Thread-Modular Shape Analysis

Josh Berdine  
Byron Cook  
Alexey Gotsman  
Tal Lev-Ami  
Roman Manevich  
G. Ramalingam  
Mooly Sagiv

Michal Segalov

MSR Cambridge  
MSR Cambridge  
University of Cambridge  
Tel Aviv University  
UCLA  
MSR India  
Tel Aviv University  
UC Berkeley  
Cadence Berkeley  
Tel Aviv University
Programs and Properties

- Concurrent programs
- Unbounded number of threads
  - parametric systems
- Unbounded number of objects
- Pointers and destructive updates

- Memory safety
  - Absence of null dereferences
  - Absence of memory leaks
- Preservation of data structure invariants
- Linearizability
- User-specified invariants
Concurrent Set [M. Maged SPAA’02]

```java
remove(key) {
    while (true) {
        <prev, cur, next, found> = locate(key)
        if (!found) return false;
        if (CAS(prev.next, <0, cur>, <0, next>))
            DeleteNode(curr2);
        if (!CAS(cur.next, <0, next>, <1, next>))
            continue;
        else  locate(key);
    }
}

add(node) {
    while (true) {
        <prev, cur, next, found> = locate(node.key)
        if (!found) return false;
        node.next = cur
        if (CAS(prev.next, <0, cur>, <0, node>))
            return true;
    }
}

locate(key) {
    restart: pred = Head;
    <tmp, curr> = pred.next;
    while (true) {
        if (curr == null) return <null, null, null, false>;
        <cmark, next> = curr.next;
        ckey = curr.key;
        if (pred.next != <0, curr>) goto restart;
        if (!cmark) {
            if (ckey >= key) return <prev, curr, next, (key == ckey) >
                pred = curr;
        } else  if (CAS(pred.next, <0, curr>, <0, next>)) DeleteNode(curr);
        else goto restart; }
    curr = next;
}
```
Concurrent Set [M. Maged SPAA’02]

remove(key) {
    while (true) {
        <prev2, cur2, next2, found> = locate(key)
        if (!found) return false;
        if (CAS(prev2.next, <0, curr2>, <0, next2>))
            DeleteNode(cur2);
        if (!CAS(cur2.next, <0, next2>, <1, next2>))
            continue;
        else locate(key);
    }
}

add(node1) {
    while (true) {
        <prev, cur, next, found> = locate(node1.key)
        if (found) return false;
        node1.next = curl
        if (CAS(prev1.next, <0, curl>, <0, node1>))
            return true;
        else continue;
    }
}

⚠️ A memory leak
What is the bug?

- A node is removed before it is marked

```java
remove(key) {
    while (true) {
        <prev, cur, next, found> = locate(key)
        if (!found) return false;
        if (!CAS(cur.next, <0, next>, <1, next>))
            continue;
        if (CAS(prev.next, <0, cur>, <1, next>))
            DeleteNode(cur);
        else locate(key);
    }
}
```
add(node1) {
    while (true) {
        <prev1, cur1, next1, found> = locate(node1.key)
        if (!found) return false;
        if (!CAS(prev1.next, <0, next1>, <0, node1>))
            return true;
        node1.next = curr1
        if (CAS(prev1.next, <0, next1>, <0, node1>))
            return true;
    }
}

remove(key) {
    while (true) {
        <prev2, cur2, next2, found> = locate(key)
        if (!found) return false;
        if (!CAS(cur2.next, <0, next2>, <1, next2>))
            continue;
        if (CAS(prev2.next, <0, cur2>, <0, next2>))
            DeleteNode(curr2);
        else locate(key);
    }
}

The CAS fails because of the markbit
Captured Invariants

- No memory leaks
  - Every “dangling” pointer is pointed-to by some thread reachable from `Head`, or has been returned by some remove method
- After a successful add, `prev` is reachable from `Head`, the node inserted is pointed-to by `prev` and it points to `curr`
- Only a single node can be added/removed by each operation
- An outgoing edge of a marked node is immutable
Challenges

- Develop an analysis which automatically proves interesting properties of concurrent heap-manipulating programs
  - Concurrency is challenging
  - The global nature of the heap
- Designing the right abstraction
- Developing effective transformers
  - Sound proof rules for atomic statements
A Singleton Buffer

