System Challenges and Opportunities for Transactional Memory

JaeWoong Chung

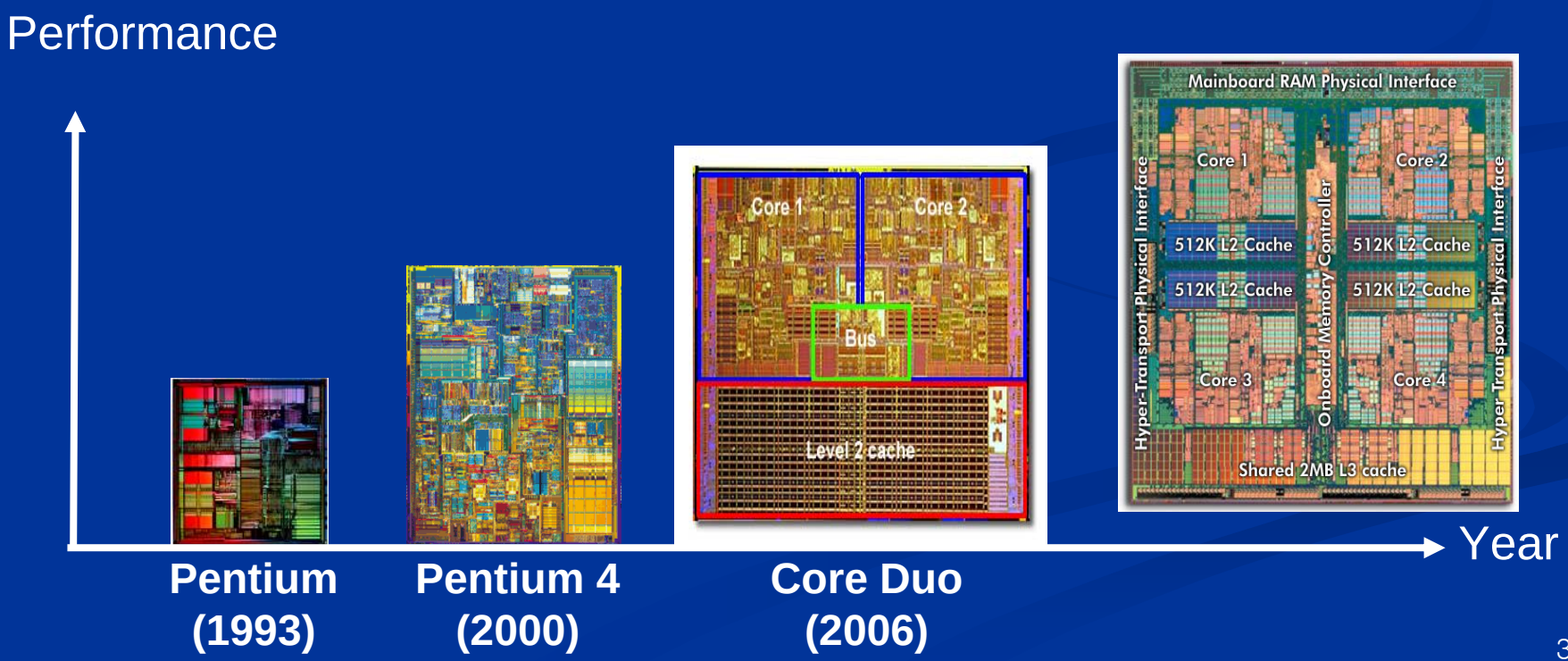
Computer System Lab
Stanford University

My thesis is about

- Computer system design that help leveraging hardware parallelism
- Transactional memory (TM) for easy parallel programming
- Contribution
  - Challenges to building an efficient and practical TM system
  - Opportunities to use TM beyond parallel programming

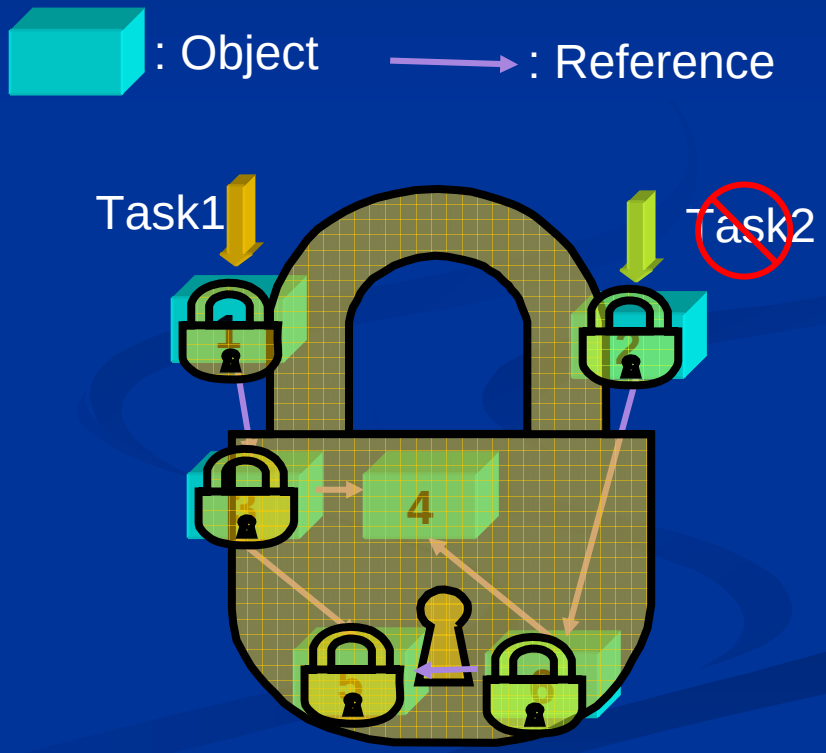
Multi Core Processors

- No more frequency race
  - The era of multi cores
- Parallel programming is not easy
  - Split a sequential task into multiple sub tasks



Locking is hard to use

- Synchronize access to shared data
- Coarse-grain locking
  - Easy to program
  - The other task is blocked
- Fine-grain locking
  - High concurrency
  - Hard to use
  - Dead lock, priority inversion, …
  - High locking overhead



Object reference graph (e.g. Java and C++)

