Jörg Domaschka, Hans P. Reiser, Andreas I. Schmied, Franz J. Hauck

Distributed Systems Lab,
Faculty of Computer Science
Ulm University
Germany

Aspectix Research Group
http://www.aspectix.org/
Motivation

Observations:

- *Non-fault-tolerant* object-based middleware usually supports multi-threading
- Multi-CPU hardware and multi-core CPUs increasingly popular
Observations:

- Non-fault-tolerant object-based middleware usually supports multi-threading
- Multi-CPU hardware and multi-core CPUs increasingly popular

Our problem domain:

- Object-based distributed application
- Active replication of stateful objects (FTflex/Aspectix)
- Reentrant mutexes, condition variables, time bounds (Java synchronisation model)
- Interactions between objects using remote invocations (nested invocations)
Motivation: Replication

- **Determinism** is mandatory for active replication
- **Multithreading** is source of non-determinism

Two clients call deposit

Concurrent execution: Nondeterministic result

```java
public void deposit(Object value) {
    if (value != null) {
        synchronized (depositeLock) {
            spaces--; // spaces < 0 ?
            if (spaces < 0) depositeLock.wait();
        }
        synchronized (fetchLock) {
            elements++;
            if (elements <= 0) fetchLock.notify();
        }
    }
}
...`
Usual approach for active replication:

\text{abcast} + \textbf{sequential request execution}
Usual approach for active replication:
abcast + **sequential request execution**

Problems:

- **Problem 1:** Deadlocks
Usual approach for active replication: abcast + **sequential request execution**

- **Problem 1:** Deadlocks
- **Problem 2:** No condition variables
Usual approach for active replication:

\[ \text{abcast} + \text{sequential request execution} \]

- Problem 1: Deadlocks
- Problem 2: No condition variables
- **Problem 3: Idle time during nested invocations**
Usual approach for active replication:
   abcast + **sequential request execution**

- Problem 1: Deadlocks
- Problem 2: No condition variables
- Problem 3: Idle time during nested invocations
- **Problem 4: Cannot benefit from multiple CPUs**
Usual approach for active replication:

- \texttt{abcast} + \textbf{sequential request execution}

- Problem 1: Deadlocks
- Problem 2: No condition variables
- Problem 3: Idle time during nested invocations
- Problem 4: Cannot benefit from multiple CPUs

\(\Rightarrow\) Need for deterministic multithreading
Outline

1. Existing Approaches

2. Code Analysis

3. Conclusions
Status Quo in Deterministic Multithreading

SAT

Single Active Thread Algorithm (Domaschka et al., PDCS’06)

- One active thread
- Multiple blocked threads
- Multiple threads in ready state
- No deadlocks
- No idle time during nested invocations
- Support for Java synchronization
- Cannot benefit from multiple CPUs
Loose Synchronization Algorithm (Basile et al., SRDS’02)

- Leader-Follower-Synchronisation:
  - Leader assigns locks and broadcasts (lock,thread) table
  - Followers grant locks according to broadcast
  - Fail-over strategy to handle leader failure

- Optimal concurrency: Identical to non-replicated execution at leader
- Requires additional communication
- Unfavourable for active replication!
  - Active replication usually used to minimize downtime after failures
  - Fail-over requires recovery procedure after accurate detection of primary failure
Preemptive Deterministic Scheduling Algorithm (Basile et al., SRDS’03)

- Concurrent execution without communication
- Uses fixed-size thread pool
- Execution in rounds:
  - Start of rounds: all threads are suspended on lock request
  - Deterministic lock assignment on the basis of thread IDs
  - Execution of all threads until all are suspended again
- Problematic assignment of request to thread pool
- Efficient only if all threads have approx. same running time and no conflicting locks
- Inefficient with nested invocations of variable duration
- No support for condition variables (wait/notify)
Multiple Active Threads Algorithm (Reiser et al., SRDS'06)

- Extension to SAT
- A single *active primary* thread can *acquire* locks
- *Non-active primary* threads wait to become active
- Secondary threads can execute concurrently
- Secondary threads ordered by arrival time

- Primary thread becomes secondary at deterministic scheduling points
- Deterministic selection of primary thread
  \[\Rightarrow\] Ensures strictly consistent state manipulations
- Solves all problems
Scheduler Integration

Goals:

- No changes to VM or Compiler
- Programmer shall use familiar constructs

Example:

```java
synchronized (depositeLock) {
    spaces++;
    if (spaces <= 0) depositeLock.notify();
}
```

⇒

```java
scheduler.lock(depositeLock);
try {
    spaces++;
    if (spaces <= 0) scheduler.mtnotify(depositeLock);
} finally {
    scheduler.unlock(depositeLock);
}
```

Jörg Domaschka (Ulm University)  Aspectix Research Group  http://www.aspectix.org/
Scheduler Integration

Goals:

- No changes to VM or Compiler
- Programmer shall use familiar constructs

⇒ Intercept native Java synchronisation via code transformation

Example:

```java
theLock = new Lock();
synchronized (theLock) {
    spaces++;
    if (spaces < 0)
        theLock.notify();
}
...
Scheduler Integration

Goals:

- No changes to VM or Compiler
- Programmer shall use familiar constructs

⇒ Intercept native Java synchronisation via code transformation

Example:

...  

\[
\text{synchronized (depositeLock) \{}
\text{spaces++;}
\text{if (spaces \leq 0)}
\text{depositeLock.notify();}
\}
\]

⇒

...  

\[
\text{scheduler.lock(depositeLock);} \\
\text{try \{} \\
\text{spaces++;}
\text{if (spaces \leq 0)}
\text{scheduler.mtnotify(depositeLock);} \\
\} \text{finally \{} \\
\text{scheduler.unlock(depositeLock);} \\
\}
\]

...  

Jörg Domaschka (Ulm University)  
Aspectix Research Group  
http://www.aspectix.org/
Open Issues

1. Last lock $\neq$ Thread termination
Open Issues

1. Last lock ≠ Thread termination

Pessimistic assumptions: All threads potentially conflict

⇒ More information for scheduler

⇒ Code analysis

Jörg Domaschka (Ulm University)  Aspectix Research Group  http://www.aspectix.org/
Open Issues

1. **Last lock $\neq$ Thread termination**

   - Primary
   - Secondary
   - $\text{lock}(x)$
   - $\text{unlock}(x)$
   - $\text{terminate}()$

2. **Pessimistic assumptions: All threads potentially conflict**
1. Last lock $\neq$ Thread termination

2. Pessimistic assumptions: All threads potentially conflict
Open Issues

1. Last lock $\not\equiv$ Thread termination

2. Pessimitic assumptions: All threads potentially conflict

⇒ More information for scheduler
Open Issues

1. Last lock ≠ Thread termination

2. Pessimitic assumptions: All threads potentially conflict

⇒ More information for scheduler
⇒ Code analysis
Goals:

- Additional information for the scheduler
- Provided at run-time
Code Analysis

Goals:
- Additional information for the scheduler
- Provided at run-time

Means:
- Extend code transformation by code analysis
- Insert additional code
- Generate static information

Assumptions:
- Deterministic application, correctly synchronized
- RMI environment: Fixed number of starting points
- No synchronized in loops
- All methods are final
- No recursion
Code Analysis

Goals:
- Additional information for the scheduler
- Provided at run-time

Means:
- Extend code transformation by code analysis
- Insert additional code
- Generate static information

Assumptions:
- Deterministic application, correctly synchronized
- RMI environment: Fixed number of starting points
- No synchronized in loops
- All methods are final
- No recursion
Environment:

- Limited number of starting points
- Acyclic execution path (no recursion, no loops)
- Limited number of synchronized statements
- Different paths may overlap
Last Lock Analysis

Environment:

- Limited number of starting points
- Acyclic execution path (no recursion, no loops)
- Limited number of synchronized statements
- Different paths may overlap

⇒ Cannot add `scheduler.lastLock()` statement
Last Lock Analysis

Environment:

- Limited number of starting points
- Acyclic execution path (no recursion, no loops)
- Limited number of synchronized statements
- Different paths may overlap

⇒ Cannot add scheduler.lastLock() statement
⇒ Static information: Which synchronized on which execution path
Last Lock Analysis

Environment:

- Limited number of starting points
- Acyclic execution path (no recursion, no loops)
- Limited number of synchronized statements
- Different paths may overlap

⇒ Cannot add scheduler.lastLock() statement
⇒ Static information: Which synchronized on which execution path
⇒ Added Code: Which synchronized is called
Last Lock Analysis

Environment:

- Limited number of starting points
- Acyclic execution path (no recursion, no loops)
- Limited number of synchronized statements
- Different paths may overlap

⇒ Cannot add scheduler.lastLock() statement
⇒ Static information: Which synchronized on which execution path
⇒ Added Code: Which synchronized is called

Implementation:

- ID per synchronized statements
- Scheduler keeps table per thread
Last Lock Analysis

Environment:

- Limited number of starting points
- Acyclic execution path (no recursion, no loops)
- Limited number of synchronized statements
- Different paths may overlap

⇒ Cannot add `scheduler.lastLock()` statement
⇒ Static information: Which synchronized on which execution path
⇒ Added Code: Which synchronized is called

Implementation:

- ID per synchronized statements
- Scheduler keeps table per thread

Branches:

- Not always called
- Inform scheduler
public void deposit(Object value) {
    if (value != null) {
        value = decrypt(value);
        scheduler.lock(3, depositeLock);
        try {
            ...
        } finally {
            scheduler.unlock(3, depositeLock);
        }
        scheduler.lock(4, fetchLock);
        try {
            ...
        } finally {
            scheduler.unlock(4, fetchLock);
        }
    } else {
        scheduler.ignore(3);
        scheduler.ignore(4);
    }
}
Lock Parameter Analysis

Goal: Mapping of mutex IDs to objects

- What will the synchronized parameter be?
- When is the synchronized parameter assigned the last time?

Realisation:

Add new call to scheduler:

```
scheduler.lockInfo(mutexID, mutex)
```

Constraints:

Only feasible for parameters without unpredictable changes

⇒ this, method local objects, final objects

Unfeasible otherwise

⇒ member, return value of a method call, global object
Lock Parameter Analysis

Goal: Mapping of mutex IDs to objects

- What will the synchronized parameter be?
- When is the synchronized parameter assigned the last time?

Realisation:

- Add new call to scheduler:
  
  `scheduler.lockInfo(mutexID, mutex)`
Lock Parameter Analysis

Goal: Mapping of mutex IDs to objects
- What will the synchronized parameter be?
- When is the synchronized parameter assigned the last time?

Realisation:
- Add new call to scheduler:
  scheduler.lockInfo(mutexID, mutex)

Constraints:
- Only feasible for parameters without unpredictable changes
  ⇒ this, method local objects, final objects
Lock Parameter Analysis

Goal: Mapping of mutex IDs to objects
- What will the synchronized parameter be?
- When is the synchronized parameter assigned the last time?

Realisation:
- Add new call to scheduler:
  \[ \text{scheduler.lockInfo}(	ext{mutexID}, \text{mutex}) \]

Constraints:
- Only feasible for parameters without unpredictable changes
  ⇒ this, method local objects, final objects
- Unfeasible otherwise
  ⇒ member, return value of a method call, global object
Object depositeLock = new Object();
final Object fetchLock = new Object();
...
public void deposit(Object value) {
    scheduler.lockInfo(4, fetchLock);
    if (value != null) {
        scheduler.lock(3, depositeLock);
        try {
            ...
        } finally {
            scheduler.unlock(3, depositeLock);
        }
        scheduler.lock(4, fetchLock);
        try {
            ...
        } finally {
            scheduler.unlock(4, fetchLock);
        }
    } else {
        scheduler.ignore(3);
        scheduler.ignore(4);
    }
    ...
}
Extending the MAT Algorithm

