The OpenTM Transactional Application Programming Interface

Woongki Baek, Chi Cao Minh, Martin Trautmann,
Christos Kozyrakis, Kunle Olukotun

Computer Systems Laboratory
Stanford University
http://tcc.stanford.edu

Motivation

- Transactional Memory (TM)
  - Simplifies parallel programming using large atomic blocks
  - Performance of fine-grain locks; simplicity of coarse-grain locks
- Current practice: TM programming with library-based APIs
  - Error-prone and difficult to maintain, port, and scale
  - Reduces effectiveness of compiler optimizations
- Needed: a high-level approach for TM programming
  - Integrated with constructs that define parallel work
  - Compiler support & optimizations
  - Portable code across multiple TM platforms

OpenTM Contributions

- OpenTM = OpenMP + TM
  - Unified model for expressing parallelism & memory transactions
    - Familiar & simple environment for high-level programming
  - TM uses: non-blocking sync, speculative parallelization
    - Extends shared-memory programming model of OpenMP
- Compiler-based OpenTM implementation
  - Based on the GCC + Gnu OpenMP (GOMP) framework
  - Retargetable to hardware, software, and hybrid TM systems
    - Automatic annotation of memory accesses + optimizations
  - Runtime system for scheduling + contention management
- Initial evaluation of performance, programmability, and runtime
  - OpenTM code is simple, compact, and scales well

Outline

- Motivation
- OpenMP Overview
- The OpenTM API
- A First OpenTM Implementation
- Evaluation
- Related Work
- Conclusions

OpenMP Parallel Model

- A widely-used API for shared-memory parallel programming
  - Consists of a set of compiler directives + runtime library

- Based on the fork-join parallel execution model
  - Master thread executes sequential code
  - Worker threads execute parallel regions
    - Parallel loops and parallel sections

- Five classes of directives & routines
  - Parallel: **`parallel`**
  - Work-sharing: **`for`**, **`sections`**, etc.
  - Synchronization: **`critical`**, **`atomic`**, **`barrier`**, etc.
  - Data environment: **`private`**, **`shared`**, etc.
  - Runtime: **`omp_set_num_threads`**, etc.

OpenMP Parallel Constructs

Parallel Loop

```
#pragma omp parallel for
  for (i=1; i<n; i++) {
    b[i]=(a[i]+a[i-1])/2.0;
  }
```

Parallel Section

```
#pragma omp parallel sections
{
  #pragma omp section
    XAXIS();
  #pragma omp section
    YAXIS();
  #pragma omp section
    ZAXIS();
}
```

Source: OpenMP API Ver. 2.5

OpenTM Transactional Model

- Implicit transactions
  - User specifies only xaction boundaries
    - No need for manual instrumentation of accesses within xaction
  - All xaction accesses implicitly operate on transactional state
    - If needed, instrumentation inserted by the compiler
- Strong isolation
  - Xactions are isolated from non-transactional accesses
  - Necessary for correct and predictable program behavior
  - Enforced by underlying TM system or by the compiler
- Virtualized transactions
  - Xactions not bounded by time, memory footprint, or nesting depth

OpenTM Transactions

- Defines boundaries of a strongly isolated transaction
  - Can be used within parallel regions of OpenMP

- Syntax: #pragma omp transaction [clauses] {structured-block}
  - Ordered clause: requires sequential commit order for xactions
  - Otherwise, commit order is serializable but not predefined

- Code example

```
#pragma omp parallel for
  for (i=0; i<N; i++) {
    #pragma omp transaction
    { bin[A[i]] = bin[A[i]] + 1; }
  }
```

OpenTM Transactional Loop

- Defines parallel loop with iterations executing as xactions
- Syntax: #pragma omp transfor [clauses]
  - Ordered clause: require sequential commit order for xactions
    - Ordered loop ⇒ speculative parallelization (TLS)
    - Unordered loop ⇒ parallel loop with non-blocking synchronization
  - Schedule clause (see syntax in paper)
    - Scheduling policy, loop chunk size, transaction size
  - Other clauses: private variables, shared variables, …
- Code example

```
#pragma omp transfor schedule (static, 42, 6)
   for (i=0; i<N; i++) {
      bin[A[i]] = bin[A[i]]+1;
   }
```