Boolean empty = true;
Object b = null;

produce() {
1: Object p = new();
2: await (empty) then {
   b = p;
   empty = false;
}
3:
}

consume() {
Object c;
4: await (!empty) then {
   c = b;
   empty = true;
}
5: use(c);
6: dispose(c);
7:
}
State Space Exploration

- Enumerate all interleavings
- Check the properties
Partial State Space Exploration 1 consumer/2 producers

2: P1: await empty then
   {b = p1; empty=false;}

2: P2: await empty then
   {b = p2; empty=false;}

4: C1: await empty then
   {c1 = b; empty=true;}

empty b p1 2: p2 2: c1 4: C1

!empty b p1 3: p2 2: c1 4: C1

4: C1: await empty then
   {c1 = b; empty=true;}

empty b p1 3: p2 2: c1 5: C1

!empty b p1 3: p2 2: c1 5: C1

4: C1: await empty then
   {c1 = b; empty=true;}

empty b p1 2: p2 3: c1 5: C1

!empty b p1 2: p2 3: c1 5: C1

4: C1: await empty then
   {c1 = b; empty=true;}

empty b p1 2: p2 3: c1 5: C1

!empty b p1 2: p2 3: c1 5: C1

4: C1: await empty then
   {c1 = b; empty=true;}
State Space Explosion (bounded number of threads)

Exponential Blowup

# Threads (Cons & Prod)

# States

0 5,000 10,000 15,000 20,000 25,000 30,000 35,000

2 4 6 8
Plan

- Thread-modular analysis
- Semi-thread-modular analysis
- Unbounded number of threads
- Empirical results
Thread-Modular Analysis

- Abstract away the correlations between local states of different threads
  - No correlations between program counters
  - Cartesian Abstraction
- Information maintained
  - Correlations between the local state and global state of each thread
- “The quadratic cost of computing transformers can be greatly reduced…”
  [Flanagan & Qadeer SPIN, 2003]
- Naturally handles unbounded number of threads
Thread-Modular Abstraction
not all combinations are feasible
Partial Abstract Interpretation

4: C1: await !empty then {
    \( c_1 = b; \) empty=true;
}

4: C2: await !empty then {
    \( c_2 = b; \) empty=true;
}

5: C1: use(c_1); dispose(c_1)

5: C2: use(c_2); dispose(c_2)

Potential Double Free!!!
A Singleton Buffer

Boolean empty = true;
Object b = null;

produce() {
1: Object p = new();
2: await (empty) then {
    b = p;
    empty = false;
}
3:
}

consume() {
Object c;
4: await (!empty) then {
    c = b; b=null;
    empty = true;
}
5: use(c);
6: dispose(c);
7:
}
Thread-Modular Analysis

- Abstract away the correlations between local states of different threads
  - No correlations between program counters
  - Cartesian Abstraction
- Information maintained
  - Correlations between the local state of each thread and the global state
- Scales with the number of threads
- Handles unbounded number of threads
- But limited precision
Increasing Precision

- Enforce program restrictions
  - Limited aliasing
  - Ownership relations [Boyapati et. al. OOPSLA’02]
  - Limited concurrency
- Enhanced analysis
  - Global instrumentation
  - Separation Domains [Gotsman et. al. PLDI’07]
  - Semi-Thread Modular Analysis [Berdine et. al. CAV’08, Segalov et. al., TR]
{  
  PIRP irp;  
  LIST_ENTRY listHead,*entry;  
  KeInitializeIrql();  
  InitializeListHead(&listHead);  
  KeAcquireSpinLock(&DeviceExtension->SpinLock, &irql);  
  do {  
    irp = KeyboardClassDequeueReadByFileObject(DeviceExtension, FileObject);  
    if (irp) {  
      irp->IoStatus.Status = STATUS_CANCELLED;  
      irp->IoStatus.Information = 0;  
      InsertTailList (&listHead, &irp->Tail.Overlay.ListEntry);  
    }  
  } while (irp != NULL);  
  KeReleaseSpinLock(&DeviceExtension->SpinLock, irql);  
  //  
  // Complete these irps outside of the spin lock  
  //  
  while (!IsListEmpty (&listHead)) {  
    entry = RemoveHeadList (&listHead);  
    irp = CONTAINING_RECORD (entry, IRP, Tail.Overlay.ListEntry);  
  }  
}
Thread-Modular Analysis

Non-disjoint resource invariants
[the rest of this talk]
Fine-grained concurrency

Separated resource invariants
[Gotsman et al., PLDI 07]
Coarse-grained concurrency