Transactional Memory

Transactional Memory

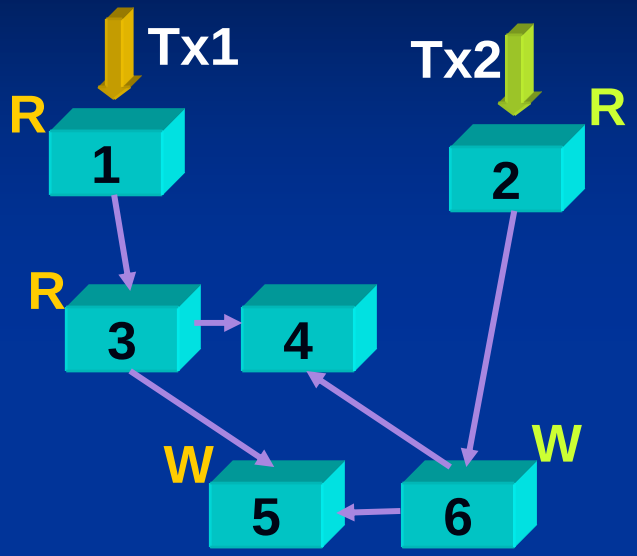
- Atomic and isolated execution of instructions
  - Atomicity : All or nothing
  - Isolation : No intermediate results
- Programmer
  - A transaction encloses instructions
  - logically sequential execution of transactions

```
TX_Begin
// Instructions
// for Task1
TX_End
```

```
TX_Begin
// Instructions
// for Task2
TX_End
```

- TM system
  - Transactions are executed in parallel without conflict
  - If conflict, one of them is aborted and restarts

TM Example



Tx 1 : $R_{[1\ 3]}$ $W_{[5]}$

Tx 2 : $R_{[2\ ]}$ $W_{[6]}$

- Data versioning
  - At TX_Begin, save register values
  - At write, save old memory values
- Conflict detection
  - Read-set and write-set per transaction
  - Conflict detection with set comparison

TM Benefits

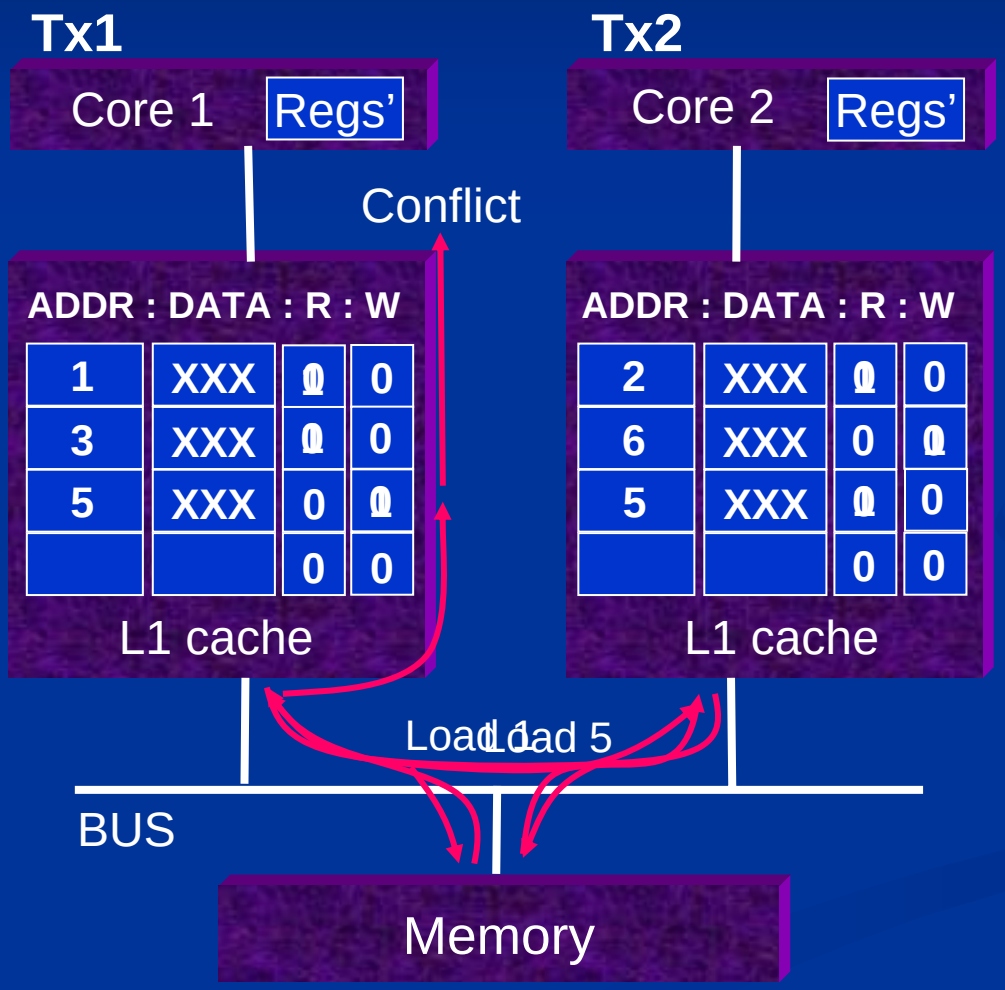
- Logically sequential execution of transactions
- Optimistic concurrency control for parallel transaction execution
- No dead lock, priority inversion, and convoying
  - TM system handles pathological cases
- Composability
- Error Recovery

TM System Design

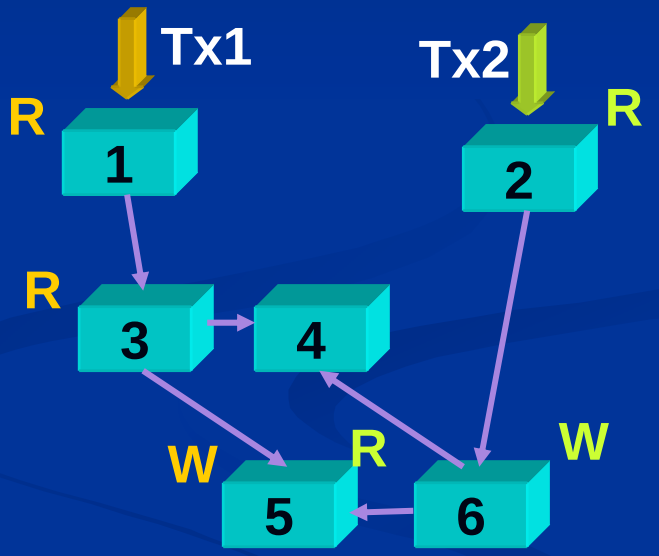
- Many proposals in hardware and software
  - Hardware acceleration for TM is crucial for performance
    - HTM is 2 ~3 times faster than STM
  - Correctness : strong isolation
- Hardware TM
  - In the beginning
    - register checkpoint
  - At memory access
    - Set read/write bits per cache line
    - Buffer new values in cache or log old values
  - Conflict detection
    - With cache coherence protocol
    - With transaction validation protocol

Hardware TM Example

- TM hardware



- TM programs



Challenges and Opportunities

- How to build efficient TM system tuned for common case?
- How to build practical TM system to deal with uncommon case?
- Can we use TM to support system software?
- Can we use TM to improve other important system metrics?

