Program Analysis and Synthesis of Parallel Systems

Roman Manevich  Ben-Gurion University
Three papers

1. *A Shape Analysis for Optimizing Parallel Graph Programs* [POPL’11]

2. *Elixir: a System for Synthesizing Concurrent Graph Programs* [OOPSLA’12]

3. *Parameterized Verification of Transactional Memories* [PLDI’10]
What’s the connection?

From analysis to language design

A Shape Analysis for Optimizing Parallel Graph Programs [POPL’11]

Elixir: a System for Synthesizing Concurrent Graph Programs [OOPSLA’12]

Similarities between abstract domains

Parameterized Verification of Transactional Memories [PLDI’10]

Creates opportunities for more optimizations.
Requires other analyses
What’s the connection?

From analysis to language design

A Shape Analysis for Optimizing Parallel Graph Programs [POPL’11]

Elixir: a System for Synthesizing Concurrent Graph Programs [OOPSLA’12]

Similarities between abstract domains

Creates opportunities for more optimizations. Requires other analyses

Parameterized Verification of Transactional Memories [PLDI’10]
A Shape Analysis for Optimizing Parallel Graph Programs

Dimitrios Prountzos$^1$
Keshav Pingali$^{1,2}$
Roman Manevich$^2$
Kathryn S. McKinley$^1$

1: Department of Computer Science, The University of Texas at Austin
2: Institute for Computational Engineering and Sciences, The University of Texas at Austin
Motivation

• Graph algorithms are ubiquitous

Computational biology

Social Networks

Computer Graphics

• **Goal:** Compiler analysis for optimization of parallel graph algorithms
Minimum Spanning Tree Problem

Diagram:
- Nodes: a, b, c, d, e, f, g
- Edges with weights:
  - c to d: 7
  - d to e: 1
  - e to f: 6
  - a to b: 5
  - b to c: 2
  - c to a: 3
  - b to g: 4

The minimum spanning tree includes edges with the lowest total weight.
Minimum Spanning Tree Problem
Boruvka’s Minimum Spanning Tree Algorithm

Build MST bottom-up

repeat {
    pick arbitrary node ‘a’
    merge with lightest neighbor ‘lt’
    add edge ‘a-lt’ to MST
} until graph is a single node
Parallelism in Boruvka

Build MST bottom-up
repeat {
    pick arbitrary node ‘a’
    merge with lightest neighbor ‘lt’
    add edge ‘a-lt’ to MST
} until graph is a single node
Non-conflicting iterations

Build MST bottom-up
repeat {
  pick arbitrary node ‘a’
  merge with lightest neighbor ‘lt’
  add edge ‘a-lt’ to MST
} until graph is a single node
Non-conflicting iterations

Build MST bottom-up

repeat {
    pick arbitrary node ‘a’
    merge with lightest neighbor ‘lt’
    add edge ‘a–lt’ to MST
} until graph is a single node
Build MST bottom-up

repeat {
    pick arbitrary node 'a'
    merge with lightest neighbor 'lt'
    add edge 'a-lt' to MST
} until graph is a single node
Optimistic parallelization in Galois

• Programming model
  – Client code has sequential semantics
  – Library of concurrent data structures
• Parallel execution model
  – Thread-level speculation (TLS)
  – Activities executed speculatively
• Conflict detection
  – Each node/edge has associated exclusive lock
  – Graph operations acquire locks on read/written nodes/edges
  – Lock owned by another thread → conflict → iteration rolled back
  – All locks released at the end
• Two main overheads
  – Locking
  – Undo actions
Generic optimization structure

- Program
- Program Analyzer
- Annotated Program
- Program Transformer
- Optimized Program
Overheads (I): locking

• Optimizations
  – Redundant locking elimination
  – Lock removal for iteration private data
  – Lock removal for lock domination

• $ACQ(P)$: set of *definitely acquired* locks per program point $P$

• Given method call $M$ at $P$:
  
  $$Locks(M) \subseteq ACQ(P) \implies\text{Redundant Locking}$$
Overheads (II): undo actions

forall (Node a : wl) {
  Set<Node> aNghbrs = g.neighbors(a);
  Node lt = null;
  for (Node n : aNghbrs) {
    minW,lt = minWeightEdge((a,lt), (a,n));
  }
  g.removeEdge(a, lt);
  Set<Node> ltNghbrs = g.neighbors(lt);
  for (Node n : ltNghbrs) {
    Edge e = g.getEdge(lt, n);
    Weight w = g.getEdgeData(e);
    Edge an = g.getEdge(a, n);
    if (an != null) {
      Weight wan = g.getEdgeData(an);
      if (wan.compareTo(w) < 0) {
        w = wan;
        g.setEdgeData(an, w);
      } else {
        g.addEdge(a, n, w);
      }
    }
    g.removeNode(lt);
  }
}

