

Verifying Array Manipulating Programs by Tiling

Authors: Supratik Chakraborty, Ashutosh Gupta, Divyesh Unadkat

R Venkatesh
TCS Research

December 11-16, 2017
Winter School in Software Engineering
TATA CONSULTANCY SERVICES
Pune, India

- Verifying array programs with complex access patterns is challenging
- State-of-the-art tools choke on many such examples
- Solution - *Inductive Compositional Reasoning*
 - ▶ *Infer* array access patterns in loops
 - ▶ *Tile* the set of indices using the inferred patterns
 - ▶ *Slice* the assertion using the tile for a single iteration of the loop
 - ▶ Compositionally *prove* universally quantified assertions on arrays

Motivating Example

```
void foo(int A[], int N) {  
    for (int i = 0; i < N; i++) {  
        if (!(i==0 || i==N-1)) {  
            if (A[i] < THRESH) {  
                A[i+1] = A[i] + 1;  
                A[i] = A[i-1];  
            }  
        } else {  
            A[i] = THRESH;  
        }  
    }  
    assert(for i in 0..N-1, A[i]>=THRESH);  
}
```

Motivating Example

```
void foo(int A[], int N) {  
    for (int i = 0; i < N; i++) {  
        if (!(i==0 || i==N-1)) {  
            if (A[i] < 5) {  
                A[i+1] = A[i] + 1;  
                A[i] = A[i-1];  
            }  
        } else {  
            A[i] = 5;  
        }  
    }  
    assert(for k in 0..N-1, A[k]>=5);  
}
```

Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if (!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Initial array

0	1	2	3	4	5	6	7	— Loop Counter
0	1	2	3	4	5	6	7	— Indices
5	9	7	1	9	2	8	1	— Cell Contents

$\neg \forall k. a[k] \geq 5$

Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if (!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k] >= 5);
}
```

Initial array

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	1	9	2	8	1

$\neg \forall k. a[k] \geq 5$

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	2	2	8	1

$i \quad i+1$

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	7	3	8	1

$i \quad i+1$

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	7	7	4	1

$i \quad i+1$

Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if (!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
    } else {
      A[i] = 5;
    }
  }
  assert(for k in 0..N-1, A[k] >= 5);
}
```

Initial array

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	1	9	2	8	1

$$\neg \forall k. a[k] \geq 5$$

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	2	2	8	1

$a[i+1] \not\geq 5$

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	7	3	8	1

$a[i+1] \not\geq 5$

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	7	7	4	1

$a[i+1] \not\geq 5$

Motivating Example

```
void foo(int A[], int N) {
  for (int i = 0; i < N; i++) {
    if (!(i==0 || i==N-1)) {
      if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
      }
      else {
        A[i] = 5;
      }
    }
  }
  assert(for k in 0..N-1, A[k]>=5);
}
```

Initial array

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	1	9	2	8	1

$\neg \forall k. a[k] \geq 5$

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	2	2	8	1

i

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	7	3	8	1

i

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
5	9	7	7	7	7	4	1

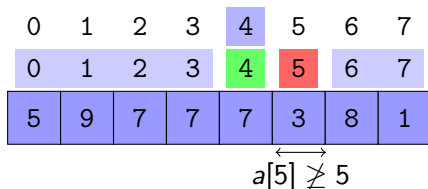
i

- Tile : LoopCounter \times Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

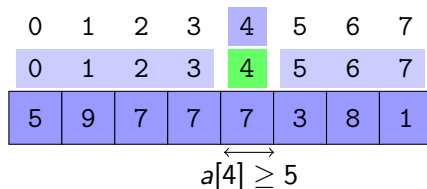
- $\text{Tile} : \text{LoopCounter} \times \text{Indices} \rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L
- $\text{Tile}(i, j) := i \leq j \leq i + 1$
- $\text{Tile}(i, j) := j == i$

- Tile : LoopCounter \times Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

- Tile(i, j) := $i \leq j \leq i + 1$

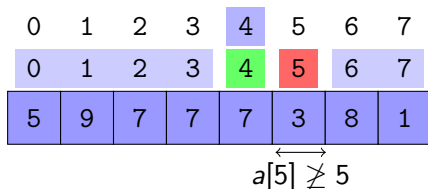


- Tile(i, j) := $j == i$



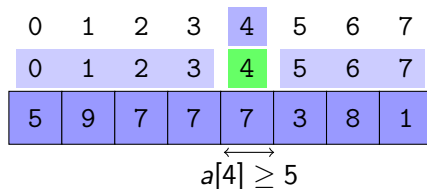
- Tile : LoopCounter \times Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