OpenTM Transactional Sections

- Defines parallel sections executing as xactions
- Syntax:

  #pragma omp transsections [clauses]

  [#pragma omp transsection {structured-block}]+

  - Ordered clause: require sequential commit order for xactions
    - Ordered loop ⇒ speculative parallelization (TLS)
    - Unordered loop ⇒ parallel section with non-blocking synchronization

- Code example (method-level speculation)

```
#pragma omp transsections ordered
    #pragma omp transsection
        WORK_A();
    #pragma omp transsection
        WORK_B();
```

Advanced Constructs (Summary)

- Conditional synchronization
  - omp_watch(): notifies runtime to monitor an address
  - omp_retry(): indicates xaction is blocked on a condition
    - Runtime system decides retry immediately or suspend thread
- Alternative execution path
  - #pragma omp orelse: alternative code runs if xaction aborts
- Transactional handlers
  - Software handlers invoked on commit, abort, or conflict
  - Associated with transaction, transfor, transsections, or orelse
- Nested transactions
  - Support for both open and closed nested xactions

Open Issues & Requirements

- Philosophy: define an intuitive first set of features for OpenTM
  - Evolve model after receiving feedback from users
- Currently required
  - User must mark functions that may be used within xactions
    - Necessary for code generation for software & hybrid TM systems
- Currently disallowed
  - Nesting of xactions and OpenMP synchronization
    - Can lead to various deadlock or livelock scenarios
  - I/O and system calls within transactions
  - Nested parallelism within transactions
- Future language considerations
  - Relaxed conflict detection (e.g., race or exclude variables)
    - May improve performance but can also lead to bugs

OpenTM Runtime System

- Scheduling of loop iterations across worker threads
  - Reuse of OpenMP options (static, guided, dynamic)
  - Extended to handle the number of iterations per xaction
    - Default is 1 but can change statically or dynamically
    - Balance xaction overhead vs. frequency of conflicts

- Contention management for conflicting xactions
  - Necessary for performance robustness and fairness
  - OpenTM runtime controls the policy of underlying TM system
    - omp_get_cm(): query current contention management policy
    - omp_set_cm(): set current contention management policy
  - Policies and parameters are an open research issue

Outline

- Motivation
- OpenMP Overview
- The OpenTM API
- A First OpenTM Implementation
- Evaluation
- Related Work
- Conclusions

