Improving Continuation-Powered Method-Level Speculation for JVM Applications

Ivo Anjo   João Cachopo

ESW
INESC-ID Lisboa/Instituto Superior Técnico/Universidade de Lisboa

December 2013

-8 days left in 2k13

...and still many applications are single-threaded

Hard to rewrite many existing applications to work in parallel

Hard to rewrite many existing applications to work in parallel

↓

Automatic Parallelization

Hard to rewrite many existing applications to work in parallel

↓

Automatic Parallelization

↓

Analyze applications, run parts in parallel

Many Java/JVM applications are irregular

- Inheritance
- Polymorphism
- Encapsulation

... other OO features

Many Java/JVM applications are irregular

- Inheritance
- Polymorphism
- Encapsulation

  ... other OO features

➜ Hard to analyze statically

Many Java/JVM applications are irregular

- Inheritance
- Polymorphism
- Encapsulation

  ... other OO features

➜ Hard to analyze statically

Speculative Parallelization

- No need to statically prove that parallelization is valid
- Correctness dynamically ensured at runtime

After task identification, how to fully utilize a parallel machine?

After task identification, how to fully utilize a parallel machine?

New techniques for improved resource

- *usage*
- *management*

In previous work we introduced **JaSPEx-MLS**...

JaSPEx-MLS

- Software-based speculative parallelization for Java

JaSPEx-MLS

- Software-based speculative parallelization for Java
- Bytecode rewriting

JaSPEx-MLS

- Software-based speculative parallelization for Java
- Bytecode rewriting
- Automatic transactification

JaSPEx-MLS

- Software-based speculative parallelization for Java
- Bytecode rewriting
- Automatic transactification
- Handling non-transactional operations

JaSPEx-MLS

- Software-based speculative parallelization for Java
- Bytecode rewriting
- Automatic transactification
- Handling non-transactional operations
- Method-Level Speculation

JaSPEx-MLS

- Software-based speculative parallelization for Java

- Bytecode rewriting

- Automatic transactification
- Handling non-transactional operations

- Method-Level Speculation
- Custom STM model

JaSPEx-MLS

- Software-based speculative parallelization for Java

- Bytecode rewriting

- Automatic transactification
- Handling non-transactional operations

- Method-Level Speculation
- Custom STM model
- Nested speculation

JaSPEx-MLS

- Software-based speculative parallelization for Java

- Bytecode rewriting

- Automatic transactification
- Handling non-transactional operations

- Method-Level Speculation
- Custom STM model
- Nested speculation

...on top of the OpenJDK Java VM + Continuation support

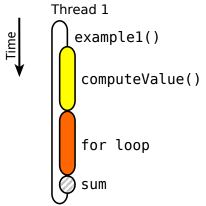
Method-Level Speculation

```
int example1() {
    int x = computeValue();
    int y = 0;
    for (...) y += ...;
    return x+y;
}
```

- Run method call in parallel with code following its return

Method-Level Speculation

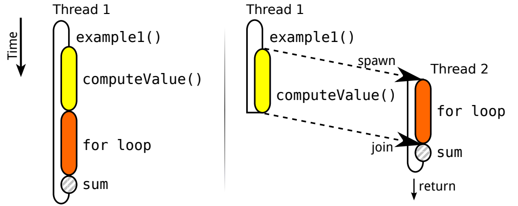
```
int example1() {
    int x = computeValue();
    int y = 0;
    for (...) y += ...;
    return x+y;
}
```



- Run method call in parallel with code following its return

Method-Level Speculation

```
int example1() {
    int x = computeValue();
    int y = 0;
    for (...) y += ...;
    return x+y;
}
```

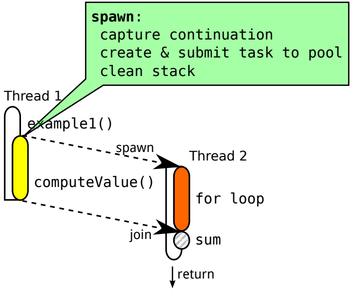


- Run method call in parallel with code following its return

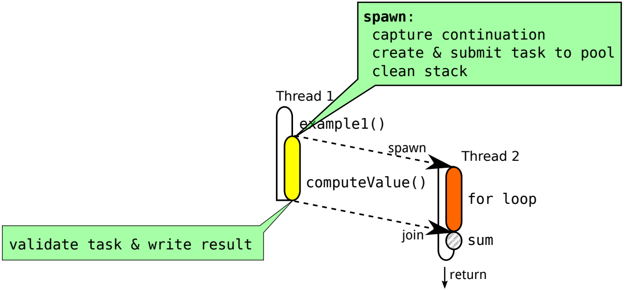
What really happens behind the scenes

```
int example1() {
    Future x = spawn computeValue();
    int y = 0;
    for (...) y += ...;
    return x.get()+y;
}
```