- Tile(i, j) := $i \leq j \leq i + 1$



- Truth of the assertion wrt tile **changes** in the next iteration

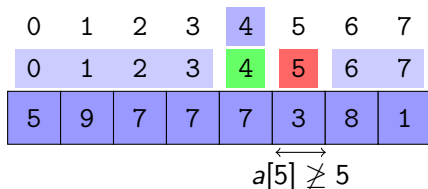
- Tile(i, j) := $j == i$



- Truth of the assertion wrt tile **doesn't change** in the future

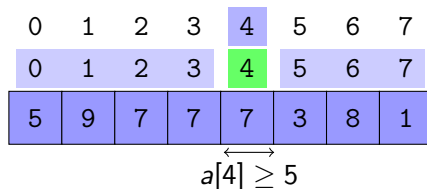
- Tile : LoopCounter \times Indices $\rightarrow \{\mathbf{tt}, \mathbf{ff}\}$ for loop L

- Tile(i, j) := $i \leq j \leq i + 1$



- Truth of the assertion wrt tile **changes** in the next iteration
- May miss update** to some indices

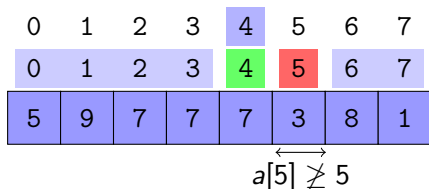
- Tile(i, j) := $j == i$



- Truth of the assertion wrt tile **doesn't change** in the future
- Doesn't miss updates** to any index

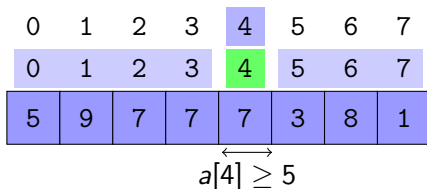
- Tile : $\text{LoopCounter} \times \text{Indices} \rightarrow \{\text{tt}, \text{ff}\}$ for loop L

- Tile(i, j) := $i \leq j \leq i + 1$



- Truth of the assertion wrt tile **changes** in the next iteration
- May miss update** to some indices

- Tile(i, j) := $j == i$



- Truth of the assertion wrt tile **doesn't change** in the future
- Doesn't miss updates** to any index

Finding the *right* tile is a challenge!

Battery Voltage Regulator

```
void BVR(int N, int MIN)
{
    int i;
    int volArray[N];

    if(N % 4 != 0) { return; }

    assume(N % 4 == 0);
    for(i = 1; i <= N/4; i++)
    {
        if(1 >= MIN)
            volArray[i*4-1] = 1;
        else
            volArray[i*4-1] = 0;
        if(3 >= MIN)
            volArray[i*4-2] = 3;
        else
```

```
            volArray[i*4-2] = 0;
        if(7 >= MIN)
            volArray[i*4-3] = 7;
        else
            volArray[i*4-3] = 0;
        if(5 >= MIN)
            volArray[i*4-4] = 5;
        else
            volArray[i*4-4] = 0;
    }

    for(i = 0; i < N; i++)
    {
        assert(volArray[i] >= MIN ||
            volArray[i] == 0);
    }
}
```

Battery Voltage Regulator

```
void BVR(int N, int MIN)
{
    int i;
    int volArray[N];

    if(N % 4 != 0) { return; }

    assume(N % 4 == 0);
    for(i = 1; i <= N/4; i++)
    {
        if(1 >= MIN)
            volArray[i*4-1] = 1;
        else
            volArray[i*4-1] = 0;
        if(3 >= MIN)
            volArray[i*4-2] = 3;
        else
```

```
            volArray[i*4-2] = 0;
        if(7 >= MIN)
            volArray[i*4-3] = 7;
        else
            volArray[i*4-3] = 0;
        if(5 >= MIN)
            volArray[i*4-4] = 5;
        else
            volArray[i*4-4] = 0;
    }

    for(i = 0; i < N; i++)
    {
        assert(volArray[i] >= MIN ||
            volArray[i] == 0);
    }
}
```