Contributions

- Challenges to building TM systems
  - Common case behavior of parallel programs
    - Extract architectural parameters for efficient TM system design
  - TM virtualization
    - Overcome the limitation of TM hardware
- Opportunity for system beyond parallel programming
  - Multithreading for dynamic binary translation
    - Guarantee correctness of DBT
  - Support for reliability, security, and fast memory snapshot
    - Improve important system metrics other than performance

Outline

- Software parallelization : a major issue for performance
- Transactional memory
- Challenges to building TM systems
  - Common case behavior of parallel programs
  - TM virtualization
- Opportunities for systems beyond parallel programming
  - Multithreading for dynamic binary translation
  - Support for reliability, security, and fast memory snapshot
- Conclusion

Challenges to Building TM Systems

Challenge 1 : Common Case Behavior of Parallel Programs

- Goal
  - Understand the common case behavior of TM programs
- Few TM programs available
  - More TM programs now but for research purpose
- Few efficient TM systems as development tool
- “chicken & egg problem”

Inferring Transactions in Multithread Programs

- Analyze existing parallel programs
  - Assumption : the inherent parallelism remains regardless of programming tools

- Mapping programming primitives to transactions

Programming primitive	Transaction primitive
Lock/Unlock	Begin/End
Parallel_For	Begin/End

Parallel Applications

- Different domains and different language

Languages	Applications
Java	MolDyn, MonteCarlo, RayTracer, Crypt, LUFact, Series, SOR, SparseMatmult, SPECjbb2000, PMD, HSQLDB
Pthread	Apache, Kingate, Bp-vision, Localize, Ultra Tic Tac Toe, MPEG2, AOL Server
ANL	Barnes, Mp3d, Ocean, Radix, FMM, Cholesky, Radiosity, FFT, Volrend, Water-N2, Water-Spatial
OpenMP	APPLU, Equake, Art, Swim, CG, BT, IS

Key Metrics of TM Programs

- Measure the key metrics of TM programs
  - Use the metrics to make suggestions for TM designs

Key metrics	Architectural parameters
Transaction length	TM primitive overhead
Read-/write-set size	Buffer size
Write-set to length ratio	Transaction commit/abort overhead
Frequency of nesting & I/O in transactions	Support for nesting and system calls

Transaction Length

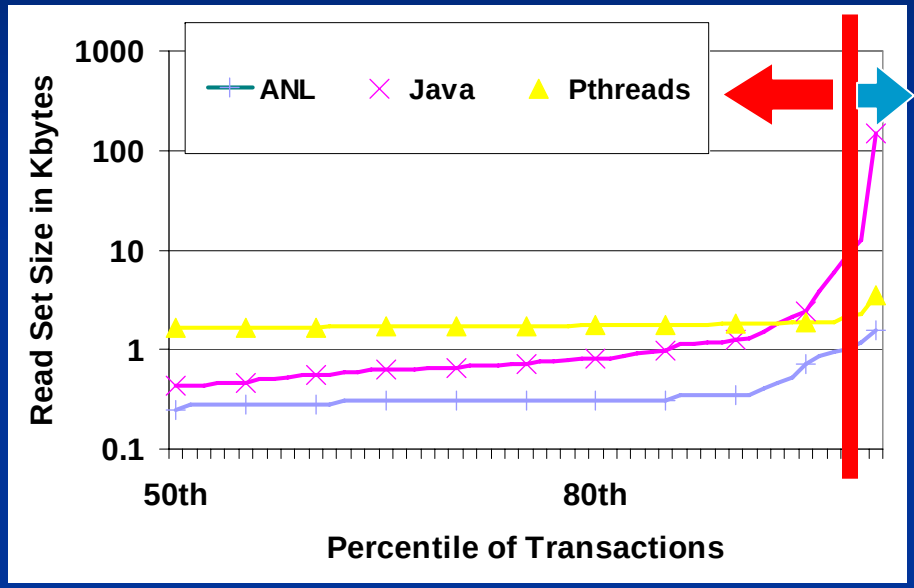
- Number of instructions executed in transaction

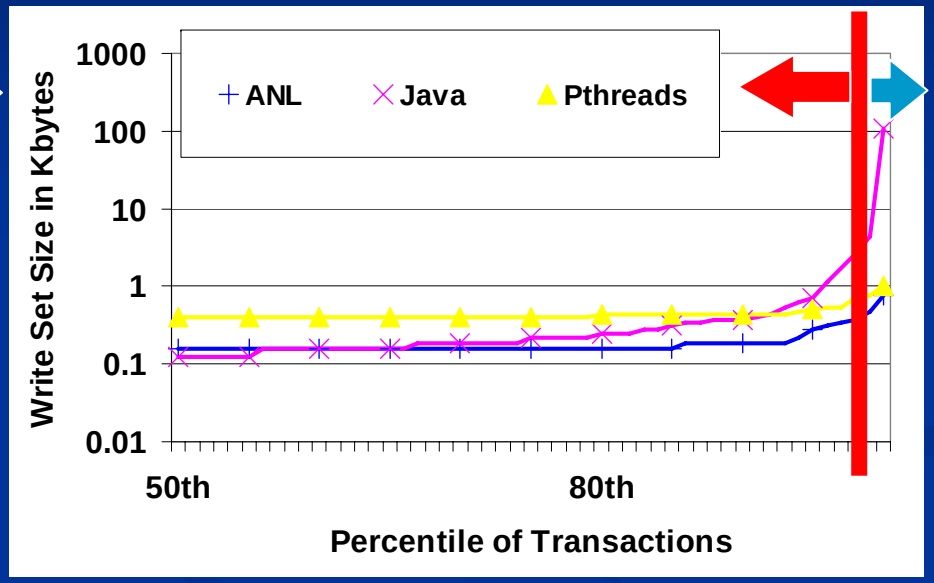
Application	Length in Instructions			
	Avg	50th %	95th %	Max
Java average	5949	149	4256	13519488
Pthreads average	879	805	1056	22591
ANL average	256	114	772	16782

- Observation : Up to 95% of transactions < 5000 instructions
  - Suggestion : Light-weight transactional primitives
- Observation : Rare but long transactions
  - Suggesion : Transaction over context-switching

Read-/Write-Set Size

- Bytes of data read/written by transaction





- Observation : 98% transactions <16KB read-set and <6KB write set
  - Suggestion : 32K L1 cache is sufficient
- Observation : Few very large transaction > 32K
  - Suggestion : Need for buffer space virtualization

Outline

- Software parallelization : a major issue for performance
- Transactional memory
- Challenges to building TM systems
  - Common case behavior of parallel programs
  - TM virtualization
- Opportunities for systems beyond parallel programming
  - Multithreading for dynamic binary translation
  - Support for reliability, security, and fast memory snapshot
- Conclusion