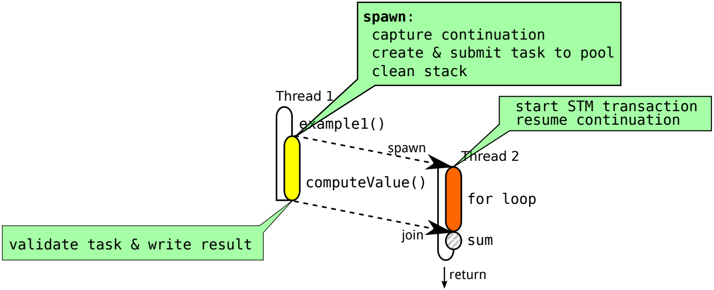
What really happens behind the scenes



What really happens behind the scenes



What really happens behind the scenes



MLS spawn

- Existing task – **method call**:
  - ➔ Keeps executing in the same thread
- New task – **code following method call**:
  - ➔ Submitted to the thread pool

    ...if there are free threads

New task

- Starts executing

  ...until
- Needs value from other unfinished task
- Needs to execute non-transactional operation
- Ready to commit, but parent still working

New task

- Starts executing

  ...until

- Needs value from other unfinished task
  - ➔ Wait for other task
- Needs to execute non-transactional operation

- Ready to commit, but parent still working

New task

- Starts executing

  ...until

- Needs value from other unfinished task
  - ➔ Wait for other task
- Needs to execute non-transactional operation
  - ➔ Wait for program-order
- Ready to commit, but parent still working

New task

- Starts executing

  ...until
- Needs value from other unfinished task
  - → Wait for other task
- Needs to execute non-transactional operation
  - → Wait for program-order
- Ready to commit, but parent still working
  - → Wait for parent task

Improving MLS (aka outline):

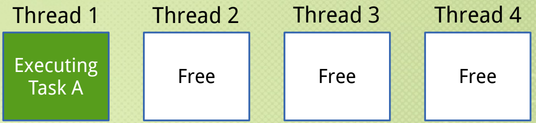
- Task buffering
- Task freezing
- STM-assisted return value prediction
- Captured memory

➜ Remove waiting, allow more tasks

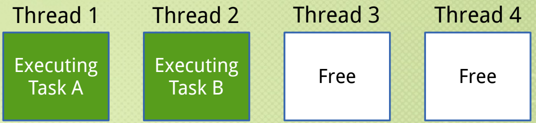
Thread Pool

Original Design:
Limited number of threads + Direct hand-offs

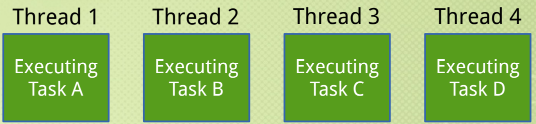
Thread Pool – Underusage



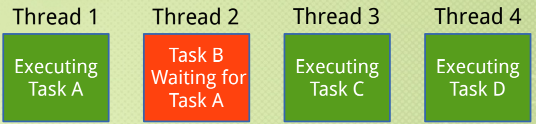
Thread Pool – Underusage



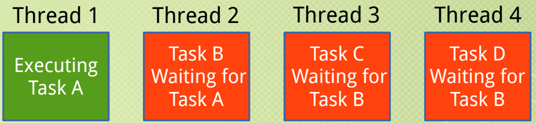
Thread Pool – Underusage



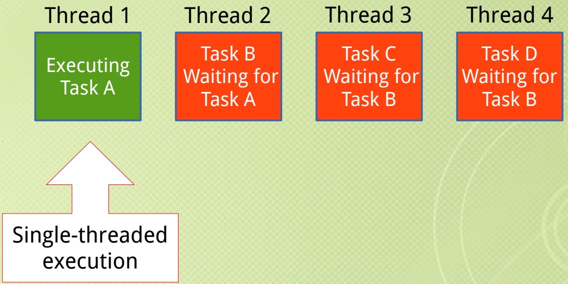
Thread Pool – Underusage



Thread Pool – Underusage

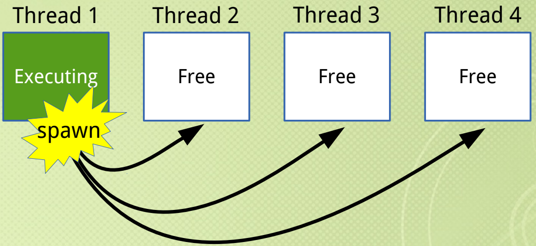


Thread Pool – Underusage

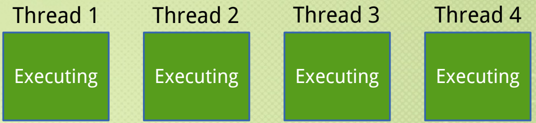


Another issue...