Program point \( P \) is failsafe if:
\[
\forall Q : \text{Reaches}(P, Q) \Rightarrow \text{Locks}(Q) \subseteq \text{ACQ}(P)
\]
Lockset analysis

- Redundant Locking
  - \( \text{Locks}(M) \subseteq \text{ACQ}(P) \)

- Undo elimination
  - \( \forall Q : \text{Reaches}(P,Q) \implies \text{Locks}(Q) \subseteq \text{ACQ}(P) \)

- Need to compute \( \text{ACQ}(P) \)

```java
GSet<Node> wl = new GSet<Node>();
wl.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }
    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
```
The optimization technically

• Each graph method \( m(\arg_1, \ldots, \arg_k, \text{flag}) \) contains optimization level flag
  • flag=LOCK – acquire locks
  • flag=UNDO – log undo (backup) data
  • flag=LOCK_UNOD – (default) acquire locks and log undo
  • flag=NONE – no extra work

• Example:
  \[ \text{Edge } e = g.\text{getEdge}(lt, n, \text{NONE}) \]
Analysis challenges

• The usual suspects:
  – Unbounded Memory $\rightarrow$ Undecidability
  – Aliasing, Destructive updates

• Specific challenges:
  – Complex ADTs: unstructured graphs
  – Heap objects are locked
  – Adapt abstraction to ADTs

• We use Abstract Interpretation \([CC’77]\)
  – Balance precision and realistic performance
Shape analysis overview

Concrete ADT Implementations in Galois library

HashMap-Graph

Tree-based Set

Graph { @rep nodes @rep edges ... }
Graph Spec

Set { @rep cont ... }
Set Spec

ADT Specifications

 Predicate Discovery

Shape Analysis

Boruvka.java

Optimized Boruvka.java
Graph<ND,ED> {

@rep set<Node> nodes
@rep set<Edge> edges

@locks(n + n.rev(src) + n.rev(src).dst + n.rev(dst) + n.rev(dst).src)

@op(
  nghbrs = n.rev(src).dst + n.rev(dst).src,
  ret = new Set<Node<ND>>(cont=nghbrs)
)

Set<Node> neighbors(Node n);
}

Assumption: Implementation satisfies Spec
Graph<ND,ED> {
  @rep set<Node> nodes
  @rep set<Edge> edges
  @locks(n + n.rev(src) + n.rev(src).dst + n.rev(dst) + n.rev(dst).src)
  @op(
    nghbrs = n.rev(src).dst + n.rev(dst).src,
    ret = new Set<Node<ND>>(cont=nghbrs)
  )
  Set<Node> neighbors(Node n);
}
Modeling ADTs

Graph<ND,ED> {

    @rep set<Node> nodes
    @rep set<Edge> edges

    @locks(n + n.rev(src) + n.rev(src).dst + n.rev(dst) + n.rev(dst).src)

    @op(
        nghbrs = n.rev(src).dst + n.rev(dst).src,
        ret = new Set<Node<ND>>(cont=nghbrs)
    )

    Set<Node> neighbors(Node n);
}

Graph Spec
Abstraction scheme

- Parameterized by set of *LockPaths*:
  \[ L(\text{Path}) \triangleq \forall o \cdot o \in \text{Path} \Rightarrow \text{Locked}(o) \]
  - Tracks subset of must-be-locked objects

- Abstract domain elements have the form:
  \[ \text{Aliasing-configs} \rightarrow 2^{\text{LockPaths}} \times \ldots \]
Joining abstract states

\[
\begin{align*}
&\ (x \neq y) \land L(y, nd) \end{align*}
\]

\[
\begin{align*}
&\ (x \neq y) \land L(y, nd) \land L(x, rev, src) \end{align*}
\]

\[
\begin{align*}
&\ (x = y) \land L(x, nd) \end{align*}
\]

\[
\begin{align*}
&\ (x \neq y) \land L(y, nd) \end{align*}
\]

\[
\begin{align*}
&\ (x = y) \land L(x, nd) \end{align*}
\]

\[
\begin{align*}
&\ (x \neq y) \land L(y, nd) \end{align*}
\]

\[
\begin{align*}
&\ (x = y) \land L(x, nd) \end{align*}
\]

\[
\begin{align*}
&\ \lor \end{align*}
\]

\[
\begin{align*}
&\ (x \neq y) \land L(y, nd) \end{align*}
\]

\[
\begin{align*}
&\ (x = y) \land L(x, nd) \end{align*}
\]

\[
\begin{align*}
&\ \lor \end{align*}
\]

\[
\begin{align*}
&\ (x \neq y) \land L(y, nd) \end{align*}
\]

\[
\begin{align*}
&\ (x = y) \land L(x, nd) \end{align*}
\]

\[
\begin{align*}
&\ \lor \end{align*}
\]

\[
\begin{align*}
&\ (x \neq y) \land L(y, nd) \end{align*}
\]

\[
\begin{align*}
&\ (x = y) \land L(x, nd) \end{align*}
\]


