USE OF TRANSACTIONAL MEMORY TO SIMPLIFY SHARED-MEMORY PARALLEL PROGRAMMING

Serge Preis, Ravi Narayanaswami

Intel Corporation

Parallel programming

- Shared memory gets more popular
  - Multi-core is getting momentum
  - 2-socket workstations for maximum desktop performance
  - Hyper-threading is back
- Shared resources access may be a bottle-neck
  - Locks required to avoid races
  - Single global lock is simple, but limiting
  - Fine-grain locks may provide best scalability
  - Fine-grain locks are hard to design, implement and test
  - Poor implementation is error-prone and limiting
  - Locks are not composable

What is transactional memory

- Syntax:

```
__tm_atomic {
    // transaction code goes here
}
```

- Semantics:
  - Isolation: effects are localized
  - Atomicity: commit or rollback
  - Retry if data conflict is detected
  - Publication and privatization safety
  - Composability via nesting
  - Fine-grain: based on data accesses

Example: Insert Node into a link list

```
{
  new_node->prev = node;
   new_node->next = node->next;
   node->next->prev = new_node;
   node->next = new_node;
}
```

Example: Insert Node into a link list

Thread 1 **Thread 2**

```
{
  new_node->prev = node;
  new_node->next = node->next;
  node->next->prev = new_node;
  node->next = new_node;
}
```

Example: Insert Node into a link list

Single global lock

Thread 1

get lock and execute

```
Lock {
   new_node->prev = node;
   new_node->next = node->next;
   node->next->prev = new_node;
   node->next = new_node;
}
```

Thread 2

waits for lock to be released

```
Lock {
   new_node->prev = node;
   new_node->next = node->next;
   node->next->prev = new_node;
   node->next = new_node;
}
```

Example: Insert Node into a link list

TM Based

Thread 1

```
__tm_atomic {
   new_node->prev = node;
   new_node->next = node->next;
   node->next->prev = new_node;
   node->next = new_node;
}
```

Thread 2

```
__tm_atomic {
   new_node->prev = node;
   new_node->next = node->next;
   node->next->prev = new_node;
   node->next = new_node;
}
```

Both threads execute in parallel, if they write to same node then abort and retry one of transactions

Transactional memory overview

- Fine-grain concurrency management without locks
  - Concurrent readers are welcome
  - Re-execute entire transaction if conflict is detected
- Simple syntax and semantics
  - Looks and behaves like single global lock
  - Simpler create race-free programs
- Possible implementations
  - Purely software (STM)
  - Software with HW acceleration (HaSTM)
  - Separate HW and SW (HyTM)
  - HW-based with TX size restrictions (RTM)
  - HW-based for short transactions, software for unbounded (VTM)
- No TM hardware is out yet
  - SUN* ROCK* planned to be released next year
  - AMD* has published spec for TM H/W assistance

Software transactional memory

- Weak atomicity: guarantees are only for transactional code
  - Same as for locks
- Unbounded: transactions of arbitrary size are supported
  - HW resources (memory) is the limit
- Instrumentation of memory accesses is required within transactions
  - Spatial and performance overhead
  - Including called functions
  - Not always possible
  - Different data and contention management techniques
  - Object based or word-based depending on language

Intel STM implementation

- C/C++ Compiler + run-time library
- Word-based STM system
- Close nesting
- Failure atomicity (__tm_abort)
- Irrevocable execution support for I/O and legacy
- Support for C++ constructs
- Highly optimized

Intel STM compiler

- Language extensions
  - Basic constructs
  - Functions and classes annotation
  - Failure atomicity
- Instrumentation to library calls
  - Transaction boundaries
  - Memory accesses
  - Function calls
- Integration with existing language constructs
  - Calls direct and indirect, OpenMP
  - C++: EH, OOP, templates
- Optimizations
  - Optimized memory instrumentation
  - Annotations propagation
  - TM mode based on transaction properties

Intel STM library

- Comprehensive and flexible ABI
  - Supports various STM implementations
  - Allows compiler optimizations
- Effective contention management with switchable strategies
  - In-place updates
  - 2 dynamically interchangeable strategies
  - Effective implementation
  - Additional obstinate mode for long transactions
- Irrevocable execution support
- Failure atomicity support for local data
- Nesting support with local aborts
- Special handling for memory allocation and copying