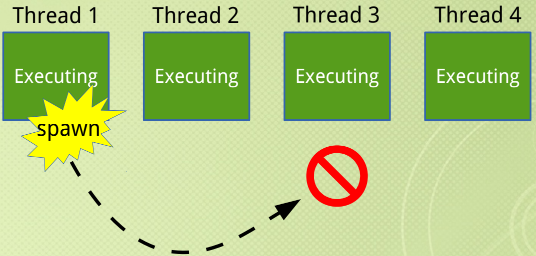
Thread Pool – Underusage II



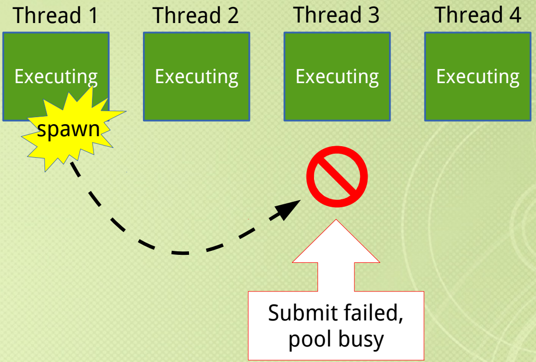
Thread Pool – Underusage II



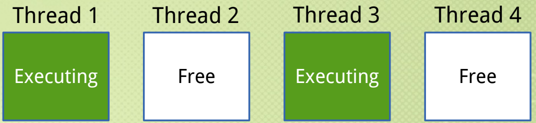
Thread Pool – Underusage II



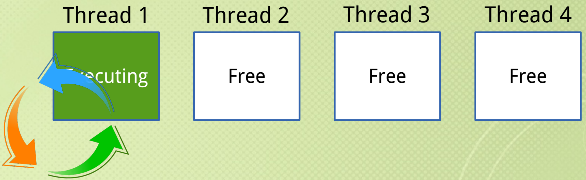
Thread Pool – Underusage II



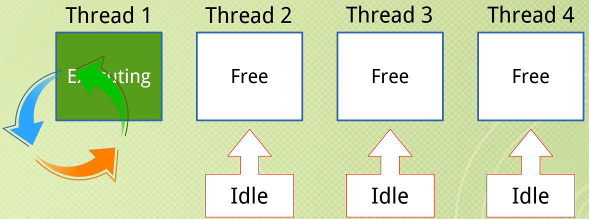
Thread Pool – Underusage II



Thread Pool – Underusage II



Thread Pool – Underusage II



Thread Pool Issues

- # Program Threads > # Hardware Threads?
  - Lots of context switching overheads
    - *...especially if they are actually working and not just waiting*

Thread Pool Issues

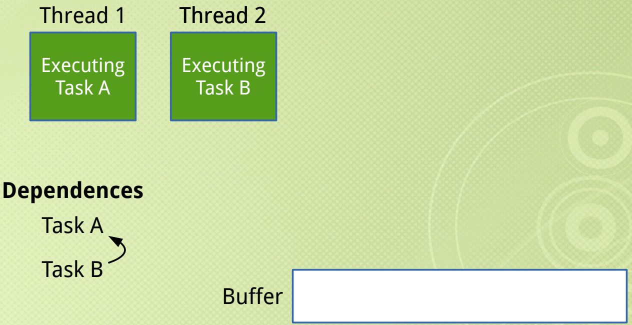
- # Program Threads > # Hardware Threads?
  - Lots of context switching overheads
    - *...especially if they are actually working and not just waiting*
  - Band-aid solution?

Thread Pool Issues

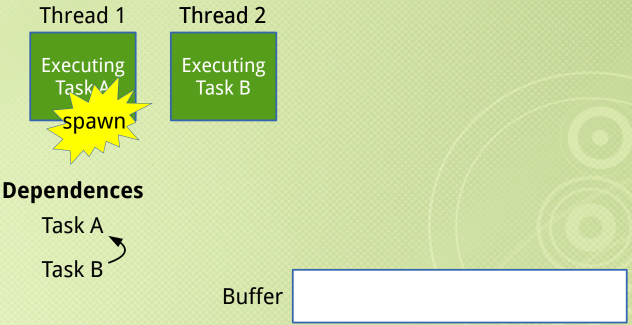
- # Program Threads > # Hardware Threads?
  - Lots of context switching overheads
    - *...especially if they are actually working and not just waiting*
  - Band-aid solution?

- What about buffering?

