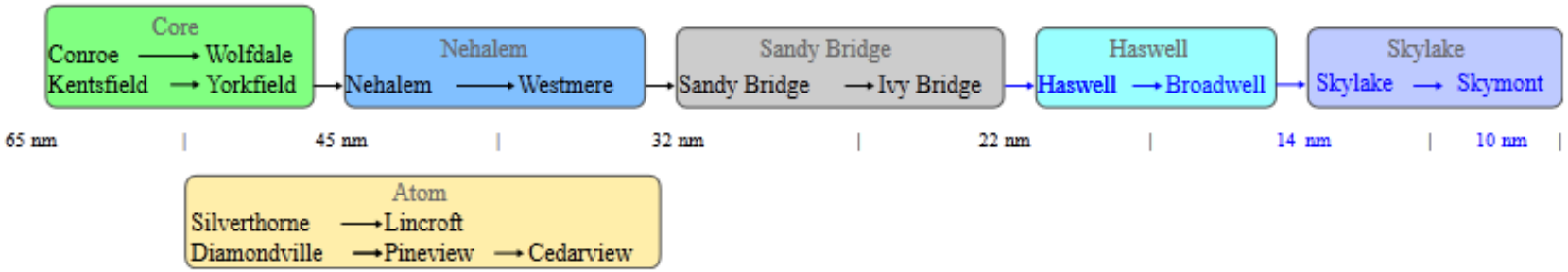


Transactional Memory (TM) (1)



Transaction Memory (TM) (2)

- GCC-4.7 introduced a Software Transactional Memory (STM). It is still experimental and not yet optimized.
- Intel announced hardware support for TM (HTM) in Haswell microarchitecture.



Haswell - future Intel microarchitecture, expected around 2013, based on a 22 nm process.

TM example (1)

```
int a=0;
__attribute__((transaction_safe))
void f()
{
    __transaction_atomic {
        ++a;
    }
}
```

Transaction types

- __transaction_atomic
  - Can’t communicate with other threads and transaction.
- __transaction_relaxed
  - Can communicate with other threads but not with other transaction.

Function attributes

- transaction_safe
- transaction_unsafe
- transaction_callable

TM example (2)

```
mutex m;
void f(int* shared){
        m.lock();
                int temp=*shared;
                if (g(temp)){
                        ++temp;
                        *shared=temp;
                }
        m.unlock();
}
```

```
void f(int* shared){
        __transaction_atomic {
                int temp=*shared;
                if (g(temp)){
                        ++temp;
                        *shared=temp;
                }
        }
}
```

TM example (3)

Using STM in gcc

```
void f(int* shared){
        __transaction_atomic {
                int temp=*shared;
                if (g(temp)){
                        ++temp;
                        *shared=temp;
                }
        }
}
```

Fake TM in C++ code

```
mutex m;
void f(int* shared){
        bool success=false;
        while (success!=true){
                int temp=*shared;
                int oldShared=temp;
                if (g(temp)){
                        ++temp;
                        m.lock();
                        if(oldShared==*shared){
                                *shared=temp;
                                success=true;
                        }
                        m.unlock
                } else {
                        success=true;
                }
        }
}
```

What is inside? (1)

```
void f()
{
    __transaction_atomic {
        ++a;
    }
}
```

```
push   %rbp
mov    %rsp,%rbp
mov    $0x29,%edi
mov    $0x0,%eax
callq  400fd8 <_ITM_beginTransaction@plt>
mov    $0x74c2ec,%edi
callq  4010b8 <_ITM_RU4@plt>
add    $0x1,%eax
mov    %eax,%esi
mov    $0x74c2ec,%edi
callq  400fe8 <_ITM_WU4@plt>
callq  400f48 <_ITM_commitTransaction@plt>
pop    %rbp
retq
```

What is inside? (2)

```
push   %rbp
mov    %rsp,%rbp
mov    $0x29,%edi
mov    $0x0,%eax
callq  400fd8 <_ITM_beginTransaction@plt>
mov    $0x74c2ec,%edi
callq  4010b8 <_ITM_RU4@plt>
add    $0x1,%eax
mov    %eax,%esi
mov    $0x74c2ec,%edi
callq  400fe8 <_ITM_WU4@plt>
callq  400f48 <_ITM_commitTransaction@plt>
pop    %rbp
retq
```

- _ITM_beginTransaction() – save the machine state, initialize transaction data and do other preparation steps.
- _ITM_RU4() – take variable address, checks that memory is not locked or recent (value is taken from global table) and read value.
- _ITM_WU4() – take variable address and value. Marking address location as recent, and keep value.
- _ITM_commitTransaction() – tries to commit, and if it fails restart transaction