Challenge 2 : TM Virtualization

- Problem
  - Limited hardware resources tuned for common cases
    - E.g. buffer size for 99% transactions
  - How do we cover uncommon cases as well?
- Cache as buffer for transactional data
  - What if cache capacity is exhausted? => space virtualization
- What if a transaction is interrupted?
  - Time virtualization
- What if transactions are nested deeply?
  - Depth virtualization

XTM: eXtended TM

- Goals
  - Virtualize TM space, time, and depth at low HW cost
  - Completely transparent to user SW
  - Minimize interference with coexisting HW transactions
- Assumption
  - Overflows, interrupts, and deep nesting are rare
- Approach
  - Transactional data and metadata in virtual memory
  - Using virtual memory support in OS
  - Data versioning & conflict detection at page granularity
  - Similar to page-based software DSM systems

XTM Overview

- Basic operation
  - On HTM overflow, rollback and restart in SW mode
  - At the first access, create a copy of original (master) page
    - Change the address mapping to the copy (private page)
    - Transactional data in private page, committed data in master page
  - At commit, make the private page the new master page
  - All orchestrated by the operating system (no HW)
- Conflict detection
  - Use TLB shoot-downs to gain exclusive page access
- Hardware requirement
  - Overflow exception

XTM Example

Timeline

↓

HTM Overflow

↓

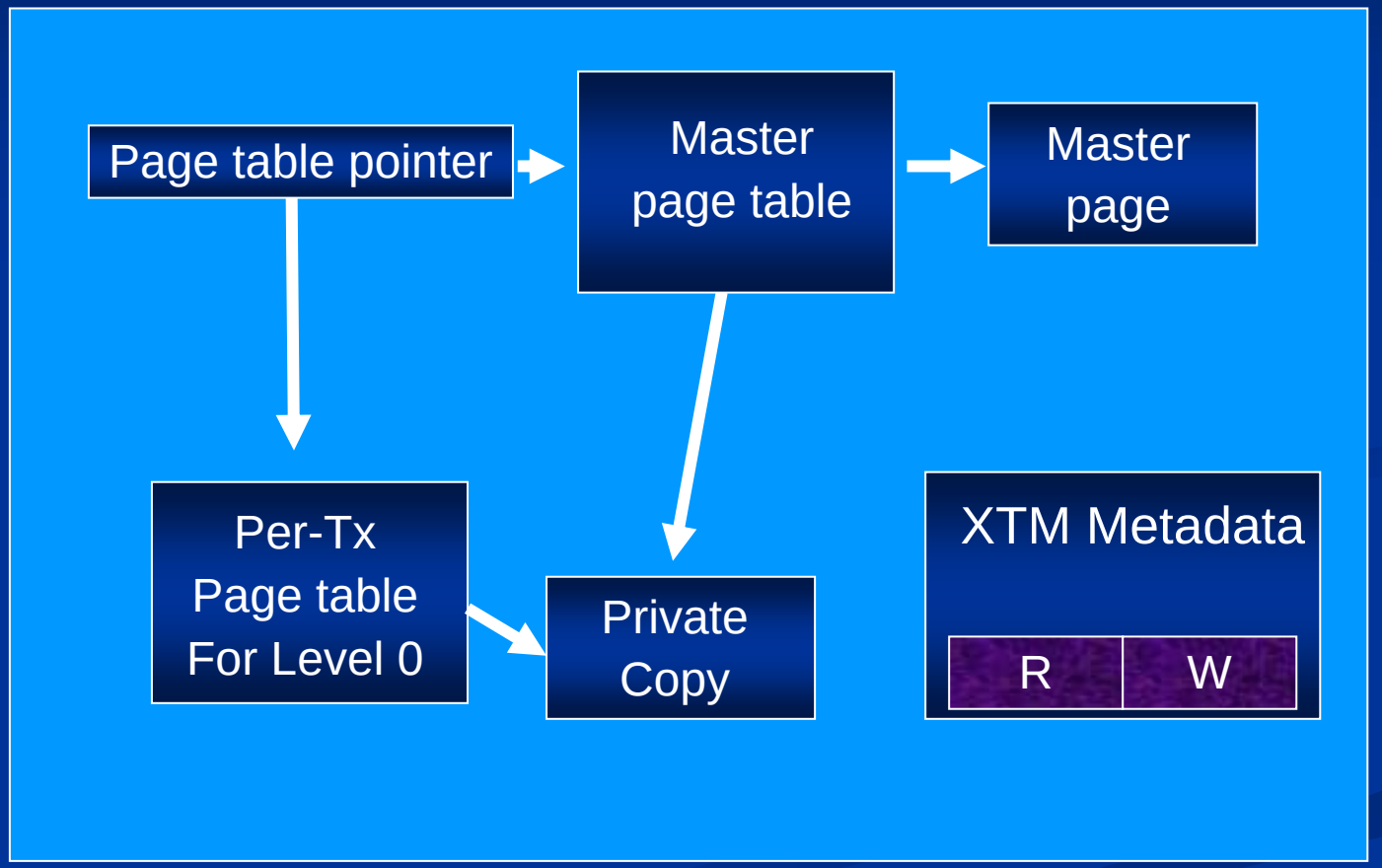
Xtn Write

↓

Xtn Read

↓

Commit

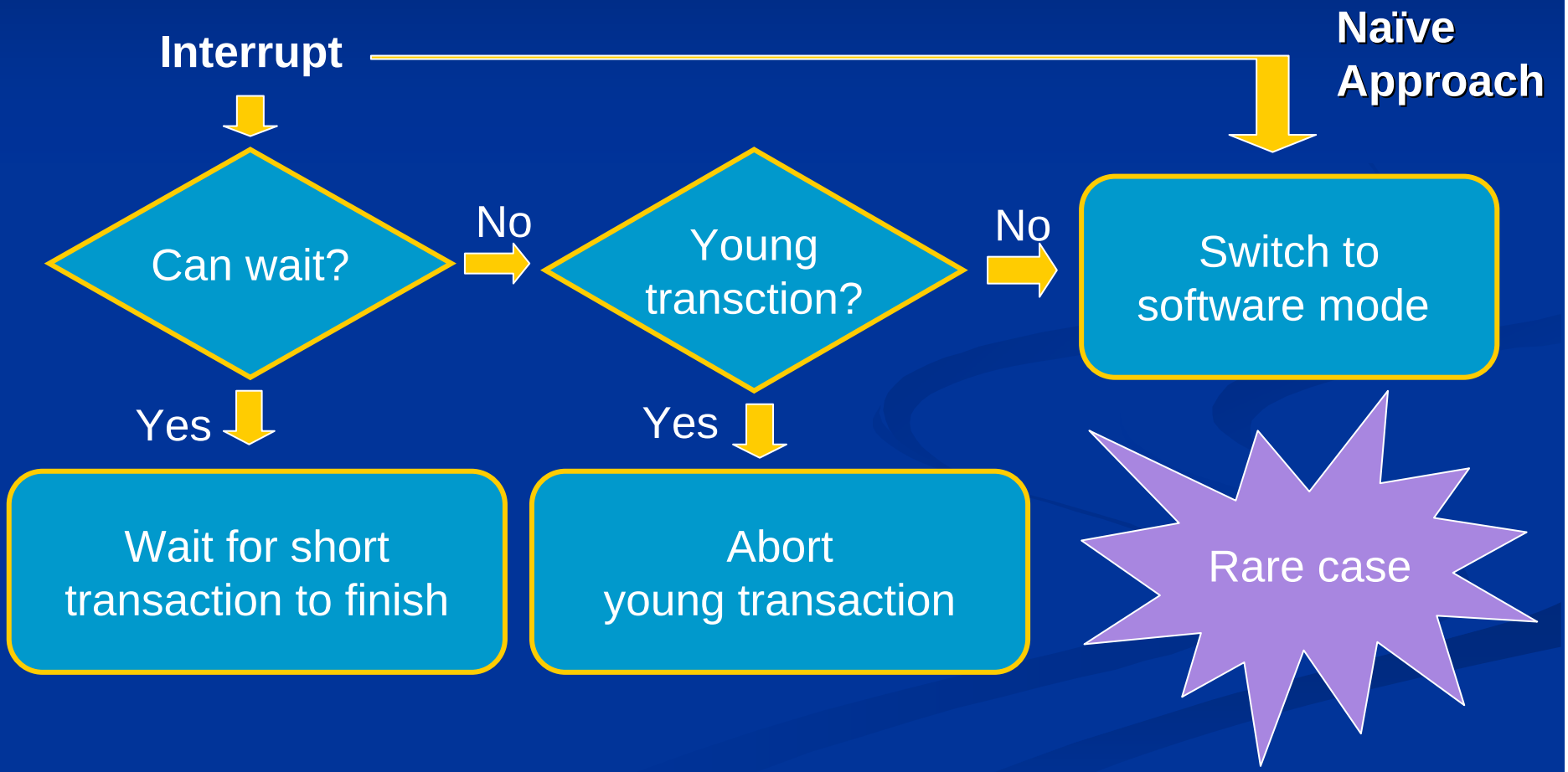


Hardware Acceleration

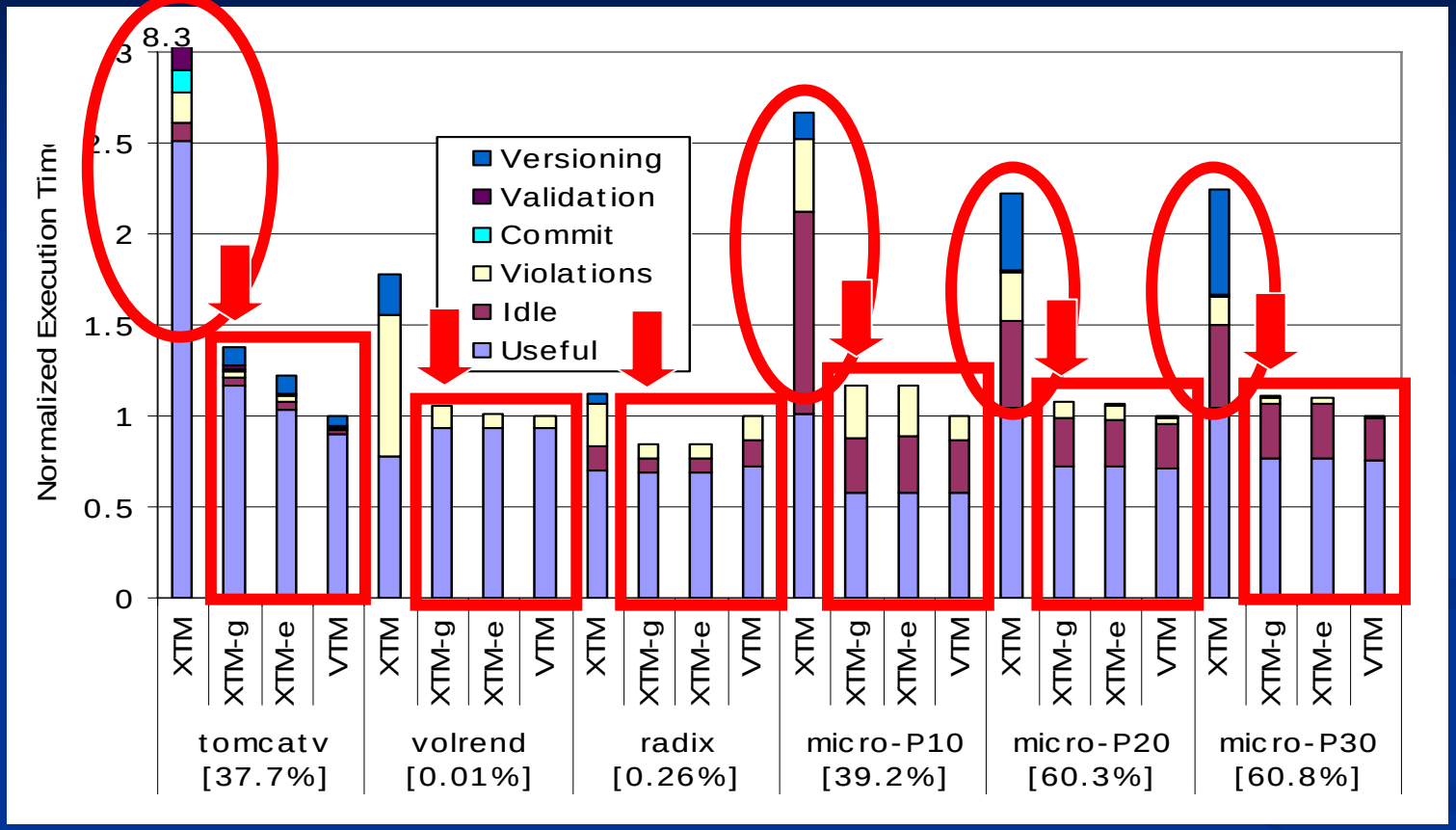
- XTM-g
  - Gradual page-by-page switching
  - Reduce the switch overhead from hardware mode to software mode
  - A portion of transactional data in private pages, the rest in the cache
  - Hardware requirement : OV(overflow) bit in page table
- XTM-e
  - Additional buffer for overflowed read/write bits
  - Reduce false conflict at page granularity
  - Hardware requirement : Eviction buffer