Single global resource invariant
[Flanagan & Qadeer, SPIN 03]
Thread Quantification for Concurrent Shape Analysis
J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, M. Sagiv
CAV’08

Semi-Thread-Modular Analysis
M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, M. Sagiv
Main Results

- A refinement of thread-modular analysis
  - Not fully modular
- Precise enough to prove properties of fine-grained concurrent programs
  - Were not automatically proved before
- Two effective methods for efficiently computing transformers
  - Summarizing Effects
  - Summarizing Abstraction
  - On a concurrent set imp. speedup is x34!
Semi-Thread-Modular Analysis

- Abstract away correlations between local states of more than two threads
- Information maintained
  - Correlations between the local state of each thread and the global state
  - May-correlations between local states of every pair of threads
    - Not necessarily symmetric
Semi-Thread-Modular Abstraction
Semi-Thread-Modular Concretization
Worst-Case Complexity

- Full state analysis
  - Shared state – $G$, Local state – $L_{tid}$
  - State space: $\mathcal{O}(G \times L_1 \times \ldots \times L_n)$
  - #states: $O(|G| \cdot |L|^n)$

- Thread-modular analysis
  - State space: $\mathcal{O}(G \times L_1) \times \ldots \times \mathcal{O}(G \times L_n)$
  - #states: $O(n \cdot |G| \cdot |L|)$

- Semi-thread-modular analysis
  - State space: $\mathcal{O}(G \times L_1 \times L_2) \times \ldots \times \mathcal{O}(G \times L_{n-1} \times L_n)$
  - #states: $O(n \cdot |G| \cdot |L|^2)$
Point-wise Transformer

6: C1: dispose(c₁)

6: C1: dispose(c₁)

7:
Point-wise Transformer

6: C1: dispose(c₁)

Is this command safe in this configuration?
Missing information on c₁
Unknown effect on b
Most-Precise Transformer

concrete element \[\gamma\] abstract element

concrete element

abstract element

operational semantics
statement st

most-precise abstract semantics [CC’79]
statement st

abstract element
Most-precise transformer

2: P1: await (empty) then { b=p₁; empty=false; }
Sound Transformer

refined element

abstract element

“simple” abstract semantics

partial concretization

most-precise abstract semantics [CC’79]

abstract element
Partial Concretization-based Transformer

3-thread substate \rightarrow \text{exec} \rightarrow \text{statement st} \rightarrow 3-thread substate

factoids \rightarrow \text{substates} \rightarrow 3-thread substate \rightarrow \text{proj} \rightarrow factoids

\text{abstract semantics} \rightarrow \text{statement st} \rightarrow \text{factoids}
Transformer for Concurrent Systems

\[ \text{TR} (F) = \{ <l', g', o> : <l, g, o> \in F, <l, g> \tau <l', g'> \} \]

\[ \cup \left\{ <l_2, g', o>, <l_2, g', \alpha(l_1')> : f_1, f_2, f_3, f_4 \in F: \right. \]

\[ <l_1, g, l_2, o> \in \text{substates}(f_1, f_2, f_3, f_4), \]

\[ <l_1, g> \tau <l_1', g'> \]

Diagram:
- 3-thread substrate
- exec(tracked) statement st
- 3-thread substrate
- substrates
- proj
- factoids
- exec (1st) statement st
- factoids
- factoids \( \cup \) factoids
Partial Concretization

C1, C2
\[
\begin{array}{c}
\text{C1, C2} \\
\begin{array}{c}
\text{c}_1 \quad \text{c}_2 \\
\text{empty} \\
\text{6:}
\end{array}
\end{array}
\times
\begin{array}{c}
\begin{array}{c}
\text{C1, C3} \\
\begin{array}{c}
\text{c}_1 \quad \text{c}_3 \\
\text{empty} \\
\text{6:}
\end{array}
\end{array}
\end{array}
\times
\begin{array}{c}
\begin{array}{c}
\text{C1, C4} \\
\begin{array}{c}
\text{c}_1 \quad \text{c}_4 \\
\text{empty} \\
\text{6:}
\end{array}
\end{array}
\end{array}
\times
\begin{array}{c}
\begin{array}{c}
\text{C2, C1} \\
\begin{array}{c}
\text{c}_2 \quad \text{c}_1 \\
\text{empty} \\
\text{4:}
\end{array}
\end{array}
\end{array}
\times
\begin{array}{c}
\begin{array}{c}
\text{C2, C2} \\
\begin{array}{c}
\text{c}_2 \quad \text{c}_4 \\
\text{empty} \\
\text{6:}
\end{array}
\end{array}
\end{array}
\times
\begin{array}{c}
\begin{array}{c}
\text{C2, C3} \\
\begin{array}{c}
\text{c}_2 \quad \text{c}_3 \\
\text{empty} \\
\text{6:}
\end{array}
\end{array}
\end{array}
\times
\begin{array}{c}
\begin{array}{c}
\text{C2, C4} \\
\begin{array}{c}
\text{c}_2 \quad \text{c}_4 \\
\text{empty} \\
\text{6:}
\end{array}
\end{array}
\end{array}
\end{array}
\]