$\text{Tile}(i, j) := 4 * i - 4 \leq j < 4 * i$

Array Reversal

```
void revcopynswap(int N)
{
    int i;
    int tmp;
    int a[N];
    int b[N];
    int rev_copy[N];

    for(i = 0; i < N; i++)
    {
        rev_copy[N-i-1] = a[i];
    }
```

```
    for(i = 0; i < N; i++)
    {
        tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }

    for(i = 0; i < N; i++)
    {
        assert(b[i] == rev_copy[N-i-1]);
    }
}
```

```
void revcopynswap(int N)
{
    int i;
    int tmp;
    int a[N];
    int b[N];
    int rev_copy[N];

    for(i = 0; i < N; i++)
    {
        rev_copy[N-i-1] = a[i];
    }
```

```
    for(i = 0; i < N; i++)
    {
        tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }

    for(i = 0; i < N; i++)
    {
        assert(b[i] == rev_copy[N-i-1]);
    }
}
```

Loop 1 - $\text{Tile}(i, j) := j == N - i - 1$

Loop 2 - $\text{Tile}(i, j) := j == i$

```
void skip(int N)
{
    int i;
    int a[N];

    if(N % 2 != 0)
    {
        return;
    }

    assume(N % 2 == 0);
    for(i = 1; i <= N/2; i++ )
    {
        if( a[2*i-2] > 2*i-2 )
        {
```

```
            a[2*i-2] = 2*i-2;
        }

        if( a[2*i-1] > 2*i-1 )
        {
            a[2*i-1] = 2*i-1;
        }
    }

    for(i = 0; i < N; i++)
    {
        assert(a[i] <= i);
    }
    return;
}
```

```
void skip(int N)
{
    int i;
    int a[N];

    if(N % 2 != 0)
    {
        return;
    }

    assume(N % 2 == 0);
    for(i = 1; i <= N/2; i++)
    {
        if( a[2*i-2] > 2*i-2 )
        {
            a[2*i-2] = 2*i-2;
        }

        if( a[2*i-1] > 2*i-1 )
        {
            a[2*i-1] = 2*i-1;
        }
    }

    for(i = 0; i < N; i++)
    {
        assert(a[i] <= i);
    }
    return;
}
```

$\text{Tile}(i, j) := 2 * i - 2 \leq j < 2 * i$

- **Reverse** the contents of the array
 - ▶ $\text{Tile}(i, j) := j == N - i - 1$
- A **bunch** of indices updated in a loop
 - ▶ $\text{Tile}(i, j) := 2 * i - 2 \leq j < 2 * i$
 - ▶ $\text{Tile}(i, j) := 3 * i - 3 \leq j < 3 * i$
 - ▶ $\text{Tile}(i, j) := 4 * i - 4 \leq j < 4 * i$
- **Adjacent** indices to the counter
 - ▶ $\text{Tile}(i, j) := j == i - 1$
 - ▶ $\text{Tile}(i, j) := j == i + 1$
- Most **common** tile in array processing loops
 - ▶ $\text{Tile}(i, j) := j == i$

```
void foo(int A[], int N) {  
    int j;  
    for (int i = 0; i < N; i++) {  
        if(!(i==0 || i==N-1)) {  
            if (A[i] < 5) {  
                A[i+1] = A[i] + 1;  
                A[i] = A[i-1];  
            }  
        } else {  
            A[i] = 5;  
        }  
        if(*) { j=i; }  
        if(*) { j=i+1; }  
    }  
    assert(for k in 0..N-1, A[k]>=5);  
}
```