Performance

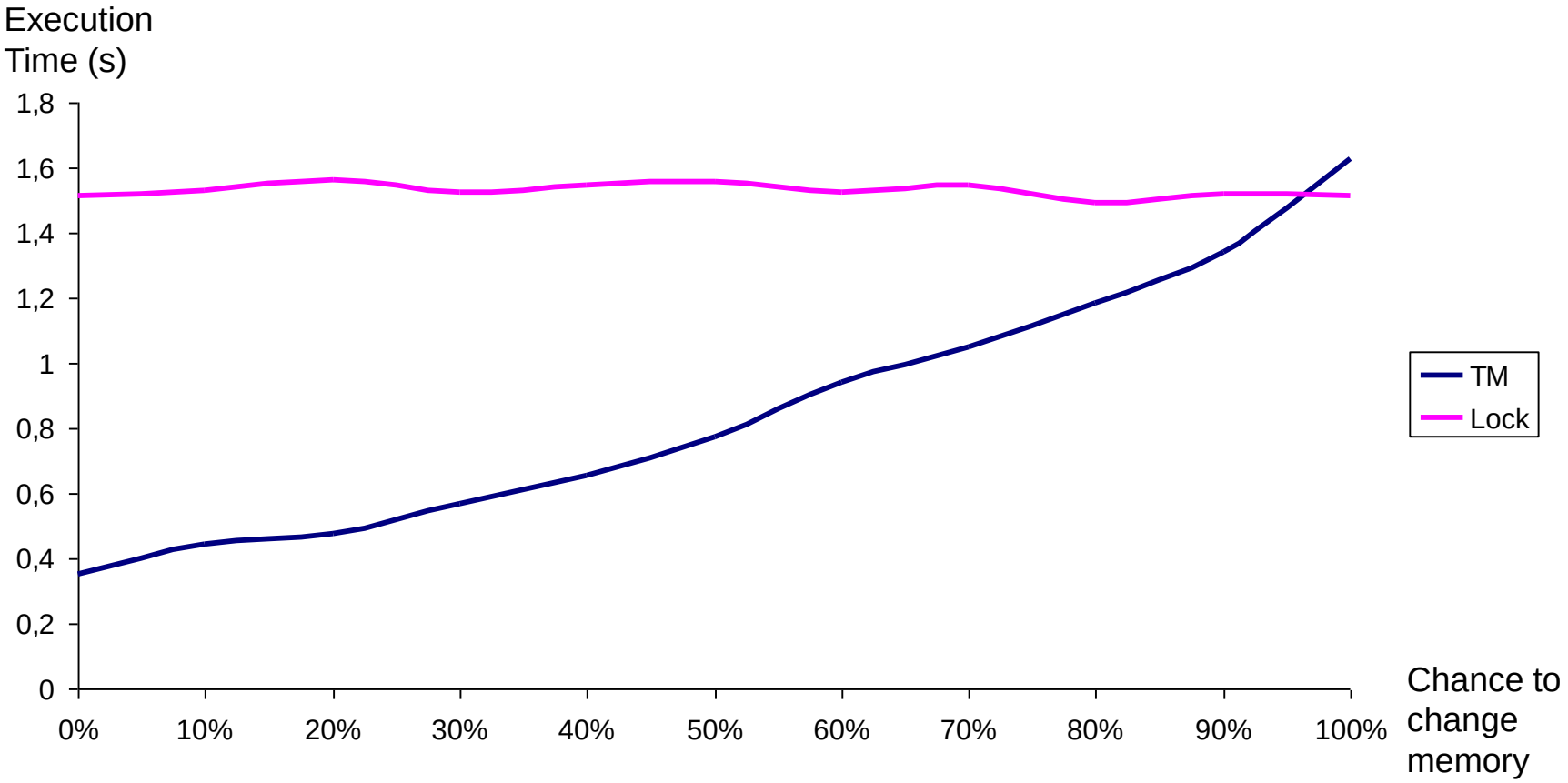
TM performance depends on:

- Collisions count
- Chance to not change memory

Tests

- Comparison TM based and lock based data structures
- TM based Queue implementation and Intel TBB concurrent_queue.

