Highly parallel programming with Transactional Memory

Ricardo Filipe
João Barreto
Motivation

- Multicore architectures are the norm
- Manycores are on their way
Why parallel programming is hard

- TM simplifies the task of synchronizing critical sections
Why parallel programming is hard

- TM simplifies the task of synchronizing critical sections
- …But forking code into threads is still hard
Challenges of forking threads

- Forking a TM thread incurs creation and management overheads
Challenges of forking threads

- Forking a TM thread incurs creation and management overheads
- TM threads must be commutative
Usually we have more available cores than application threads...

2 Threads
4+ Cores...
Is there any parallelism hidden inside each thread that the programmer forks?
Motivational results

- RBTree – “contains” op, 1 TM thread, 2 TLS threads
Motivational results

- STMBench7 – long Txs, 1 TM thread, 3 TLS threads
How do we explore intra-thread parallelism?

CHALLENGE ACCEPTED
I’m awesome! Divided my application into more TM threads!
Nested Transactional Memory

- Inner TM threads don’t run in parallel efficiently
  - Hinders expert programmers and libraries designed to run in TM
“Solving parallel nested transactions is complex and its efficient implementation appears to be questionable”
Parallel Nested TM: NepalTM

- First level of threads is managed with TM
- Deep nesting levels use mutual exclusion locks
Parallel Nested TM: XCilk

- Gives a provable performance guarantee for conflict free write-only TM parallel nested applications
• Gives a provable performance guarantee for conflict free write-only TM parallel nested apps

• Introduces Lemma for conflicts in parallel nested TMs:

  • T reads from l: if there exists a transaction $T'$ such that $T' \in \text{writers}(l)$, $T' \neq T$ and $T' \notin \text{ancestors}(T)$.

  • T writes to l: if there exists a transaction $T'$ such that $T' \in \text{readers}(l) \cup \text{writers}(l)$, $T' \neq T$ and $T' \notin \text{ancestors}(T)$. 
• Each version owner lock now has owner information always present for ancestor query
• Uses a list to validate the ancestors read-sets on each write
Parallel Nested TM: NesTM

- Each version owner lock now has owner information always present for ancestor query
- Uses a list to validate the ancestors read-sets on each write
Undesirable features

- Visible reads
- Linear overhead
- All operations are “writes”
I’m not that good... Help!
Thread Level Speculation does some work for you
Challenges of TLS

• Non-optimal automatic code division
Challenges of TLS

- Non-optimal automatic code division
- Mirror sequential code execution
Challenges of TLS

- Non-optimal automatic code division
- Mirror sequential code execution
- Trashes TM when simply used on top of it
Thread Level Speculation

- In-Place Write Commit Approaches
  - Each task commits when it ends
  - Rollback goes to the task before the conflict
Thread Level Speculation

- **Serial Write Commit Approaches**
  - Last task of the thread commits all writes
  - “Future tasks” can be waiting some time
Conclusions:
So much untapped parallelism!

• Solve the problem of parallel nested transactions efficiently

• Solve Thread Level Speculation on top of Transactional Memory adequately
Questions?

rfilipe@gsd.inesc-id.pt

www.gsd.inesc-id.pt