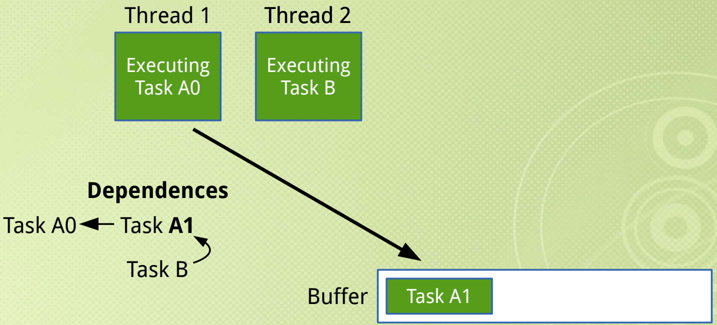
Thread Pool – Naive Buffering



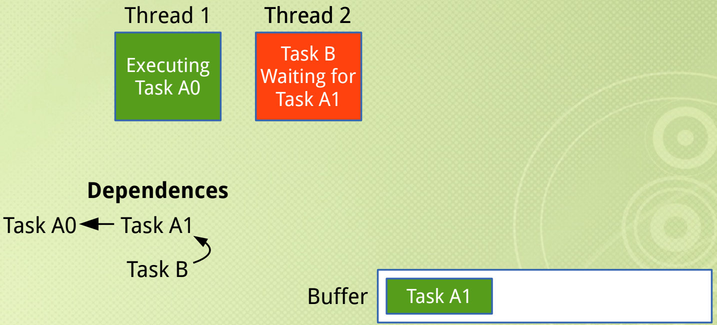
Thread Pool – Naive Buffering



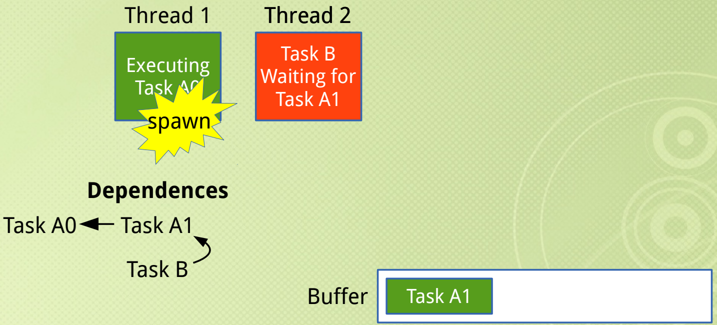
Thread Pool – Naive Buffering



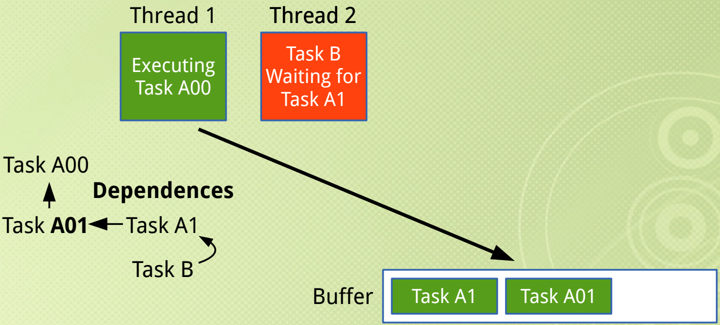
Thread Pool – Naive Buffering



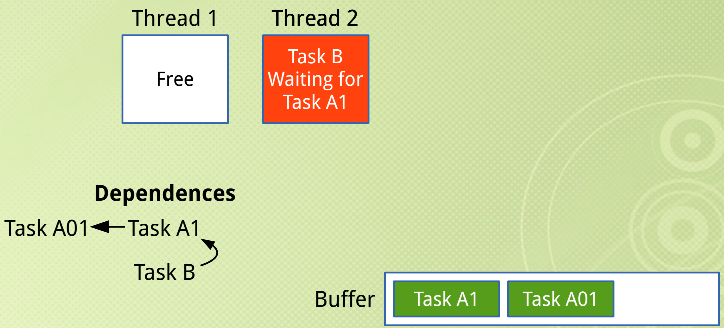
Thread Pool – Naive Buffering



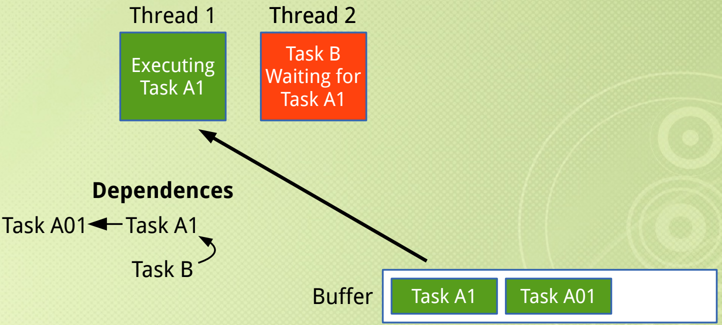
Thread Pool – Naive Buffering



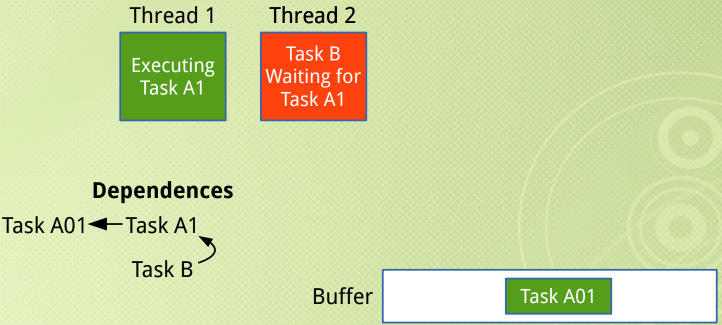
Thread Pool – Naive Buffering



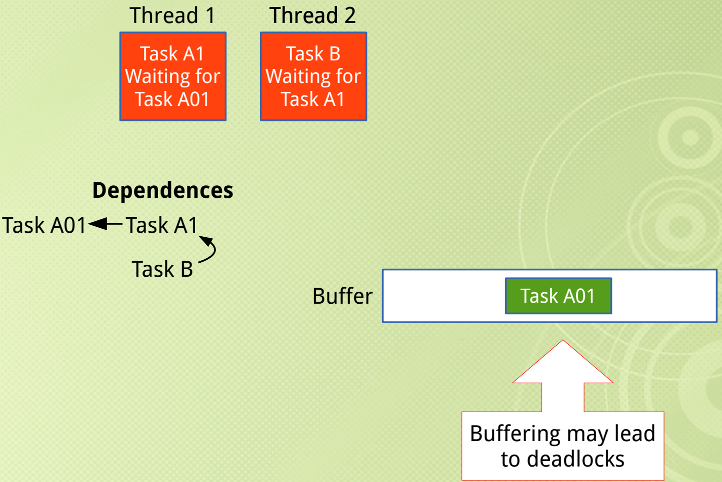
Thread Pool – Naive Buffering



Thread Pool – Naive Buffering



Thread Pool – Naive Buffering



Hybrid Thread Pool Queue

- Try to combine advantages from buffering
  - More efficient handoffs
  - Better resource utilization

  ...while still preserving correctness

Hybrid Thread Pool Queue