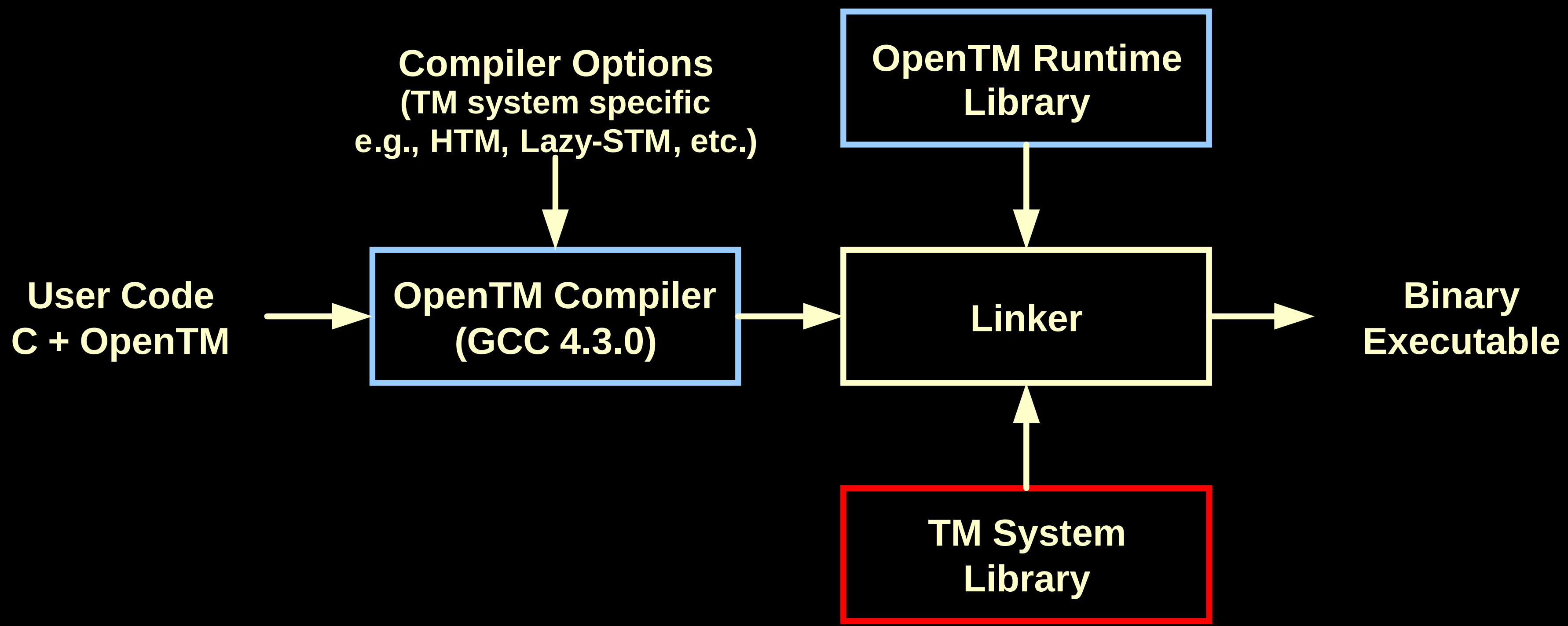
Implementation Approaches

- Source-to-source translation
  - OpenTM $\Rightarrow$ C with library calls $\Rightarrow$ executable
  - Pros: simple to prototype
  - Cons: debugging intermediate code, lack of optimizations
  - Our initial OpenTM system followed this approach
    - Using the Cetus source-to-source framework
- Compiler-based system
  - OpenTM $\Rightarrow$ executable
  - Pros: high-level debugging, full compiler optimizations
  - Cons: compiler complexity
  - Our current OpenTM system follows this approach
    - Based on GCC + GOMP to maximize reuse and portability

Our OpenTM Implementation

- Compiler
  - GCC 4.3.0 + Gnu OpenMP (GOMP) environment
  - Modified parser, IR, and code generator
  - Currently working on code optimizations for TM
- Interface to underlying TM system
  - Defined a simple API to interface code with TM system
    - Supports hardware, software, and hybrid TM systems
    - Supports both lazy and eager systems for STM
  - Compiler can easily retarget to any TM system that follows this API
- OpenTM runtime system
  - A set of runtime library routines for OpenMP and OpenTM
  - Simple conditional synchronization (immediate retry)
    - Currently working on optimized runtime system

OpenTM Code Generation



Evaluation Methodology

- Three TM systems on top of simulated x86-based CMP
  - Hardware TM (similar to Stanford's TCC)
  - Software TM system (Sun's TL2)
  - Hybrid TM system (similar to Stanford's SigTM)

- Applications
  - Four applications: delaunay, genome, kmeans, vacation
  - One microbenchmark: histogram