C1: Executing
C2: Tracked
C3: Other
Partial Concretization (Substates)

C1, C2

C1, C3

C2, C1

C2, C3

no information on C4
6: C1: dispose(c) (exec)
6: C1: dispose(c) (project)
Reducing Quadratic Factors

\[ TR (F) = \{ <l', g', o> : <l, g, o> \in F, <l, g> \tau <l', g'> \} \cup \]
\[ \begin{cases} <l_2, g', o>, <l_2, g', \alpha(l_1')> : f_1, f_2, f_3, f_4 \in F: \\
\quad <l_1, g, l_2, o> \in \text{substates}(f_1, f_2, f_3, f_4), \\
\quad <l_1, g> \tau <l_1', g'> 
\end{cases} \]

• Exploit redundancies in the action
  • Cannot affect locals of other threads
  • Use asymmetry between the two abstractions
  • Can prove no loss of information
• Summarizing Effects
• Apply aggressive abstraction to the executing threads
  • Potential loss of precision
  • Summarizing Abstraction
Exploiting Redundancies

6: C1: dispose(c)

these states identical up to the PCs which are invisible to the executing thread
Exploiting Redundancies

6: C1: dispose(c)

C1, C2

C1, C3

C2, C1

C2, C3

substates

proj
Exploiting Redundancies 6: C1: dispose(c)

C1, C2

C1, C3

C2, C1

C2, C3

substates

proj

X

X

X

X

uframe

exec

6: C1: dispose(c)

empty
Summarizing Abstraction

\[ \text{TR} (F) = \{ <l', g', o> : <l, g, o> \in F, <l, g> \tau <l', g'> \} \cup \]

\[ \left\{ <l_2, g', o>, <l_2, g', \alpha(l_1')> : f_1, f_2, f_3, f_4 \in F : \right. \]

\[ \left. <l_1, g, l_2, o> \in \text{substates}(f_1, f_2, f_3, f_4), \right. \]

\[ <l_1, g> \tau <l_1', g'> \]

- Summarizing Effects reduces the tracked thread’s number of states
- Summarizing Abstraction reduces state of executing thread
  - Our heuristic – keep only information accessed by statement
- Significant reduction in size of partial concretization
  - Especially in heap-manipulating programs
  - Precise enough in our benchmarks
A Singleton Buffer - Modified

Boolean empty = true;
Object b = null;

produce() {
    Object p = new();
    await (empty) then {
        b = p;
        empty = false;
    }
}

consume() {
    Object c;
    Boolean x;
    4: await (!empty) then {
        c = b;
        empty = true;
    }
    5: x = f(c);
    6: dispose(c);
    7: use(x);
    8:
}
Example 6: C1: dispose(c)

C1,C2
\[x_1 c_1 c_2\]
6: empty

C1,C3
\[x_1 c_1 c_3\]
6: empty

C2,C1
\[c_1 c_2\]
6: empty

C2,C3
\[c_2 c_3\]
6: empty

proj
C2,C3
\[c_2 c_3\]
5: empty

substates

exec
C1: dispose(c)
Example

6: C1: dispose(c)

exec

C1: dispose(c)

substates

proj

C2,C1

C2,C3

C2,C1

C2,C3

48
Gain of Precision in Summarizing Abstractions

6: Dispose(c1)
Unbounded Number of Threads

- Abstract thread identifiers
- Usually no extra loss of precision
- Universally quantity over threads
Thread-Modular Abstraction for Unbounded Number of threads

P1
\[ p_1 \times p_2 \times p_3 \]

P2
\[ p_1 \times p_2 \times p_3 \]

P3
\[ p_1 \times p_2 \times p_3 \]

C1
\[ c_1 \times c_2 \times c_3 \]