- Augment pool with deadlock detector
  - Detects when all threads are waiting simultaneously
- Buffer tasks by default, fallback to direct hand-offs when issue detected
- Preserves correctness
  - Deadlocks are very rare (workload-specific)
  - Better performance in the normal case

Previously...

MLS spawn

- Existing task – **method call**:
  - ➔ Keeps executing in the same thread
- New task – **code following method call**:
  - ➔ Submitted to the thread pool

    ...if there are free threads

Now...

MLS spawn

- Existing task – **method call**:
  - ➔ Keeps executing in the same thread
- New task – **code following method call**:
  - ➔ Submitted to the thread pool

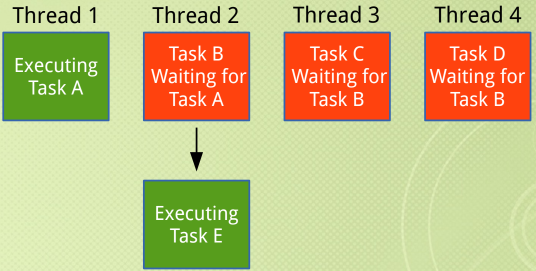
    ...if there a [illegible] *ads*

BUFFER

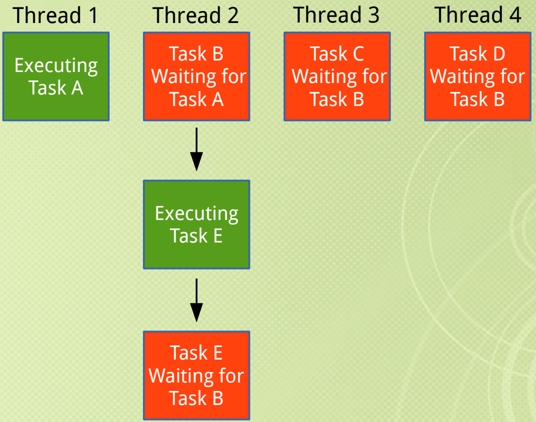
Going back to the waiting part...

Reuse threads?