Aliasing is crucial for precision

May-be-locked does not enable our optimizations

#Aliasing-configs : small constant (≈6)
Example invariant in Boruvka

```
GSet<Node> wl = new GSet<Node>();
w1.addAll(g.getNodes());
GBag<Weight> mst = new GBag<Weight>();

foreach (Node a : wl) {
    Set<Node> aNghbrs = g.neighbors(a);
    Node lt = null;
    for (Node n : aNghbrs) {
        minW,lt = minWeightEdge((a,lt), (a,n));
    }

    g.removeEdge(a, lt);
    Set<Node> ltNghbrs = g.neighbors(lt);
    for (Node n : ltNghbrs) {
        Edge e = g.getEdge(lt, n);
        Weight w = g.getEdgeData(e);
        Edge an = g.getEdge(a, n);
        if (an != null) {
            Weight wan = g.getEdgeData(an);
            if (wan.compareTo(w) < 0)
                w = wan;
            g.setEdgeData(an, w);
        } else {
            g.addEdge(a, n, w);
        }
    }
    g.removeNode(lt);
    mst.add(minW);
    wl.add(a);
}
```

The immediate neighbors of $a$ and $lt$ are locked

(a ≠ lt)
∧ L(a) ∧ L(a.rev(src)) ∧ L(a.rev(dst))
∧ L(a.rev(src).dst) ∧ L(a.rev(dst).src)
∧ L(lt) ∧ L(lt.rev(dst)) ∧ L(lt.rev(src))
∧ L(lt.rev(dst).src) ∧ L(lt.rev(src).dst)
.....
Heuristics for finding LockPaths

• **Hierarchy Summarization (HS)**
  – $x.(\text{fld} \ )^*$
  – Type hierarchy graph acyclic $\rightarrow$ bounded number of paths

  – Preflow-Push:
    • $L(S.\text{cont}) \land L(S.\text{cont.nd})$
    • Nodes in set $S$ and their data are locked
Footprint graph heuristic

- **Footprint Graphs (FG)** \cite{Calcagno2007}:
  - All acyclic paths from arguments of ADT method to locked objects
  - \( x.(\text{fld} \mid \text{rev(fld)})^* \)
  - Delaunay Mesh Refinement:
    - \( L(S.\text{cont}) \land L(S.\text{cont}.\text{rev}(\text{src})) \land L(S.\text{cont}.\text{rev}(\text{dst})) \)
      - \( \land L(S.\text{cont}.\text{rev}(\text{src}).\text{dst}) \land L(S.\text{cont}.\text{rev}(\text{dst}).\text{src}) \)
    - Nodes in set \( S \) and all of their immediate neighbors are locked

- **Composition of HS, FG**
  - Preflow-Push: \( L(a.\text{rev}(\text{src}).\text{ed}) \)
Experimental evaluation

• Implement on top of TVLA
  – Encode abstraction by 3-Valued Shape Analysis [SRW TOPLAS’02]

• Evaluation on 4 Lonestar Java benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Analysis Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boruvka MST</td>
<td>6</td>
</tr>
<tr>
<td>Preflow-Push Maxflow</td>
<td>7</td>
</tr>
<tr>
<td>Survey Propagation</td>
<td>12</td>
</tr>
<tr>
<td>Delaunay Mesh Refinement</td>
<td>16</td>
</tr>
</tbody>
</table>

• Inferred all available optimizations
• # abstract states practically linear in program size
Impact of optimizations for 8 threads

- Boruvka MST: Baseline 250 sec, Optimized 86 sec (2.9x speedup)
- Delaunay Mesh Refinement: Baseline 60 sec, Optimized 10 sec (6x speedup)
- Survey Propagation: Baseline 50 sec, Optimized 10 sec (5x speedup)
- Preflow-Push Maxflow: Baseline 200 sec, Optimized 18 sec (11.4x speedup)

8-core Intel Xeon @ 3.00 GHz
Note 1

• How to map abstract domain presented so far to TVLA?
  – Example invariant: \((x \neq y \land L(y.nd)) \lor (x = y \land L(x.nd))\)
  – Unary abstraction predicate \(x(v)\) for pointer \(x\)
  – Unary non-abstraction predicate \(L[x.p]\) for pointer \(x\) and path \(p\)
  – Use partial join
  – Resulting abstraction similar to the one shown
Note 2

• How to come up with abstraction for similar problems?
  1. Start by constructing a manual proof
     • Hoare Logic
  2. Examine resulting invariants and generalize into a language of formulas
     • May need to be further specialized for a given program – interesting problem (machine learning/refinement)