Time Virtualization

- Interrupt and context-switch procedure



Performance Analysis



- XTM causes no cost for applications without overflow
- XTM-g presents a good cost/performance tradeoff point
  - 20% faster to 50% slower than a fully-hardware solution

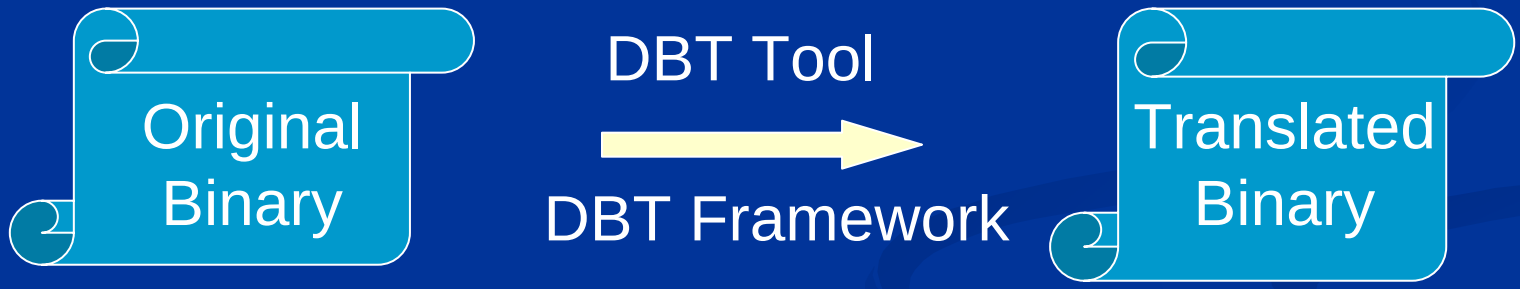
Outline

- Software parallelization : a major issue for performance
- Transactional memory
- Challenges to building TM systems
  - Common case behavior of parallel programs
  - TM virtualization
- Opportunities for systems beyond parallel programming
  - Multithreading for dynamic binary translation
  - Support for reliability, security, and fast memory snapshot
- Conclusion

Opportunities for Systems beyond Parallel Programming

Opportunity 1 : Dynamic Binary Translation

- DBT
  - Binary code is translated in run-time
  - PIN, Valgrind, DynamoRIO, StarDBT, etc



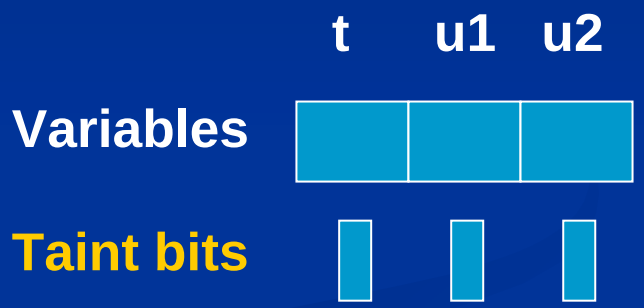
- DBT use cases
  - Translation on new target architecture
  - JIT optimizations in virtual machines
  - Binary instrumentation
    - Profiling, security, debugging, …

Example: Dynamic Information Flow Tracking (DIFT)

- **t = XX ; // untrusted data from network**
- **taint(t) = 1;**

…….

- **swap t, u1;**
- **swap taint(t), taint(u1);**
- **u2 = u1;**
- **taint(u2) = taint(u1);**



- Track untrusted data
  - A taint bit per memory byte
  - Security policy uses the taint bit.
    - E.g. no syscall with untrusted data

Problem : DBT with Parallel Program

Thread 1

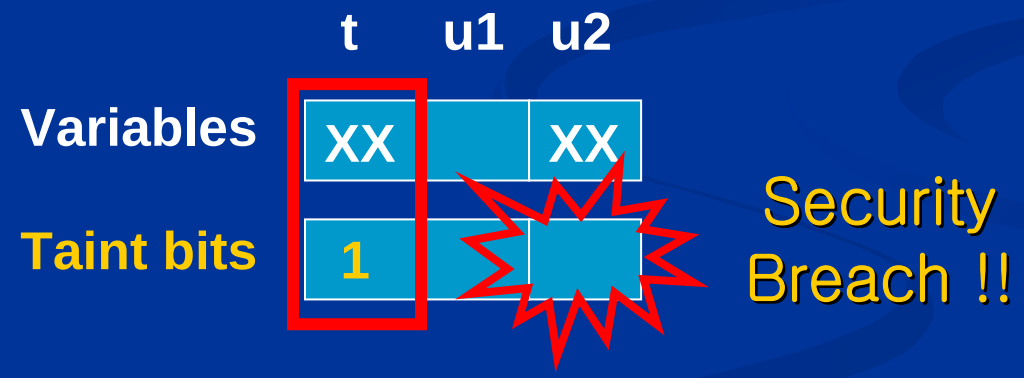
swap t, u1;

swap taint(t), taint(u1);

Thread2

u2 = u1;

taint(u2) = taint(u1);



- Atomicity between original and instrumented instructions for correctness

How to Guarantee Atomicity?

- Easy but unsatisfactory solutions
  - No multithreaded programs (StarDBT)
  - Serialization (Valgrind)
- Hard solution : Locking
  - Idea : Enclose original and instrumented instruction with lock
  - Fine-grained locks
    - locking overhead, convoying, limited scope of DBT optimizations
  - Coarse-grained locks
    - performance degradation
  - Lock nesting between app & DBT locks
    - potential deadlock
  - Tool developers should be feature + multithreading experts