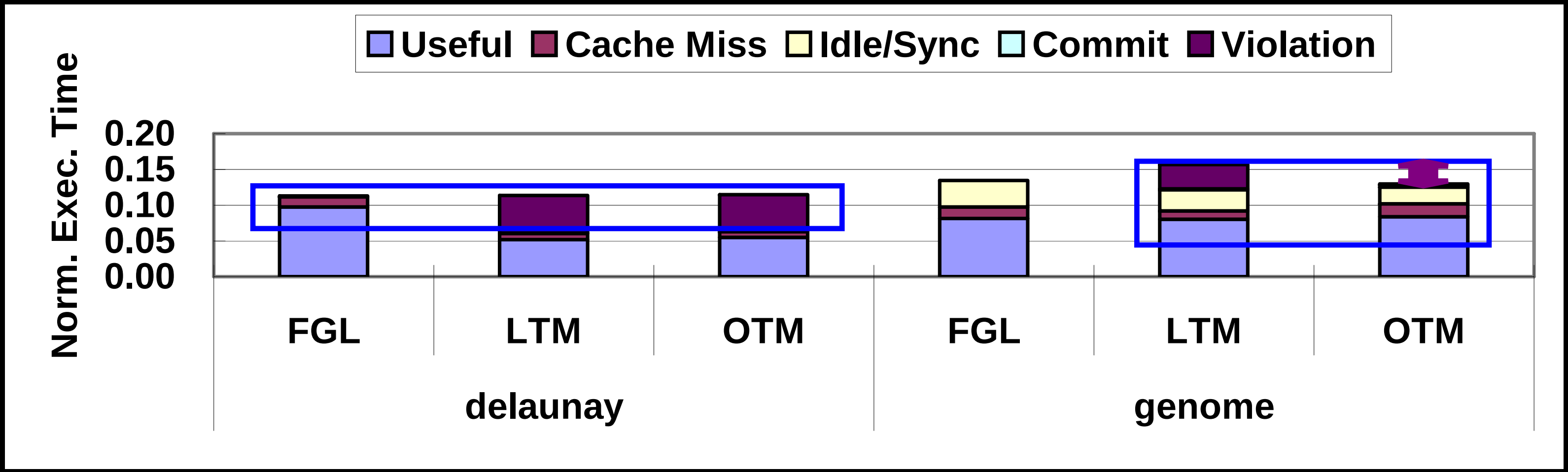
- Code versions
  - OpenTM code (OTM)
    - Automatic generation of binaries for HTM, STM, and hybrid TM
  - Low-level code that uses directly the TM API (LTM)
  - Parallel code with coarse-grain (CGL) and fine-grain (FGL) locks

Programmability

App.	File	# of extra C lines			
		FGL	LTM-HTM	LTM-STM	OTM
delaunay	cavity.c	43	0	0	0
genome	sequencer.c	25	32	58	11
vacation	rbtree.c	11	0	105	0

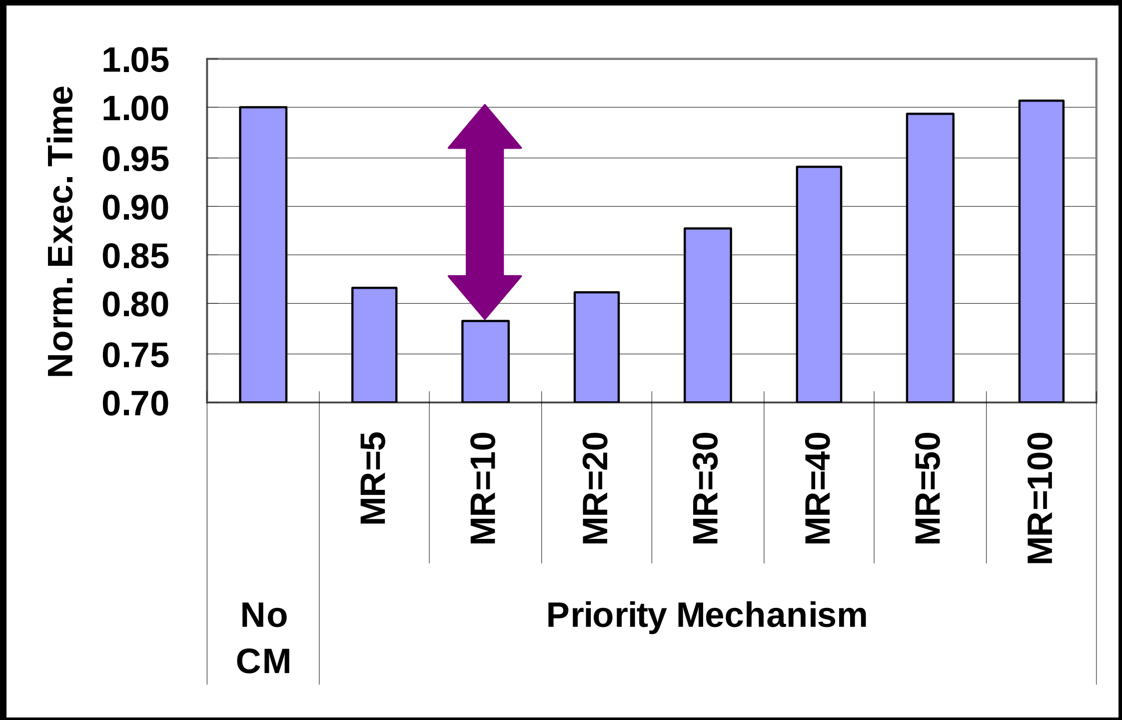
- vs. FGL
  - Manual orchestration to shared states
- vs. LTM-STM
  - Manual instrumentation for all load/store within xactions
  - Highly error-prone (missing barrier) or low-performance (redundant barrier)
- vs. LTM-HTM
  - Significant code transformation for parallelization & loop scheduling