– How to get sound transformers?
Note 3

• How did we avoid considering all interleavings?
• Proved non-interference side theorem
Elixir: A System for Synthesizing Concurrent Graph Programs

Dimitrios Prountzos\textsuperscript{1}
Roman Manevich\textsuperscript{2}
Keshav Pingali\textsuperscript{1}

\textsuperscript{1}. The University of Texas at Austin
\textsuperscript{2}. Ben-Gurion University of the Negev
Goal

Allow programmer to easily implement correct and efficient parallel graph algorithms

• Graph algorithms are ubiquitous
  Social network analysis, Computer graphics, Machine learning, …

• Difficult to parallelize due to their irregular nature

• Best algorithm and implementation usually
  – Platform dependent
  – Input dependent

• Need to easily experiment with different solutions

• Focus: Fixed graph structure
  • Only change labels on nodes and edges
  • Each activity touches a fixed number of nodes
Example: Single-Source Shortest-Path

- **Problem Formulation**
  - Compute shortest distance from source node $S$ to every other node

- **Many algorithms**
  - Bellman-Ford (1957)
  - Dijkstra (1959)
  - Chaotic relaxation (Miranker 1969)
  - Delta-stepping (Meyer et al. 1998)

- **Common structure**
  - Each node has label $dist$ with known shortest distance from $S$

- **Key operation**
  - relax-edge($u, v$)
    - if $dist(A) + W_{AC} < dist(C)$
      - $dist(C) = dist(A) + W_{AC}$
Dijkstra’s algorithm

Scheduling of relaxations:
• Use *priority queue of nodes*, ordered by label *dist*
• Iterate over nodes *u* in priority order
• On each step: relax all neighbors *v* of *u*
  – Apply *relax-edge* to all *(u,v)*
Chaotic relaxation

- Scheduling of relaxations:
- Use unordered *set of edges*
- Iterate over edges \((u, v)\) in any order
- On each step:
  - Apply relax-edge to edge \((u, v)\)
Insights behind Elixir

What should be done

How it should be done

Operators

Parallel Graph Algorithm

Unordered/Ordered algorithms

Order activity processing

Static Schedule

Dynamic Schedule

Schedule

Identify new activities

Operator Delta

"TAO of parallelism"
PLDI 2011
Insights behind Elixir

Parallel Graph Algorithm

Operators

Schedule

Order activity processing

Identify new activities

Static Schedule

Dynamic Schedule

Dijkstra-style Algorithm

q = new PrQueue
q.enqueue(SRC)
while (! q.empty) {
    a = q.dequeue
    for each e = (a,b,w) {
        if dist(a) + w < dist(b) {
            dist(b) = dist(a) + w
            q.enqueue(b)
        }
    }
}
Contributions

- **Language**
  - Operators/Schedule separation
  - Allows exploration of implementation space

- **Operator Delta Inference**
  - Precise Delta required for efficient fixpoint computations

- **Automatic Parallelization**
  - Inserts synchronization to atomically execute operators
  - Avoids data-races / deadlocks
  - Specializes parallelization based on scheduling constraints
SSSP in Elixir

Graph [ 
    nodes(node: Node, dist: int) 
    edges(src: Node, dst: Node, wt: int) 
] 
relax = [ 
    nodes(node a, dist ad) 
    nodes(node b, dist bd) 
    edges(src a, dst b, wt w) 
    bd > ad + w ] → 
[ bd = ad + w ] 

sssp = iterate relax >> schedule
Operators

Graph [
  nodes(node : Node, dist : int)
  edges(src : Node, dst : Node, wt : int)
]

relax = [ nodes(node a, dist ad)
  nodes(node b, dist bd)
  edges(src a, dst b, wt w)
  bd > ad + w ] ➔
  [ bd = ad + w ]

sssp = iterate relax >> schedule

Cautious by construction – easy to generalize

Redex pattern

Guard

Update

ad

w

a

bd

if bd > ad + w

b

ad+w

ad

w

a

b
Graph [ 
  nodes(node : Node, dist : int)
  edges(src : Node, dst : Node, wt : int)
]

relax = [ nodes(node a, dist ad)
         nodes(node b, dist bd)
         edges(src a, dst b, wt w)
         bd > ad + w ] ➔
         [ bd = ad + w ]

sssp = iterate relax >> schedule
Scheduling examples

Graph [ 
  nodes(node : Node, dist : int) 
  edges(src : Node, dst : Node, wt : int) 
]

relax = [ 
  nodes(node a, dist ad) 
  nodes(node b, dist bd) 
  edges(src a, dst b, wt w) 
  bd > ad + w ] ➔ 
[ bd = ad + w ]

sssp = iterate relax ➔ schedule

Dijkstra-style
metric ad ➔ group b

Locality enhanced Label-correcting

group b ➔ unroll 2 ➔ approx metric ad

q = new PrQueue
q.enqueue(SRC)
while (! q.empty) {
  a = q.dequeue
  for each e = (a,b,w) {
    if dist(a) + w < dist(b) {
      dist(b) = dist(a) + w
      q.enqueue(b)
    }
  }
}