Transactional Memory for Correctness of DBT

- Idea
  - Original and instrumented instructions in a transaction

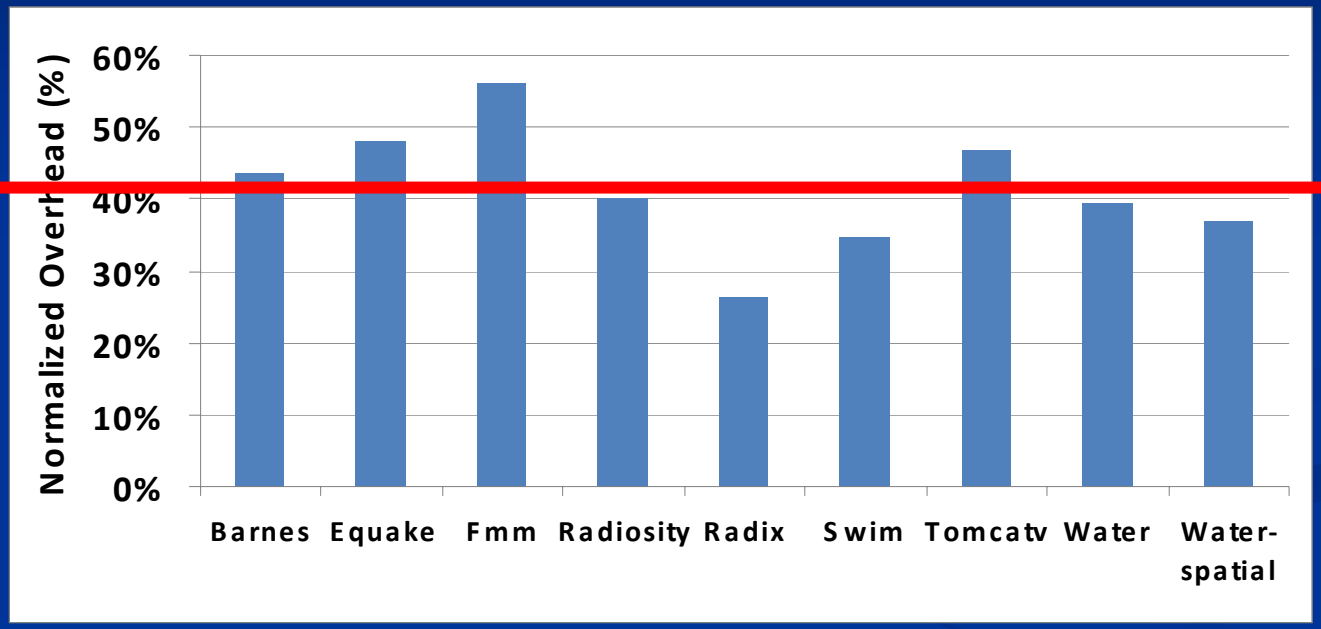
Thread 1	Thread2
TX_Begin	**TX_Begin**
swap t, u1;	u2 = u1;
swap taint(t), taint(u1);	taint(u2) = taint(u1);
TX_End	**TX_End**

- Advantages
  - Atomic execution
  - High performance through optimistic concurrency
  - Support for nested transactions

Granularity of Transaction Instrumentation

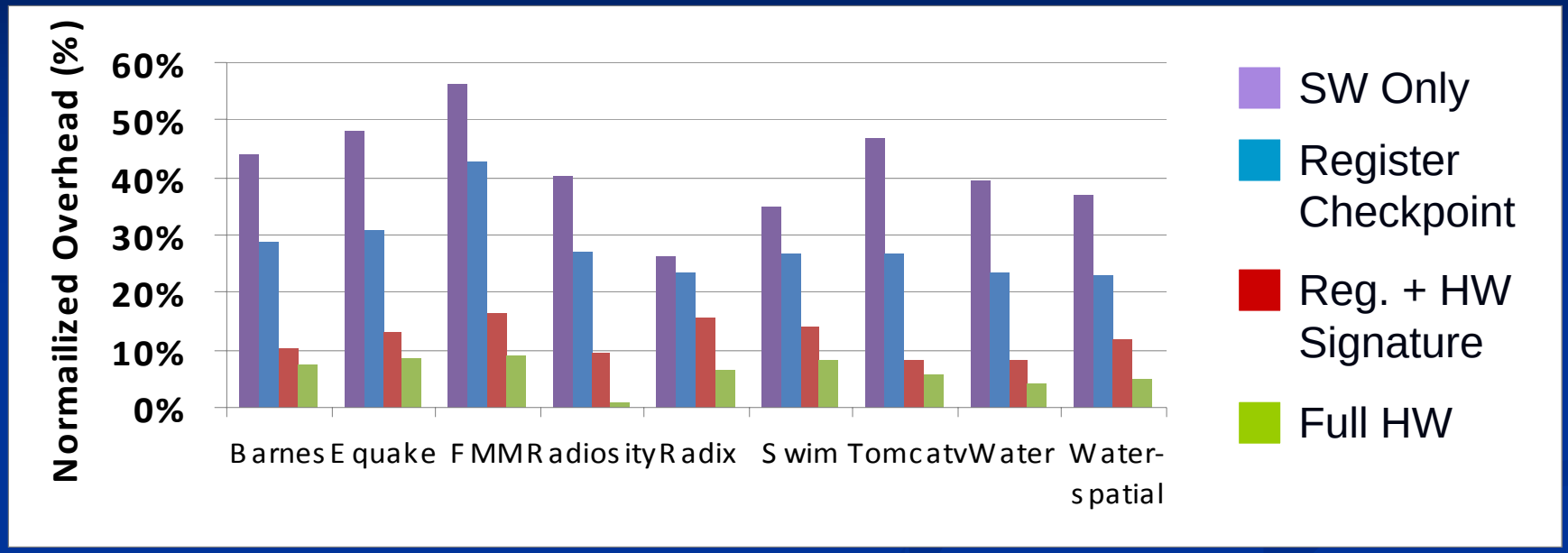
- Per instruction : short
  - High overhead of executing TX_Begin and TX_End
  - Limited scope for DBT optimizations
- Per basic block : long
  - Amortizing the TX_Begin and TX_End overhead
  - Easy to match TX_Begin and TX_End
- Per trace : longer
  - Further amortization of the overhead
  - Potentially high transaction conflict
- Profile-based sizing : dynamic
  - Optimize transaction size based on transaction abort ratio

Baseline Performance Results



- 8 CPUs
- Software TM and DIFT on PIN
- 41 % overhead on the average
  - Transaction at the DBT trace granularity

Hardware Acceleration



- Overhead reduction
  - 28% with register checkpoint
  - 12% with register checkpoint + hardware signature
  - 6% with full hardware TM

Outline

- Software parallelization : a major issue for performance
- Transactional memory
- Challenges to building TM systems
  - Common case behavior of parallel programs
  - TM virtualization
- Opportunities for systems beyond parallel programming
  - Multithreading for dynamic binary translation
  - Support for reliability, security, and fast memory snapshot
- Conclusion