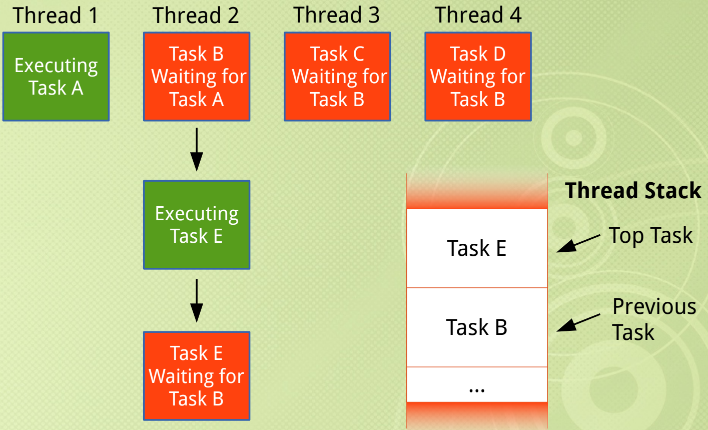
Going back to the waiting part...



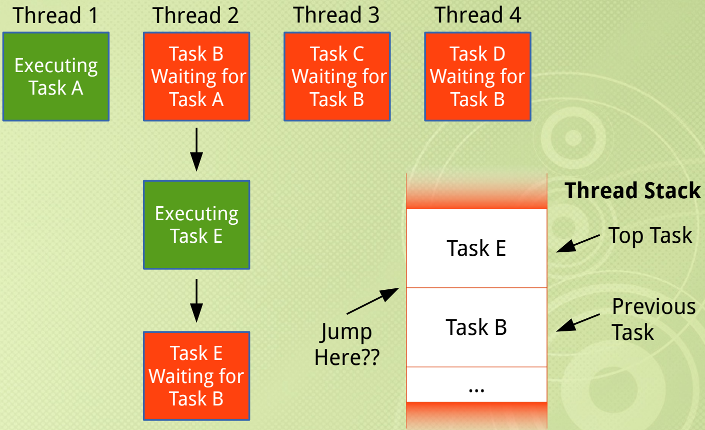
Going back to the waiting part...



Going back to the waiting part...



Going back to the waiting part...

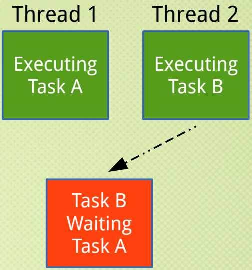


Task Freeze

Idea:

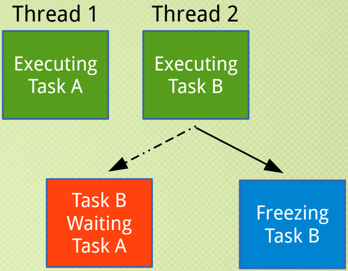
- Use JVM with continuation support to save and transfer task state

Task Freeze

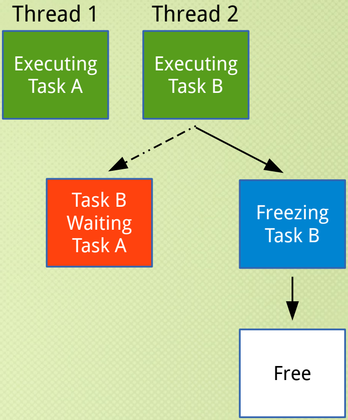


Thread 1
Thread 2
Executing Task A
Executing Task B
Task B Waiting Task A

Task Freeze

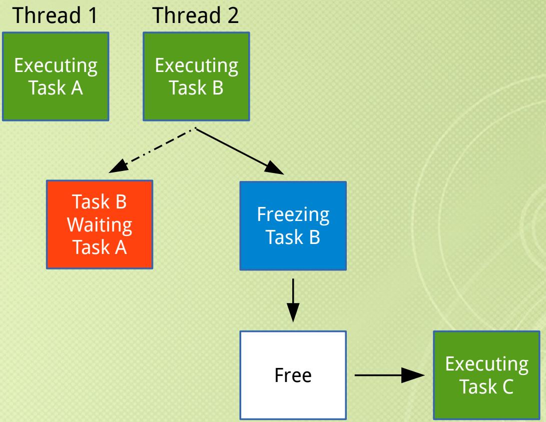

Thread 1
Thread 2
Executing Task A
Executing Task B
Task B Waiting Task A
Freezing Task B

Task Freeze

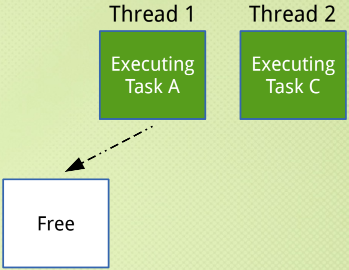


Thread 1
Thread 2
Executing Task A
Executing Task B
Task B Waiting Task A
Freezing Task B
Free

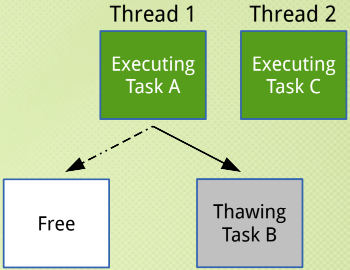
Task Freeze


Thread 1
Thread 2
Executing Task A
Executing Task B
Task B Waiting Task A
Freezing Task B
Free
Executing Task C