Operator Delta Inference

Parallel Graph Algorithm

Operators Schedule

Order activity processing Identify new activities

Static Schedule Dynamic Schedule
Identifying the delta of an operator

relax$_1$
Delta Inference Example

\[ \begin{align*}
\text{relax}_2 & \quad w_2 \\
\text{relax}_1 & \quad w_1 
\end{align*} \]

- Assume \((da + w_1 < db)\)
- Assume \(\neg (dc + w_2 < db)\)
- \(db\_post = da + w_1\)
- Assert \(\neg (dc + w_2 < db\_post)\)

Query Program

SMT Solver

(c,b) does not become active
Delta inference example – active

- Assume $(da + w_1 < db)$
- Assume $\neg (db + w_2 < dc)$
- $db_{post} = da + w_1$
- Assert $\neg (db_{post} + w_2 < dc)$

Apply relax on all outgoing edges $(b,c)$ such that:
- $dc > db + w_2$
- and $c \not\equiv a$

Query Program

SMT Solver
Influence patterns

- a \rightarrow b=c \rightarrow d
- c \rightarrow a=d \rightarrow b
- a=c \rightarrow b
- a \rightarrow b=d
- a=c \rightarrow b=d
- a=d \rightarrow b=c
System architecture

Algorithm Spec

→ Elixir

Synthesize code
Insert synchronization

→ C++ Program

Galois/OpenMP Parallel Runtime

Parallel Thread-Pool
Graph Implementations
Worklist Implementations
Experiments

<table>
<thead>
<tr>
<th>Explored Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Grouping</strong></td>
</tr>
<tr>
<td><strong>Unrolling</strong></td>
</tr>
<tr>
<td><strong>Dynamic Scheduler</strong></td>
</tr>
</tbody>
</table>

... 

Compare against hand-written parallel implementations
SSSP results

- 24 core Intel Xeon @ 2 GHz
- USA Florida Road Network (1 M nodes, 2.7 M Edges)

Group + Unroll improve locality
Breadth-First search results

Scale-Free Graph
1 M nodes, 8 M edges

USA road network
24 M nodes, 58 M edges
Conclusion

- **Graph algorithm = Operators + Schedule**
  - Elixir language: imperative operators + declarative schedule

- Allows exploring implementation space
- Automated reasoning for efficiently computing fixpoints
- Correct-by-construction parallelization
- Performance competitive with hand-parallelized code
Parameterized Verification of Software Transactional Memories

Michael Emmi  Rupak Majumdar
Roman Manevich
Motivation

• **Transactional memories [Herlihy ‘93]**
  – Programmer writes code w. coarse-grained atomic blocks
  – Transaction manager takes care of conflicts providing illusion of sequential execution

• **Strict serializability** – correctness criterion
  – Formalizes “illusion of sequential execution”

• **Parameterized verification**
  – Formal proof for given implementation
  – For every number of threads
  – For every number of memory objects
  – For every number and length of transactions
STM terminology

- **Statements**: reads, writes, commit, abort
- **Transaction**: reads and writes of variables followed by **commit** (committing transaction) or **abort** (aborting transaction)
- **Word**: interleaved sequence of transactions of different threads
- **Conflict**: two statements conflict if
  - One is a read of variable X and other is a commit of a transaction that writes to X
  - Both are commits of transactions that write to X
Safety property: strict serializability

• There is a serialization for the committing threads such that order of conflicts is preserved
• Order of non-overlapping transactions remains the same
Safety property: strict serializability

- There is a serialization for the committing threads such that order of conflicts is preserved
- Order of non-overlapping transactions remains the same
- Example word:
  \[(\text{rd} \ X \ t_1), (\text{rd} \ Y \ t_2), (\text{wr} \ X \ t_2), (\text{commit} \ t_2), (\text{commit} \ t_1)\]

\[
\Rightarrow
\]

Can be serialized to:
\[(\text{rd} \ X \ t_1), (\text{commit} \ t_1), (\text{rd} \ Y \ t_2), (\text{wr} \ X \ t_2), (\text{commit} \ t_2)\]
Main results

• First automatic verification of \emph{strict serializability} for \emph{transactional memories} – TPL, DSTM, TL2

• New proof technique:
  – Template-based invisible invariant generation
  – Abstract checking algorithm to check inductive invariants

Challenging – requires reasoning on both universal and existential properties
Outline

• Strict serializability verification approach
• Automating the proof
• Experiments
• Conclusion
• Related work
Proof roadmap 1