Last lock analysis
- After last lock primaries lose their status

Lock parameter analysis
- Secondaries can acquire locks
- iff no conflicts with primaries and preceding secondary
- Requires all future locks to be known
Loops:

- Allow synchronized in loops
- Unknown quantity of synchronized and mutexes
- In general: no lock parameter analysis possible
Relaxing Assumptions

Loops:
- Allow synchronized in loops
- Unknown quantity of synchronized and mutexes
- In general: no lock parameter analysis possible

**final methods:**
- Problem: Compile-time vs. run-time type
- Oblige developer to ensure match
  ⇒ Error prone
- Run-time checks
  ⇒ Overhead
Relaxing Assumptions

Loops:
- Allow synchronized in loops
- Unknown quantity of synchronized and mutexes
- In general: no lock parameter analysis possible

final methods:
- Problem: Compile-time vs. run-time type
  ⇒ Error prone
- Oblige developer to ensure match
  ⇒ Error prone
- Run-time checks
  ⇒ Overhead

Recursion:
- Use pessimistic algorithm?
Multithreading is essential for object replication systems
- Re-use of existing code
- Avoidance of deadlocks and idle time
- Client coordination with condition variables
- Efficient execution on multi-CPU hardware
Conclusion

Multithreading is essential for object replication systems
- Re-use of existing code
- Avoidance of deadlocks and idle time
- Client coordination with condition variables
- Efficient execution on multi-CPU hardware

Code analysis helps to reduce pessimism in scheduling
- Restrictions on code
- Last lock analysis
- Lock parameter analysis
Conclusion

Multithreading is essential for object replication systems

- Re-use of existing code
- Avoidance of deadlocks and idle time
- Client coordination with condition variables
- Efficient execution on multi-CPU hardware

Code analysis helps to reduce pessimism in scheduling

- Restrictions on code
- Last lock analysis
- Lock parameter analysis

Ongoing work:

- Implementation and evaluation of last lock and lock parameter analysis
- Evaluation: Overhead of processing information
- Sophisticated data-flow analysis
Thank you!

Questions?
Mandatory properties of replicated object implementation:

- Correct coordination of access to shared state with mutex locks
  - State modification only while holding corresponding mutex
  - No implicit (atomic variables) or wait-free synchronisation

- Piecewise determinism (PD):
  - Replica implementation must be determinisic
  - In practice: No local time stamps, random numbers, etc.
ADETS-MAT: How it works

Primary Threads
- Active Primary
- MutexWait Map
- CondWait Map

Secondary Threads
- PC (Primary Candidates)
- Non-PC

Key Functions:
- lock()
- wait()
- schedule()
- notify()
- nested invocation, yield()
- receive nested invocation reply

New Request
Execution of *active primary*:
- Scheduling identical to a single-active-thread approach
- `wait()`, blocking `lock()`: → suspended active thread
- nested invocation, `yield()`: → secondary thread (non-PC)

Selection of *active primary*:
1) Select suspended active thread
2) Select secondary thread (PC) according to message order
Execution of *secondary threads*:
- Incoming request creates and starts thread immediately
- Incoming invocation reply resumes waiting thread

*Secondary thread synchronisation*:
- lock() or wait(): suspend
- notify(), unlock(): thread continues, action recorded and deferred (executed as soon as thread becomes active primary)
- Wait can time out
- Concurrency between explicit notification and timeout
- Consistency in ADETS-MAT: Explicit timeout message
Experimental Evaluation

- Comparision with PDS and LSA: Simple Execution Patterns
  request: compute – lock – state update – unlock
- Bounded buffer with condition variables
Example 1: compute – lock – update – unlock
Example 2: Bounded buffer with condition variables

![Graph showing time/invocation (ms) vs. number of consumers for different processes.

Legend:
- SAT
- MAT
- SEQ

Data points for different numbers of consumers:
- 0 consumers
- 10 consumers
- 20 consumers
- 30 consumers
- 40 consumers (time/invocation (ms) values could be inferred from the graph.)]