Using array access patterns

- Introduce a loop counter (say i)
- Store values of update indices (say in j)
- Infer a relation between i and j
- Use arithmetic invariant generators for inference
- Inferred relation
 $\text{Tile}(i, j) := i \leq j \leq i + 1$
- Removing overlap
 $\text{Tile}(i, j) := j == i$

$$\begin{aligned}
 \text{PB} &::= \text{St} \\
 \text{St} &::= v := E \mid A[E] := E \mid \text{assume}(\text{BoolE}) \mid \\
 &\quad \text{if}(\text{BoolE}) \text{ then } \text{St} \text{ else } \text{St} \mid \\
 &\quad \text{for } (\ell := 0; \ell < E; \ell := \ell + 1) \{ \text{St} \} \mid \\
 &\quad \text{St} ; \text{St} \\
 E &::= E \text{ op } E \mid A[E] \mid v \mid \ell \mid c \\
 \text{BoolE} &::= E \text{ relop } E \mid \text{BoolE AND BoolE} \mid \\
 &\quad \text{NOT BoolE} \mid \text{BoolE OR BoolE}
 \end{aligned}$$

- No unstructured jumps
- Loop counter goes from 0 to some max value
- Assignment statements in body do not update loop counter

- Notation
 - ▶ I denotes a sequence of array index variables
 - ▶ \mathcal{A} is a set of array variables
 - ▶ Inv is a (possibly weak) loop invariant for loop L

- Notation
 - ▶ I denotes a sequence of array index variables
 - ▶ \mathcal{A} is a set of array variables
 - ▶ Inv is a (possibly weak) loop invariant for loop L
- Example Post-conditions/assertions
 - ▶ $\forall i$ between 0 and N , $A[i]$ is greater equal to minimum
 - ▶ $\forall i$ if i is even & between 0 and N then $A[i] = i$

- Notation

- ▶ I denotes a sequence of array index variables
- ▶ \mathcal{A} is a set of array variables
- ▶ Inv is a (possibly weak) loop invariant for loop L

- Example Post-conditions/assertions

- ▶ $\forall i$ between 0 and N , $A[i]$ is greater equal to minimum
- ▶ $\forall i$ if i is even & between 0 and N then $A[i] = i$

- Formalization of Post-conditions

- ▶ $\text{Post} \triangleq \forall I (\Phi(I) \implies \Psi(\mathcal{A}, I))$
- ▶ $\Phi(I)$ - quantifier-free formula in theory of arithmetic over integers
- ▶ $\Psi(\mathcal{A}, I)$ - quantifier-free formula in combined theory of arrays and arithmetic over integers

If following conditions hold on the tile, we have proven the property

T1: Covers range

T2: Sliced post-condition holds inductively

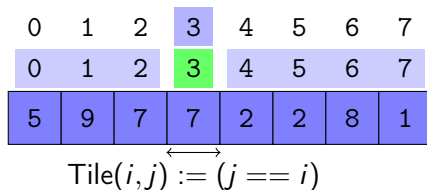
T3: Non-interference across tiles

T1: Covers Range

Indices of interest must be covered by some *tile*

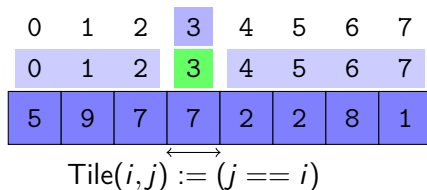
T1: Covers Range

Indices of interest must be covered by some *tile*



T1: Covers Range

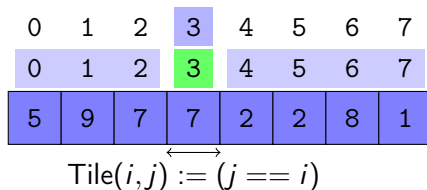
Indices of interest must be covered by some *tile*



- $\eta_1 \equiv \forall j (\Phi(j) \implies \exists i (\text{Tile}(i, j)))$

T1: Covers Range

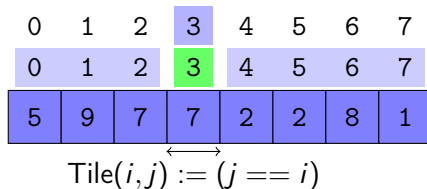
Indices of interest must be covered by some *tile*



- $\eta_1 \equiv \forall j (\Phi(j) \implies \exists i (\text{Tile}(i, j)))$
- $\eta_2 \equiv \forall i, j (\text{Tile}(i, j) \implies \Phi(j))$

T1: Covers Range

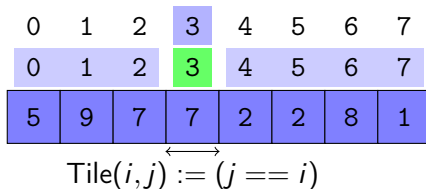
Indices of interest must be covered by some *tile*