Goal: prove model $M$ is strictly serializable

1. Given a strict-serializability reference model $RSS$ reduce checking strict-serializability to checking that $M$ refines $RSS$

2. Reduce refinement to checking safety
   - Safety property SIM: whenever $M$ can execute a statement so can $RSS$
   - Check SIM on product system $M \times RSS$
Proof roadmap 2

3. Model STMs $M$ and $RSS$ in first-order logic
   - TM models use set data structures and typestate bits

4. Check safety by generating strong enough candidate inductive invariant and checking inductiveness
   - Use observations on structure of transactional memories
   - Use sound first-order reasoning
Reference strict serializability model

- Guerraoui, Singh, Henzinger, Jobstmann [PLDI’08]
- **RSS**: Most liberal specification of strictly serializable system
  - Allows largest language of strictly-serializable executions
- **M** is strictly serializable iff every word of **M** is also a word of **RSS**
  - \( \text{Language}(M) \subseteq \text{Language}(RSS) \)
  - **M** refines **RSS**
Modeling transactional memories

- $M_{n,k} = (\text{predicates, actions})$
  - Predicate: ranked relation symbol $p(t), q(t,v), ...$
  - Binary predicates used for sets so instead of $rs(t,v)$ I’ll write $v \in rs(t)$
  - Action:
    
    $a(t,v) = \text{if } pre(a) \text{ then } p'(v) = ..., q'(u,v) = ...$

- Universe = set of $k$ thread individuals and $n$ memory individuals

- State $S$ = a valuation to the predicates
Reference model (RSS) predicates

- **Typestates:**
  - RSS.finished(t), RSS.started(t), RSS.pending(t), RSS.invalid(t)

- **Read/write sets**
  - RSS.rs(t,v), RSS.ws(t,v)

- **Prohibited read/write sets**
  - RSS.prs(t,v), RSS.pws(t,v)

- **Weak-predecessor**
  - RSS.wp(t₁,t₂)
DSTM predicates

• Typestates:
  – DSTM.finished(t), DSTM.validated(t), DSTM.invalid(t), DSTM.aborted(t)

• Read/own sets
  – DSTM.rs(t,v), DSTM.os(t,v)
RSS commit(t) action

if $\neg$RSS.invalid(t) $\land$ $\neg$RSS.wp(t,t) then

$\forall t_1, t_2 . \text{RSS.wp}'(t_1, t_2) \leftrightarrow t_1 \neq t \land t_2 \neq t \land$

(RSS.wp(t_1, t_2) $\lor$ RSS.wp(t, t_2) $\land$

(RSS.wp(t_1, t) $\lor$

$\exists v . v \in \text{RSS.ws}(t_1) \land v \in \text{RSS.ws}(t)$))

...
DSTM commit(t) action

if DSTM.validated(t) then
\[ \forall t_1 . \ DSTM.v\text{alidated}'(t_1) \iff t_1 \neq t \land \]
\[ \neg \exists v . \ v \in DSTM.rs(t_1) \land v \in DSTM.os(t_1) \]
...

read-own conflict
FOTS states and execution

- state $S_1$
- thread $t_1$
- thread $t_2$
- memory location $v$
- DSTM

$\text{rd } v \rightarrow t_1$
FOTS states and execution

State S2

DSTM.started

t1

t2

DSTM.rs

v

Predicate evaluation
DSTM.started(t1)=1

Predicate evaluation
DSTM.rs(t1,v)=1

DSTM

rd v t1
FOTS states and execution

DSTM

state S3

DSTM.started

DSTM.ws

DSTM.rs

v

t1

t2

wr v t2
Product system

• The product of two systems: $A \times B$
• Predicates = $A$.predicates $\cup$ $B$.predicates
• Actions =
  commit(t) = \{ if $(A$.pre $\land B$.pre) then ... \}
  rd(t,v) = \{ if $(A$.pre $\land B$.pre) then ... \}
  ...
• $M$ refines RSS iff on every execution SIM holds: $\land_{a \in \text{action}} M$.pre(a) $\rightarrow$ RSS.pre(a)
Checking DSTM refines RSS

• The only precondition in RSS is for commit(t)

• We need to check SIM =
  \( \forall t. \text{DSTM.validated}(t) \rightarrow \neg \text{RSS.invalid}(t) \land \neg \text{RSS.wp}(t,t) \) holds for DSTM \( \times \) RSS for all reachable states