Task Freeze

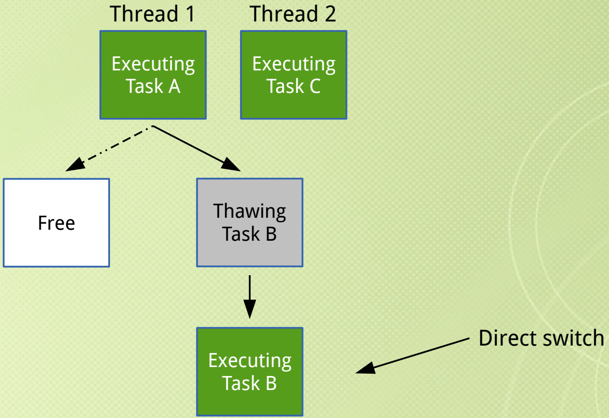


Task Freeze



Thread 1
Thread 2
Executing Task A
Executing Task C
Free
Thawing Task B

Task Freeze


Thread 1
Thread 2
Executing Task A
Executing Task C
Free
Thawing Task B
Executing Task B
Direct switch

Improved Runtime Model

New task

- Starts executing

  ...until

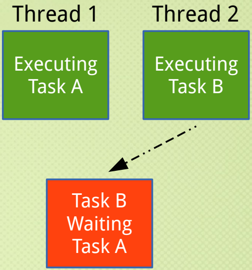
- Needs value from other unfinished task
  - Wai FREEZE ask
- Needs to execute non-transactional operation
  - Wai FREEZE n-order
- Ready to commit, but parent still working
  - Wai f FREEZE task

Return Value Prediction

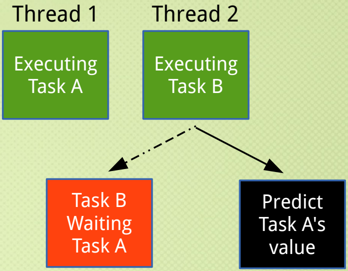
Idea:

- Instead of waiting, guess the value needed

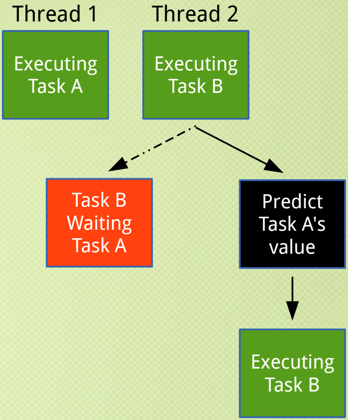
Return Value Prediction



Return Value Prediction



Return Value Prediction



Return Value Prediction

Our idea:

- STM-assisted RVP
  - Use STM to register prediction in the **read-set**
  - Use STM to validate prediction
    - Validated during transaction commit

Return Value Prediction

- STM-assisted RVP
  - Supports obtaining multiple predictions (from multiple other tasks)
  - Pluggable predictor framework

Improved Runtime Model II

New task

- Starts executing

  ...until

- Needs value from other unfinished task
  - → Wai FREEZE as PREDICT
- Needs to execute non-transactional operation
  - → Wai FREEZE n-order
- Ready to commit, but parent still working
  - → Wai FREEZE task

Captured Memory

- Memory allocated cannot escape its allocating transaction
  - ➜ Objects are transaction-local until commit

Captured Memory

- If objects are transaction-local, we can access them directly

  → **No STM overhead**

Captured Memory in MLS

Issue:

- Objects may escape their allocating transactions under MLS

```
void example2() {
    A a = new A();                     Task A
    spawn computeValue();

    a.field = 10;                      Task B
    …
}
```

➔ No such thing as captured memory in our model?

Captured Memory in MLS

- But, looking at the original sequential program order

Dependences

Task A

Task B

- Objects may only escape to tasks **more speculative** than the ones that allocated them

Captured Memory in MLS

Solution

- Access objects in captured memory directly
- **More speculative** tasks still use STM to access the objects

Captured Memory: Further Advantages

```
void example3() {
    A a = new A();
    a.i++;                                   Task A
    spawn computeValue();

    a.i--;                                   Task B
    …
}
```

Task B would always abort

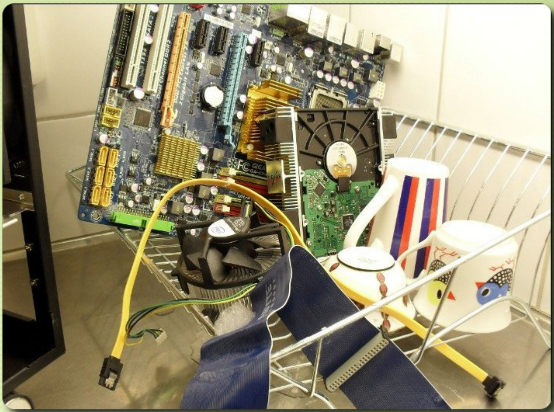
Captured Memory: Further Advantages

```
void example3() {
    A a = new A();
    a.i++;                          Task A
    spawn computeValue();

    a.i--;                          Task B
    …
}
```