Performance Comparison



- vs. FGL
  - Compares favorably
  - delaunay: FGL code is marginally faster by avoiding overhead of aborted Tx's
- vs. LTM
  - Compares favorably
  - genome: OpenTM code is faster with easy tuning (scheduling policy/Tx's size)

Runtime System



- Contention management
  - Handle Starving Elder (SE) pathology using a simple priority mechanism
  - After MR (max. retry) aborts, give high priority to the aborted xaction
  - Tradeoff: starvation vs. serialization

Related Work

- TM programming for unmanaged code (C/C++)
  - [Wang'07]: no work sharing constructs; targets STM only
  - [von Praun'07]: supports only ordered xactions
  - [Milovanonic'07]: defines transaction construct for OpenMP
    - Lacks several advanced features & compiler-based implementation
  - [Felber'07]: no work sharing constructs; targets STM only
- TM programming for managed code (Java/C#)
  - [Ald-Tabatabai'06]: compiler optimizations for STM
  - [Haris'06]: compiler optimizations for STM
  - [Carlstrom'06]: conditional synchronization using TM

Conclusions

- OpenTM = OpenMP + TM
  - Unified model for expressing parallelism & memory transactions
  - Compiler-based system for optimizations and portability
  - Runtime system for dynamic scheduling and contention management
  - Good performance with simple and portable high-level code

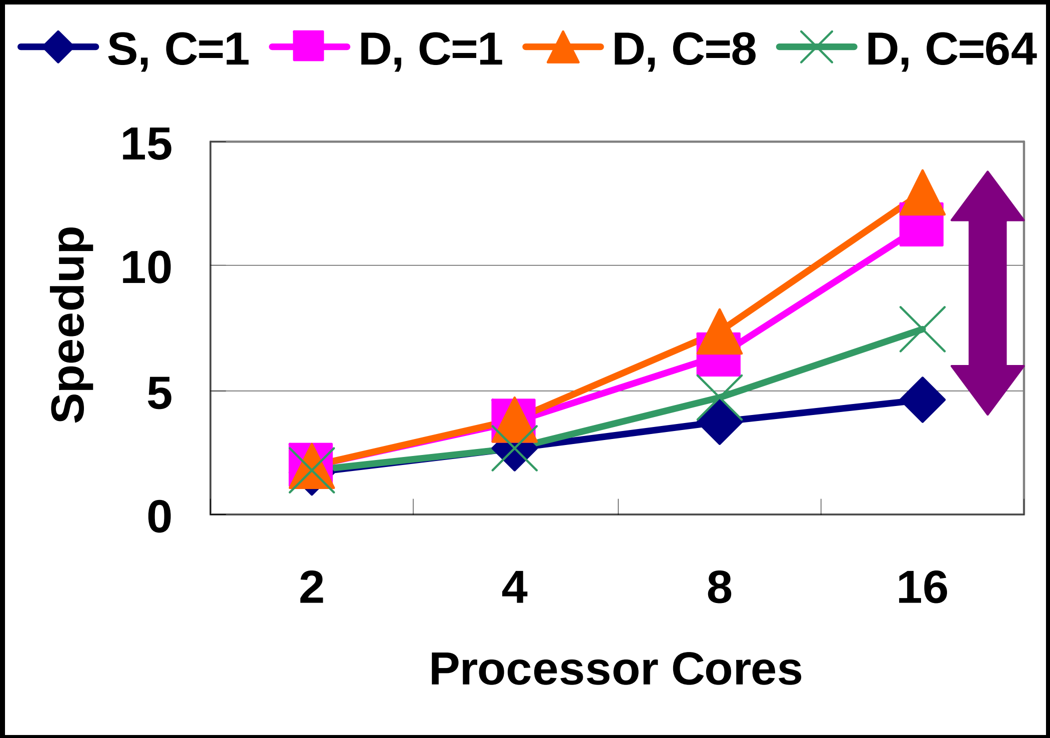
- Future work
  - Open-source our OpenTM environment
    - Compiler and runtime
  - Compiler optimizations
    - Primarily for software and hybrid TM systems
  - Further language and runtime features