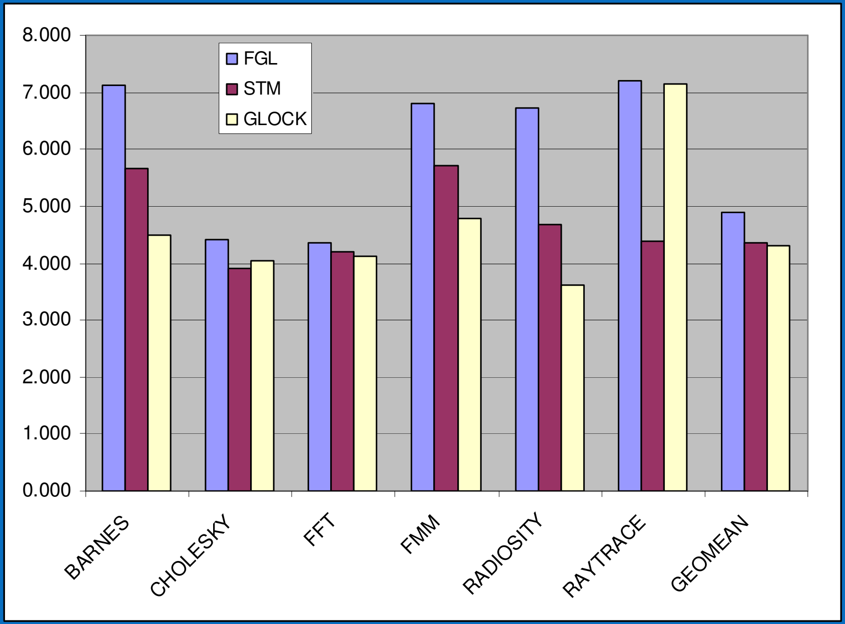
Example

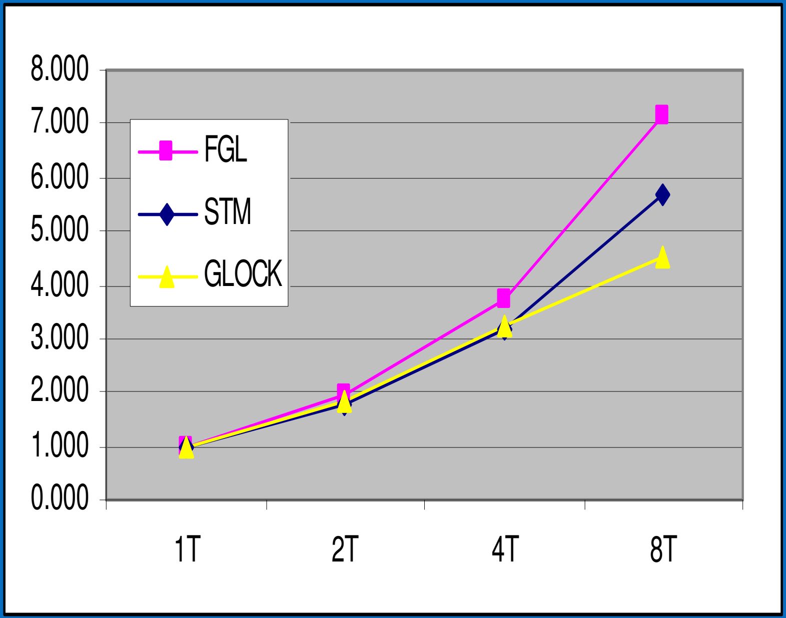
```
__declcpec(tm_callable)
foo();


//...
__tm_atomic {
   foo(++c);
}
```

```
Label:
   action = StartTX();
   if (action & restoreLiveVariables)
     liveX = saved_liveX;
   if (action & retryTransaction)
     goto Label;
   else if (action & saveLiveVariables)
        saved_liveX = liveX;
   if (action & InstrumentedCode) {
      temp = ReadInt(c);
      temp = temp + 1;
      WriteInt(c);
      foo_$TXN(c); // TM-version of foo
   } else {
      c = c + 1;
      foo(c);
   }
   CommitTX();
```

Performance data (SPLASH2 benchmarks)





Speedup vs. serial execution on 8T

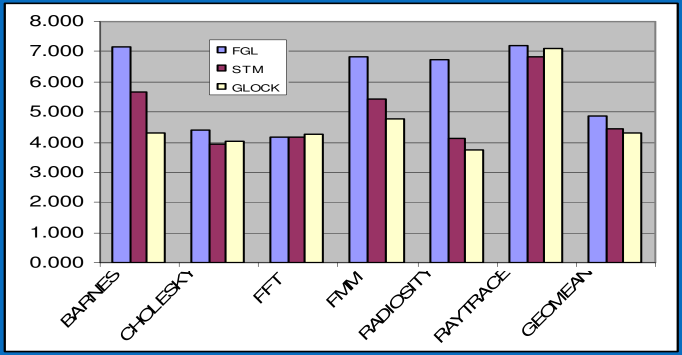
Scalability of BARNESS benchmark

Performance of STM depends highly on workload

Software and services group

Performance analysis

- For many workloads STM outperforms Single Global Lock
  - On 8 or more threads
  - Results still lower than Fine Grain Locks
  - More optimizations and improvements are on the way
- RAYTRACE is and example where ceiling is hit
  - When all 8 threads are running the HW is 100% busy and thus STM code increases the machine load beyond the limit and performance drops
  - Picture is quite different for 8T on 16-way HW, but for 16T is the same
  - Optimization to run short transactions in Global Lock mode helps much to this benchmark
- Optimizations are not good for all benchmarks: RADIOSYTY performs best in pure STM runs



Small transactions optimization is ON

Conclusion

- Pros
  - Simple programming model
  - Composability
  - Failure atomicity
  - Decent scalability at 8 threads and beyond for many workloads
  - Popular paradigm in modern HPC languages
    - Fortress* (SUN*), X10* (IBM*), Chapel* (Cray Inc.*)
- Cons
  - Overhead
  - Profitability highly depend on workload
  - Retries eat power
- Intel C/C++ STM compiler prototype edition is publicly available at http://whaif.intel.com
  - Includes Intel STM library
  - Documentation is also available
  - Active discussion forum for questions and comments

Q&A