Opportunity 2 : Improving Other System Metrics

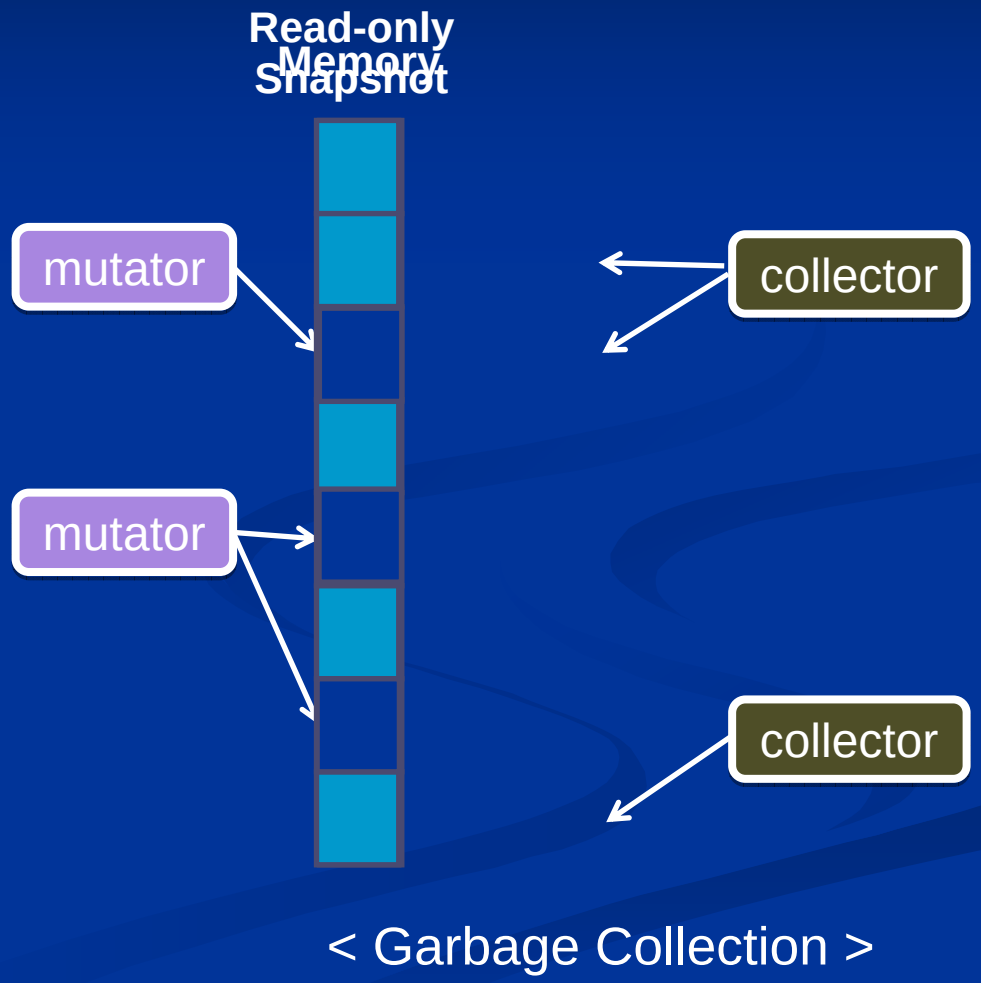
- TM hardware consists of
  - Fine-grain data versioning HW
  - Fine-grain access tracking HW
  - Fast exception handlers
- Can use such HW for other purposes
  - Reliability, Security, …
- The benefits for SW
  - Finer granularity (compared to VM-based approach)
  - User-level event handling (compared to VM-based approach)
  - No instrumentation overhead (compared to DBT-based approach)
  - Simplified code (compared to DBT-based approach)

Outline for TM Hardware Application

- Reliability
  - Global & local checkpoints (data versioning)
- Security
  - Fine-grain read/write barriers (address tracking)
  - Isolated execution (data versioning)
- Memory snapshot (data versioning)
  - Concurrent garbage collector
  - Dynamic memory profiler

Memory Snapshot

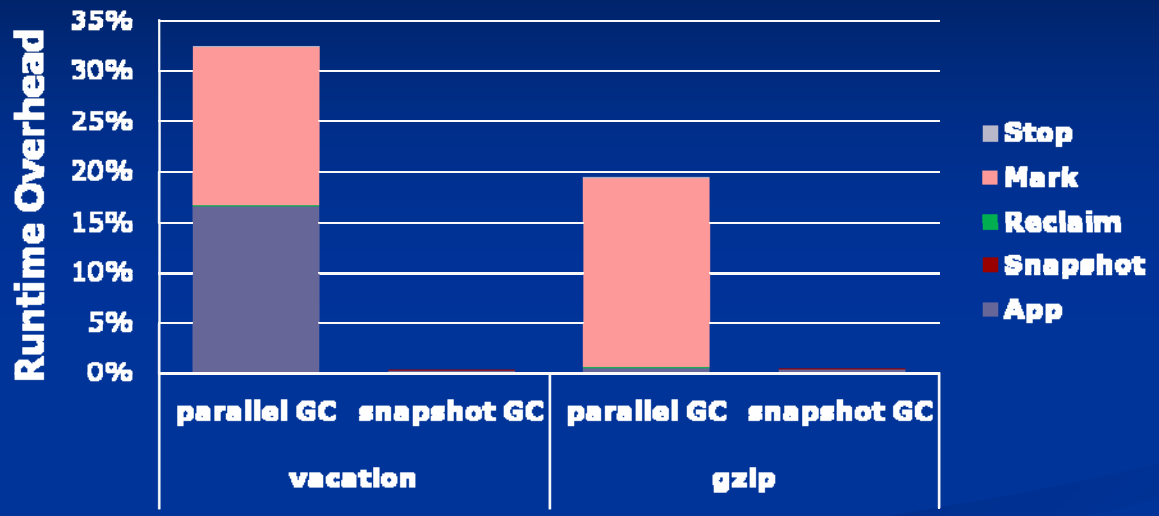
- Snapshot
  - Read-only image
  - Multiple regions
  - Shared by multiple threads
- Applications
  - Service threads that analyze memory in parallel with app threads
  - Garbage collection, memory profiling (heap & stack), …



TM Hardware ⇒ Snapshot

- Feature correspondence
  - TM metadata ⇒ track data written since snapshot
  - TM versioning ⇒ storage for progressive snapshot
    - Including virtualization mechanism
  - TM conflict detection ⇒ catch errors
    - Writes to read-only snapshot
- Differences & additions
  - Data versioning for single thread Vs. multiple thread
  - Table to record snapshot regions
- Resulting snapshot system
  - Fast : O(# CPUs) Scan (create) and O(1) write/read
  - Small memory footprint : O(# memory locations written)

GC Overhead



- Parallel GC: stop app threads & run GC threads
  - 20% to 30% overhead for memory intensive apps
- Snapshot GC ⇒ GC is essentially free
  - Fast : Stop app, take snapshot, then run GC & app concurrently
  - Simple : +100 lines over parallel GC by Boehm
    - Fundamentally simpler than any other concurrent GC

Conclusion

- Challenges to building TM systems
  - Common case behavior of parallel programs
    - Extract architectural parameters for efficient TM system design
  - TM virtualization
    - Overcome the limitation of TM hardware
- Opportunity for system beyond parallel programming
  - Multithreading for dynamic binary translation
    - Fix correctness issue for DBT
  - Support for reliability, security, and fast memory snapshot
    - Improve important system metrics other than performance

Acknowledgement

- KyungHae, wife
- Parents, brother, in-laws
- Prof. Kozyrakis, advisor
- Prof. Olukotun, associate advisor
- Prof. Garcia-Molina
- Prof. Saraswat
- TCC group mates and research colleagues
- Korean mafia