• Proof rule:

\[
\begin{align*}
\text{DSTM} \times \text{RSS} & \not\models \text{SIM} \quad \triangleright \\
\hline
\text{DSTM refines RSS}
\end{align*}
\]

how do we check this safety property?
Checking safety by invisible invariants

• How do we prove that property $\psi$ holds for all reachable states of system $M$?
• Pnueli, Ruah, Zuck [TACAS’01]
• Come up with inductive invariant $\varphi$ that contains reachable states of $M$ and strengthens $\text{SIM}$:

\[
\begin{align*}
\textbf{I1: } & \text{ Initial } \models \varphi & \textbf{I2: } & \varphi \land \text{ transition } \models \varphi' & \textbf{I3: } & \varphi \models \psi \\
\hline
& \hfill M \models \psi
\end{align*}
\]
Strict serializability proof rule

I1: Initial ⊨ φ  I2: φ ∧ transition ⊨ φ'  I3: φ ⊨ SIM

DSTM \times RSS \models SIM

Proof roadmap:

1. Divine candidate invariant φ
2. Prove I1, I2, I3
Two challenges

Proof roadmap:
1. Divine candidate invariant $\varphi$
2. Prove $I_1$, $I_2$, $I_3$

But

how do we find a candidate $\varphi$?
infinite space of possibilities

given candidate $\varphi$ how do we check the proof rule?
checking $A \vDash B$ is undecidable for first-order logic
Our solution

Proof roadmap:
1. Divine candidate invariant \( \varphi \)
2. Prove \( I_1, I_2, I_3 \)
3. \( \text{utilize insights on transactional memory implementations} \)

\[ I_1: \text{Initial } \models \varphi \quad I_2: \varphi \land \text{transition } \models \varphi' \quad I_3: \varphi \models \text{SIM} \]

\[ \text{DSTM } \times \text{RSS } \models \text{SIM} \]

But

how do we find a candidate \( \varphi \)?
use templates and iterative weakening

given candidate \( \varphi \) how do we check the proof rule?
use abstract checking
Invariant for DSTM×RSS

P1: \( \forall t, t_1 . \text{RSS}.wp(t, t_1) \land \neg \text{RSS}.invalid(t) \land \neg \text{RSS}.pending(t) \Rightarrow \exists v . v \in \text{RSS}.ws(t_1) \land v \in \text{RSS}.ws(t) \)

P2: \( \forall t, v . v \in \text{RSS}.rs(t) \land \neg \text{DSTM}.aborted(t) \Rightarrow v \in \text{DSTM}.rs(t) \)

P3: \( \forall t, v . v \in \text{RSS}.ws(t) \Rightarrow v \in \text{DSTM}.os(t) \)

P4: \( \forall t . \text{DSTM}.validated(t) \Rightarrow \neg \text{RSS}.wp(t, t) \)

P5: \( \forall t . \text{DSTM}.validated(t) \Rightarrow \neg \text{RSS}.invalid(t) \)

P6: \( \forall t . \text{DSTM}.validated(t) \Rightarrow \neg \text{RSS}.pending(t) \)

Inductive invariant involving only RSS – can use for all future proofs
Templates for DSTM×RSS

P1: \( \forall t, t_1 . \Phi_1(t, t_1) \land \neg \Phi_2(t) \land \neg \Phi_3(t) \Rightarrow \exists v . v \in \Phi_4(t_1) \land v \in \Phi_5(t) \)

P2: \( \forall t, v . v \in \Phi_1(t) \land \neg \Phi_2(t) \Rightarrow v \in \Phi_3(t) \)

P3: \( \forall t, v . v \in \Phi_1(t) \land \neg \Phi_2(t) \Rightarrow v \in \Phi_3(t) \)

P4: \( \forall t . \Phi_1(t) \Rightarrow \neg \Phi_2(t, t) \)

P5: \( \forall t . \Phi_1(t) \Rightarrow \neg \Phi_2(t) \)

P6: \( \forall t . \Phi_1(t) \Rightarrow \neg \Phi_2(t) \)
Templates for DSTM × RSS

\[ \forall t, t_1 . \Phi_1(t, t_1) \land \neg \Phi_2(t) \land \neg \Phi_3(t) \Rightarrow \exists v . v \in \Phi_4(t_1) \land v \in \Phi_5(t) \]

\[ \forall t, v . v \in \Phi_1(t) \land \neg \Phi_2(t) \Rightarrow v \in \Phi_3(t) \]

\[ \forall t . \Phi_1(t) \Rightarrow \neg \Phi_2(t, t) \]

Why templates?
• Makes invariant separable
• Controls complexity of invariants
• Adding templates enables refinement
Mining candidate invariants

• Use predefined set of templates to specify structure of candidate invariants
  – $\forall t, v \cdot \Phi_1 \land \Phi_2 \rightarrow \Phi_3$
  – $\Phi_1$, $\Phi_2$, $\Phi_3$ are predicates of $M$ or their negations
  – Existential formulas capturing 1-level conflicts
    $\exists v \cdot v \in \Phi_4(t_1) \land v \in \Phi_5(t_2)$