Task B would always abort

➔ Captured memory is very beneficial to the MLS model

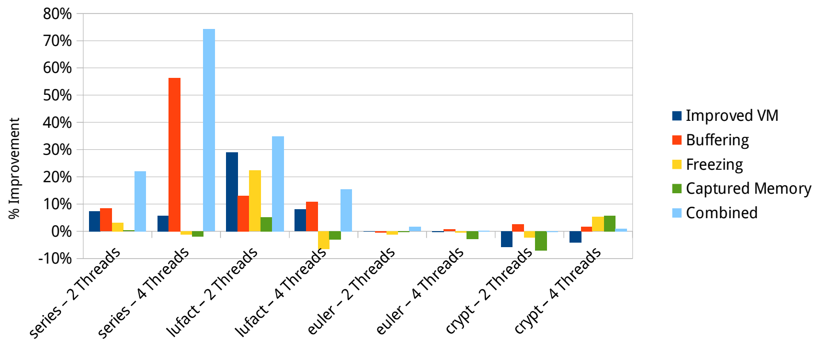


Benchmarks

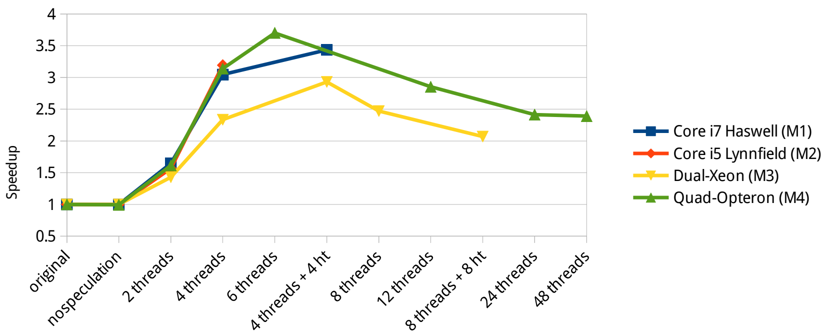
- **M1**: Intel Core i7 *Haswell* 4770 / 16GB
  - 4 cores + ht: 8 hw threads
- **M2**: Intel Core i5 *Lynnfield* 750 / 8GB
  - 4 cores: 4 hw threads
- **M3**: Intel Xeon *Nehalem* E5520 (dual-socket) / 24GB
  - 8 cores + ht: 16 hw threads
- **M4**: AMD Opteron *Magny-Cours* 6168 (quad-socket) / 128GB
  - 48 cores: 48 hw threads

- 64-bit Ubuntu / CentOS

- Java Grande Forum benchmark suite
- No code modifications applied

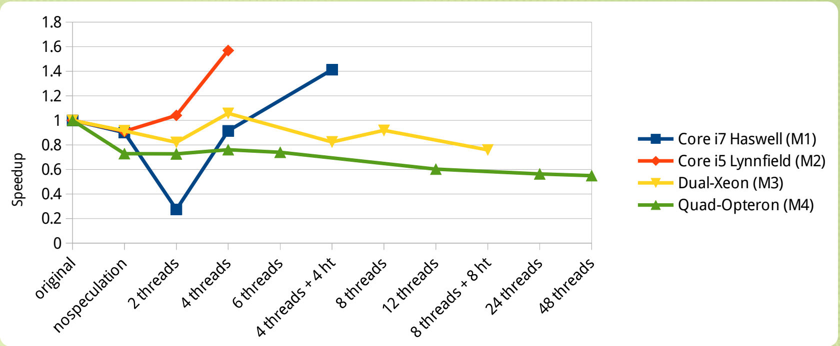


- Improvement of each new extension + their combined usage
- New features work together
  - Freezing can cause slowdown when used without buffering
  - Improved VM speeds up old code and new extensions



- Absolute speedup in the `series` benchmark

- Barnes-Hut from the Lonestar Benchmark Suite
- No code modifications applied



- Bigger bytecode transformations overhead than series
  - 10% for M1-M3
  - 30% for M4

- Buffering disabled for M1 due to bug

Conclusions

- Speculative parallelization is a viable approach for irregular JVM applications
- Presented several techniques that work together to enhance MLS
  - Task buffering via hybrid thread pool queue
  - Task freezing
  - STM-assisted Return Value Prediction
  - Captured memory
- Benchmarks from implemented techniques in JaSPEx-MLS yield speedups for unmodified applications on real hardware on a production-ready JVM

Thank you!