C2
\[ c_1 \times c_2 \times c_3 \]

C3
\[ c_1 \times c_2 \times c_3 \]
Thread-Modular Abstraction for Unbounded Number of threads

∀t:

(pc(t)=3 ∧ p(t) ≠ b ∧ valid(p(t)) ∧ valid(b) ∧ empty) ∨
(pc(t)=3 ∧ p(t) = b ∧ valid(p(t)) ∧ valid(b) ∧ empty) ∨
(pc(t)=5 ∧ c(t) ≠ b ∧ valid(c(t)) ∧ valid(b) ∧ empty) ∨
(pc(t)=5 ∧ c(t) = b ∧ valid(c(t)) ∧ valid(b) ∧ empty)
Semi-Thread-Modular Abstraction for Unbounded Number of threads

C1,C2

\[ \begin{array}{c}
  c_1 \downarrow \\
  c_2 \downarrow \\
  6: \text{empty}
\end{array} \]

C1,C3

\[ \begin{array}{c}
  c_1 \downarrow \\
  c_3 \downarrow \\
  6: \text{empty}
\end{array} \]

C2,C1

\[ \begin{array}{c}
  c_2 \downarrow \\
  c_1 \downarrow \\
  6: \text{empty}
\end{array} \]

C2,C3

\[ \begin{array}{c}
  c_2 \downarrow \\
  c_3 \downarrow \\
  4: \text{empty}
\end{array} \]

Ct Cs

\[ \begin{array}{c}
  c_t \downarrow \\
  c_s \downarrow \\
  6: \text{empty}
\end{array} \]

\[ \begin{array}{c}
  c_t \downarrow \\
  c_s \downarrow \\
  4: \text{empty}
\end{array} \]

53
∀\( t, s: s \neq t \Rightarrow \)
\[
(pc(t)=6 \land c(t) \neq c(s) \land valid(c(t)) \land valid(c(s)) \land empty)
\]
\[
\lor
\]
\[
(pc(t)=6 \land valid(c(t)) \land c(s)=null \land empty)
\]
\[
\lor
\]
\[
(pc(t)=4 \land c(t)=null \land valid(c(s)) \land empty)
\]
\[
\lor
\]
\[
(pc(t)=4 \land c(t)=null \land c(s)=null \land empty)
\]
In the TR

- Proofs of soundness
- No loss of precision from summarizing effects
- Combination with heap abstraction
  - Meet is important
Evaluation

- Implemented (semi-)thread-modular shape analysis using HEDEC/TVLA
  - Unbounded number of threads
  - Unbounded number of objects
  - Call strings for procedures
- Thread-modular unable to prove properties without additional (global) instrumentation
- Semi-thread-modular analysis proves required properties
- Reproduce the injected errors
Evaluation

![Graph showing evaluation results over time for different algorithms. The x-axis represents algorithms: Hand Over Hand, HS Set, DCAS PLDI 08, CAS PLDI 08, M Maged (SPAA 02), M Maged Opt. The y-axis represents time in seconds ranging from 0 to 60000. Different colors represent different conditions: No Summ Abs No Summ Effects, No Summ Abs With Summ Effects, With Summ Abs No Summ Effects, With Summ Abs With Summ Effects.]
Evaluation

Speedup

- Hand Over Hand
- HS Set
- DCAS PLDI 08
- CAS PLDI 08
- M Maged (SPAA 02)
- M Maged Opt

Legend:
- Light Blue: Summ. effects speedup
- Purple: Summ Abs. speedup
- Yellow: Both speedup
Related Work

- Process centric abstractions
  - [C. A. R. Hoare '72] [Owicki & Gries '76]
  - [E. Clarke TOPLAS’80] [Talupur et al. VMCAI ’06]
  - [Flanagan & Qadeer, SPIN’03] many more…
  - [Malkis, Podelski, Rybalchenco, SAS’07]

- Thread-modular shape analysis
  - [Gotsman et al. PLDI ’07]
  - [Manevich et al. SAS’08]
  - [Calcagno et al. SAS’07]
  - R-G reasoning [Vafeiadis et al. ’06-’09]
Summary

A new abstraction for concurrent systems

- Scalable in the number of threads
- Handles unbounded number of threads
- Semi-thread-modular program analysis

Provably sound analysis

Potential loss of precision

- Abstraction
- Transformers
  - But precise enough - 0 false alarms

Reducing quadratic factors