Test of TM based and lock based data structure



Queue (1)

Pushing and popping 8B data

number of popping threads								
8	-261.36%	-488.39%	-562.00%	-573.47%	-652.33%	-680.65%	-716.05%	-798.02%
7	-240.40%	-519.14%	-563.18%	-623.56%	-656.14%	-760.92%	-802.54%	-773.59%
6	-229.74%	-522.36%	-630.09%	-621.91%	-682.20%	-749.78%	-813.48%	-832.11%
5	-299.59%	-579.84%	-641.36%	-660.47%	-729.34%	-811.24%	-793.65%	-802.12%
4	-209.22%	-570.46%	-661.55%	-700.04%	-724.81%	-747.37%	-841.53%	-799.52%
3	-244.18%	-592.57%	-668.33%	-704.29%	-758.90%	-800.64%	-814.80%	-839.49%
2	-266.21%	-604.20%	-735.40%	-780.55%	-833.89%	-750.87%	-897.26%	-805.65%
1	-246.13%	-527.78%	-761.41%	-850.39%	-922.82%	-983.47%	-973.48%	-928.85%
	1	2	3	4	5	6	7	8

number of pushing threads

5% TM faster

-5% TBB faster

Queue (2)

Pushing and popping 2,5KB data

number of popping threads

8	-34.02%	-30.13%	-39.38%	-9.50%	34.43%	45.92%	31.51%	22.62%
7	-38.51%	-34.17%	-39.80%	1.35%	37.88%	42.78%	31.68%	23.79%
6	-26.09%	-24.68%	-32.84%	4.55%	40.99%	38.62%	27.45%	24.87%
5	-18.41%	-30.70%	-45.73%	13.79%	46.54%	25.37%	26.71%	20.46%
4	-50.56%	-36.58%	-28.06%	28.69%	33.40%	20.19%	14.37%	19.07%
3	-67.92%	-37.79%	-10.42%	28.13%	11.18%	11.78%	5.61%	2.33%
2	-71.88%	-72.99%	-40.59%	21.27%	16.41%	5.34%	-0.26%	-3.37%
1	-69.48%	-60.09%	4.07%	-2.82%	-1.93%	-7.91%	2.82%	-10.89%
	1	2	3	4	5	6	7	8

number of pushing threads

5% TM faster

-5% TBB faster

Queue (3)

Pushing and popping 5KB data

number of popping threads								
8	-17.91%	8.63%	10.71%	23.31%	57.43%	61.67%	49.19%	36.30%
7	-27.00%	9.05%	12.35%	33.62%	58.24%	54.64%	44.86%	38.14%
6	-2.62%	7.91%	16.80%	40.82%	56.74%	47.41%	38.52%	31.44%
5	2.79%	-8.26%	12.69%	44.08%	50.79%	36.52%	30.75%	27.03%
4	-6.67%	-9.73%	21.01%	43.04%	37.78%	30.04%	23.97%	17.80%
3	-24.81%	0.90%	13.25%	32.59%	24.12%	17.82%	17.25%	12.15%
2	-66.07%	-30.30%	-0.10%	25.56%	14.00%	8.88%	7.12%	6.00%
1	-44.36%	-10.59%	10.04%	6.79%	10.20%	7.25%	2.49%	0.61%
	1	2	3	4	5	6	7	8

number of pushing threads

5% TM faster

-5% TBB faster

Queue (4)

Pushing and popping 10KB data

number of popping threads

8	-26,01%	11,95%	6,66%	11,59%	53,94%	56,47%	36,14%	27,69%
7	-34,76%	10,13%	14,31%	22,07%	53,65%	36,65%	28,04%	19,73%
6	-26,83%	7,30%	16,17%	25,85%	37,72%	25,68%	20,31%	17,42%
5	-4,79%	11,92%	22,81%	29,34%	19,81%	10,67%	10,33%	9,70%
4	-4,27%	-6,95%	11,57%	15,80%	10,12%	6,26%	3,30%	4,52%
3	-25,08%	-5,42%	1,75%	2,00%	1,54%	-3,44%	-0,41%	-0,73%
2	-48,65%	-39,18%	-10,08%	2,23%	-3,52%	0,66%	-7,81%	-1,36%
1	-30,06%	-7,46%	-1,20%	-2,18%	0,50%	-2,82%	-3,91%	-2,16%
	1	2	3	4	5	6	7	8

number of pushing threads

5% TM faster

-5% TBB faster

Conclusion

- Experimental STM in GCC-4.7 works and gives correct results.
- Transactional memory allows to make parallel safe programming easier.
- Sometimes performance is not as good as expected, so we need to wait for optimizations.
- We expect better performance once there is hardware support.