- $\eta_1 \equiv \forall j (\Phi(j) \implies \exists i (\text{Tile}(i, j)))$
- $\eta_2 \equiv \forall i, j (\text{Tile}(i, j) \implies \Phi(j))$
- Validity of $\eta_1 \wedge \eta_2$ ensures T1

T1: Covers Range

Indices of interest must be covered by some *tile*



- $\eta_1 \equiv \forall j (\Phi(j) \implies \exists i (\text{Tile}(i, j)))$
- $\eta_2 \equiv \forall i, j (\text{Tile}(i, j) \implies \Phi(j))$
- Validity of $\eta_1 \wedge \eta_2$ ensures T1
- Involves a quantifier alternation; can be handled by SMT solvers

T1: Covers Range

- $\neg(\eta_1 \wedge \eta_2)$ must be unsat

T1: Covers Range

- $\neg(\eta_1 \wedge \eta_2)$ must be unsat
- Negated smt formula is as shown below

```
(declare-fun size () Int)
(declare-fun i () Int)
(declare-fun j () Int)
(assert (or
  (and (>= j 0) (< j size)
    (forall ((i Int))
      (=> (and (>= i 0) (< i size)) (not (= j i)) )))
  (and (>= i 0) (< i size) (= j i)
    (not (and (>= j 0) (< j size))))))
(check-sat)
```

T1: Covers Range

- $\neg(\eta_1 \wedge \eta_2)$ must be unsat
- Negated smt formula is as shown below

```
(declare-fun size () Int)
(declare-fun i () Int)
(declare-fun j () Int)
(assert (or
  (and (>= j 0) (< j size)
    (forall ((i Int))
      (=> (and (>= i 0) (< i size)) (not (= j i)) )))
  (and (>= i 0) (< i size) (= j i)
    (not (and (>= j 0) (< j size))))))
(check-sat)
```

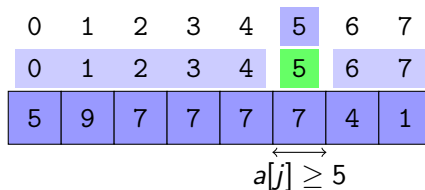
- State-of-the-art solvers can prove unsatisfiability of such formulae

T2: Sliced Post-condition holds Inductively

Post-condition wrt indices in the i^{th} tile holds inductively

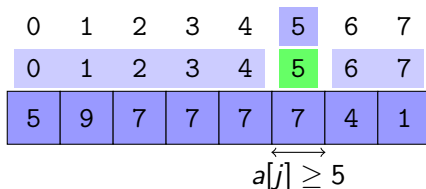
T2: Sliced Post-condition holds Inductively

Post-condition wrt indices in the i^{th} tile holds inductively



T2: Sliced Post-condition holds Inductively

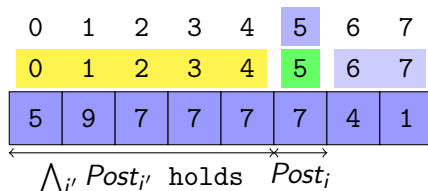
Post-condition wrt indices in the i^{th} tile holds inductively



- Sliced post-condition for the i^{th} tile
 $\text{Post}_i \triangleq \forall j (\text{Tile}(i, j) \wedge \Phi(j) \implies \Psi(A, j))$

T2: Sliced Post-condition holds Inductively

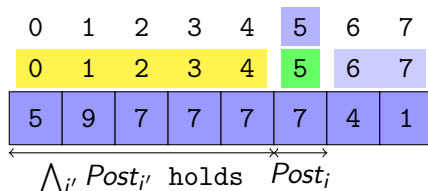
Post-condition wrt indices in the i^{th} tile holds inductively



- Sliced post-condition for the i^{th} tile
 $\text{Post}_i \triangleq \forall j (\text{Tile}(i, j) \wedge \Phi(j) \implies \Psi(A, j))$

T2: Sliced Post-condition holds Inductively

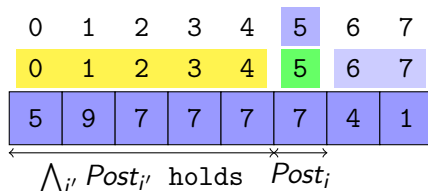
Post-condition wrt indices in the i^{th} tile holds inductively