• Mine candidate invariants from concrete execution
Iterative invariant weakening

\[ I_1: \text{Initial} \models \varphi \quad I_2: \varphi \land \text{transition} \models \varphi' \quad I_3: \varphi \models \text{SIM} \]

\[ \text{DSTM} \times \text{RSS} \models \text{SIM} \]

- Initial candidate invariant \( C_0 = P_1 \land P_2 \land \ldots \land P_k \)
- Try to prove \( I_2: \varphi \land \text{transition} \models \varphi' \)
  \[ C_1 = \{ P_i \mid \text{I0} \land \text{transition} \models P_i \text{ for } P_i \in \text{I0} \} \]
- If \( C_1 = C_0 \) then we have an inductive invariant
- Otherwise, compute
  \[ C_2 = \{ P_i \mid C_1 \land \text{transition} \models P_i \text{ for } P_i \in C_1 \} \]
- Repeat until either
  - found inductive invariant – check \( I_3: C_k \models \text{SIM} \)
  - Reached top \( \{ \} \) – trivial inductive invariant
Weakening illustration
Abstract proof rule

\[ \text{I1: Initial } \models \varphi \quad \text{I2: } \varphi \land \text{transition } \models \varphi' \quad \text{I3: } \varphi \models \text{SIM} \]

\[ \text{DSTM } \times \text{RSS } \models \text{SIM} \]

\[ \text{I1: } \alpha(\text{Initial}) \subseteq \varphi \quad \text{I2: } \text{abs_transition}(\alpha(\varphi)) \subseteq \varphi' \quad \text{I3: } \alpha(\varphi) \subseteq \text{SIM} \]

\[ \text{DSTM } \times \text{RSS } \models \text{SIM} \]
Conclusion

• Novel invariant generation using templates – extends applicability of invisible invariants
• Abstract domain and reasoning to check invariants without state explosion
• Proved strict-serializability for TPL, DSTM, TL2 – BLAST and TVLA failed
## Verification results

<table>
<thead>
<tr>
<th>property</th>
<th>TPL</th>
<th>DSTM</th>
<th>TL2</th>
<th>RSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bound for invariant gen.</td>
<td>(2,1)</td>
<td>(2,1)</td>
<td>(2,1)</td>
<td>(2,1)</td>
</tr>
<tr>
<td>No. cubes</td>
<td>8</td>
<td>184</td>
<td>344</td>
<td>7296</td>
</tr>
<tr>
<td>Bounded time</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>Invariant mining time</td>
<td>6</td>
<td>13</td>
<td>26</td>
<td>57</td>
</tr>
<tr>
<td>#templates</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>#candidates</td>
<td>22</td>
<td>53</td>
<td>97</td>
<td>19</td>
</tr>
<tr>
<td>#proved</td>
<td>22</td>
<td>30</td>
<td>68</td>
<td>14</td>
</tr>
<tr>
<td>#minimal</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>avg, time per invariant</td>
<td>3.7</td>
<td>20.3</td>
<td>36</td>
<td>43.4</td>
</tr>
<tr>
<td>avg. abs. size</td>
<td>31.7</td>
<td>256.9</td>
<td>1.19k</td>
<td>2.86k</td>
</tr>
<tr>
<td>Total time</td>
<td>3.5m</td>
<td>54.3m</td>
<td>129.3m</td>
<td>30.9m</td>
</tr>
</tbody>
</table>
Insights on transactional memories

• Transition relation is symmetric – thread identifiers not used
\[ p'(t,v) \leftrightarrow \ldots \forall t_1 \ldots \exists t_2 \]

• Executing thread \( t \) interacts only with arbitrary thread or conflict-adjacent thread

• Arbitrary thread:
\[ \exists v . \ v \in TL2.rs(t_1) \lor v \in TL2.ws(t_1) \]

• Conflict adjacent:
\[ \exists v . \ v \in DSTM.rs(t_1) \lor v \in DSTM.ws(t) \]
Conflict adjacency

\[ \exists v \cdot v \in rs(t) \land v \in DSTM.ws(t_2) \]

\[ \exists v \cdot v \in ws(t_1) \land v \in DSTM.ws(t_2) \]
Conflict adjacency
Related work

• Reduction theorems
  Guerarroui et al. [PLDI’08, CONCUR’08]

• Manually-supplied invariants – fixed number of threads and variables + PVS
  Cohen et al. [FMCAS’07]

• Predicate Abstraction + Shape Analysis
  – SLAM, BLAST, TVLA
Related work

• Invisible invariants
  Arons et al. [CAV’01] Pnueli et al. [TACAS’01]

• Templates – for arithmetic constraints

• Indexed predicate abstraction
  Shuvendu et al. [TOCL’07]

• Thread quantification
  Berdine et al. [CAV’08]
  Segalov et al. [APLAS’09]
Thank You!