Thanks & Questions?

Woongki Baek wkbaek@stanford.edu

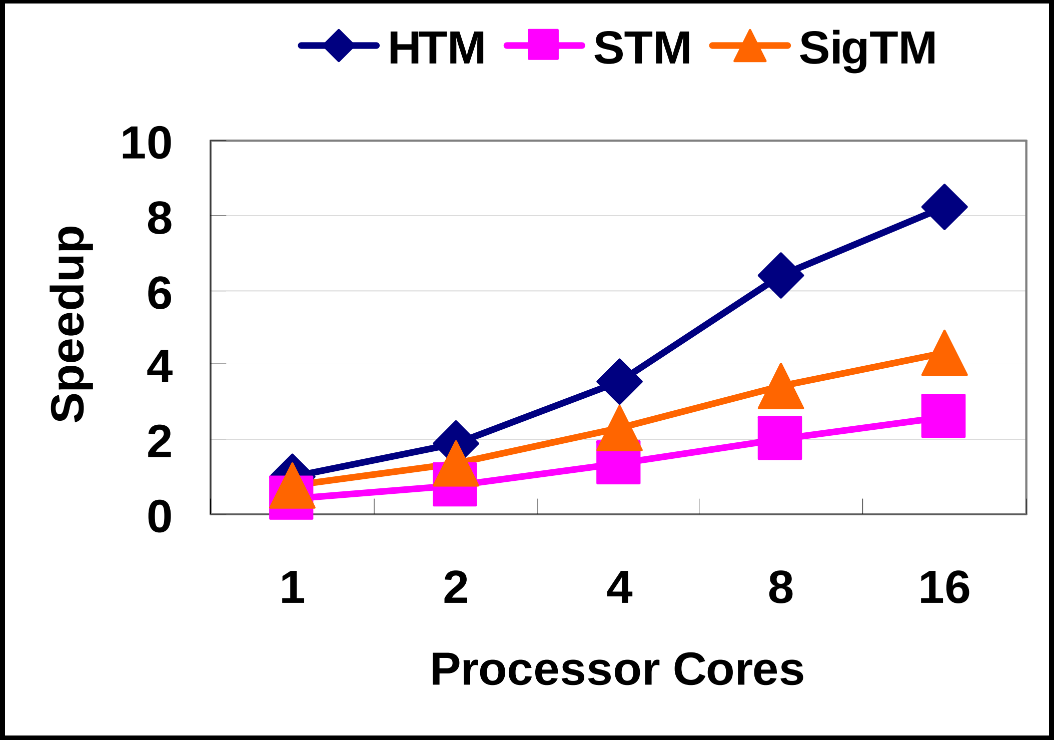
Backup Slides

Runtime System



- Dynamic scheduling
  - Smaller vs. larger chunk size
  - Less imbalance & violations vs. more scheduling overhead

Code Generation



- OpenTM code: retargetable to hardware, software, and hybrid TM system
- Performance comparison
  - HTM: fastest, SigTM: 2x faster than STM (see our ISCA'07 paper for details)
- More aggressive compiler optimizations: in progress