- Sliced post-condition for the i^{th} tile
 $\text{Post}_i \triangleq \forall j (\text{Tile}(i, j) \wedge \Phi(j) \implies \Psi(A, j))$
- $\{\text{Inv} \wedge \bigwedge_{i': 0 \leq i' < i} \text{Post}_{i'}\} \text{L}_{\text{body}} \{\text{Inv} \wedge \text{Post}_i\}$ must be valid

T2: Sliced Post-condition holds Inductively

Post-condition wrt indices in the i^{th} tile holds inductively



- Sliced post-condition for the i^{th} tile

$$\text{Post}_i \triangleq \forall j (\text{Tile}(i, j) \wedge \Phi(j) \implies \Psi(A, j))$$
- $\{\text{Inv} \wedge \bigwedge_{i': 0 \leq i' < i} \text{Post}_{i'}\} \text{L}_{\text{body}} \{\text{Inv} \wedge \text{Post}_i\}$ must be valid
- After removing quantification

$$\{\text{Inv} \wedge \text{Tile}(i, j) \wedge \Phi(j) \wedge \Psi(A, j')\} \text{L}_{\text{body}} \{\text{Inv} \wedge \Psi(A, j)\}$$

T2: Sliced Post-condition holds Inductively

Original Program

Transformed Program

T2: Sliced Post-condition holds Inductively

Original Program

```
void foo(int A[], int N) {
    for (int i = 0; i < N; i++)
    {
        if(!(i==0 || i==N-1)) {
            if (A[i] < 5) {
                A[i+1] = A[i] + 1;
                A[i] = A[i-1];
            }
        } else {
            A[i] = 5;
        }
    }
    assert(for k in 0..N-1, A[k]>=5);
}
```

Transformed Program

```
i=*; j=*; jp=*;
assume(0 <= i < N);
assume(j == i);
assume(jp == i-1);
assume(A[jp] >= 5);

if(!(i==0 || i==N-1)) {
    if (A[i] < 5) {
        A[i+1] = A[i] + 1;
        A[i] = A[i-1];
    }
} else {
    A[i] = 5;
}

assert(A[j] >= 5);
```

T2: Sliced Post-condition holds Inductively

Original Program

```
void foo(int A[], int N) {  
    for (int i = 0; i < N; i++)  
    {  
  
        if(!(i==0 || i==N-1)) {  
            if (A[i] < 5) {  
                A[i+1] = A[i] + 1;  
                A[i] = A[i-1];  
            }  
        } else {  
            A[i] = 5;  
        }  
  
    }  
    assert(for k in 0..N-1, A[k]>=5);  
}
```

Transformed Program

```
i=*; j=*; jp=*;  
assume(0 <= i < N);  
assume(j == i);  
assume(jp == i-1);  
assume(A[jp] >= 5);  
  
if(!(i==0 || i==N-1)) {  
    if (A[i] < 5) {  
        A[i+1] = A[i] + 1;  
        A[i] = A[i-1];  
    }  
} else {  
    A[i] = 5;  
}  
  
assert(A[j] >= 5);
```

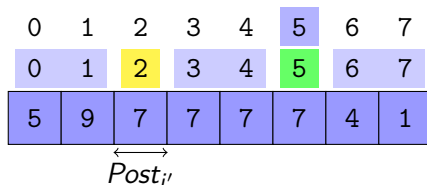
- Use BMC to ensure T2 by checking the loop free code

T3: Non-interference across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established

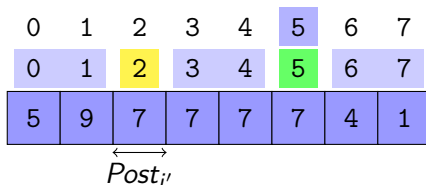
T3: Non-interference across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established



T3: Non-interference across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established

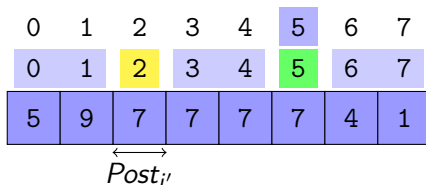


- Sliced post-condition for the i'^{th} tile

$$Post_{i'} \triangleq \forall j' (Tile(i', j') \wedge \Phi(j') \implies \Psi(A, j'))$$

T3: Non-interference across Tiles

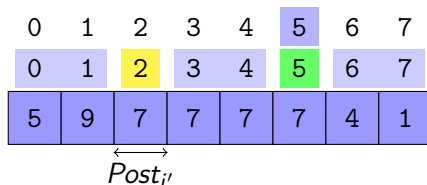
No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established



- Sliced post-condition for the i'^{th} tile
 $Post_{i'} \triangleq \forall j' (Tile(i', j') \wedge \Phi(j') \implies \Psi(A, j'))$
- $\{Inv \wedge (0 \leq i' < i) \wedge Post_{i'}\} L_{body} \{Post_{i'}\}$ must be valid

T3: Non-interference across Tiles

No iteration $i > i'$ interferes with the truth of $Post_{i'}$, once established



- Sliced post-condition for the i'^{th} tile
 $Post_{i'} \triangleq \forall j' (Tile(i', j') \wedge \Phi(j') \implies \Psi(A, j'))$
- $\{Inv \wedge (0 \leq i' < i) \wedge Post_{i'}\} L_{body} \{Post_{i'}\}$ must be valid
- After removing quantification
 $\{Inv \wedge (0 \leq i' < i) \wedge Tile(i', j') \wedge \Phi(j') \wedge \Psi(A, j')\} L_{body} \{\Psi(A, j')\}$

T3: Non-interference across Tiles

Original Program

Transformed Program

T3: Non-interference across Tiles

Original Program

```
void foo(int A[], int N) {  
    for (int i = 0; i < N; i++)  
    {  
  
        if(!(i==0 || i==N-1)) {  
            if (A[i] < 5) {  
                A[i+1] = A[i] + 1;  
                A[i] = A[i-1];  
            }  
        } else {  
            A[i] = 5;  
        }  
  
    }  
    assert(for k in 0..N-1, A[k]>=5);  
}
```

Transformed Program

```
i=*; ip=*; jp=*;  
assume(0 <= i < N);  
assume(0 <= ip < i);  
assume(jp == ip);  
assume(A[jp] >= 5);  
  
if(!(i==0 || i==N-1)) {  
    if (A[i] < 5) {  
        A[i+1] = A[i] + 1;  
        A[i] = A[i-1];  
    }  
} else {  
    A[i] = 5;  
}  
  
assert(A[jp] >= 5);
```

T3: Non-interference across Tiles

Original Program

```
void foo(int A[], int N) {  
    for (int i = 0; i < N; i++)  
    {  
  
        if(!(i==0 || i==N-1)) {  
            if (A[i] < 5) {  
                A[i+1] = A[i] + 1;  
                A[i] = A[i-1];  
            }  
        } else {  
            A[i] = 5;  
        }  
  
    }  
    assert(for k in 0..N-1, A[k]>=5);  
}
```

Transformed Program

```
i=*; ip=*; jp=*;  
assume(0 <= i < N);  
assume(0 <= ip < i);  
assume(jp == ip);  
assume(A[jp] >= 5);  
  
if(!(i==0 || i==N-1)) {  
    if (A[i] < 5) {  
        A[i+1] = A[i] + 1;  
        A[i] = A[i-1];  
    }  
} else {  
    A[i] = 5;  
}  
  
assert(A[jp] >= 5);
```

- Use BMC to ensure T3 by checking the loop free code

- Inductive Reasoning

T2 Sliced post-condition holds for each iteration

- Inductive Reasoning

T2 Sliced post-condition holds for each iteration

- Compositional Reasoning

T3 Truth of sliced post-condition once established is not altered subsequently

T1 Tiles cover the entire range of array indices of interest

Inductive Compositional Reasoning

- Inductive Reasoning

T2 Sliced post-condition holds for each iteration

- Compositional Reasoning

T3 Truth of sliced post-condition once established is not altered subsequently

T1 Tiles cover the entire range of array indices of interest

Theorem

Suppose $\text{Tile} : \text{LoopCounter} \times \text{Indices} \rightarrow \{\text{tt}, \text{ff}\}$ satisfies T1, T2 and T3. If $\text{Pre} \Rightarrow \text{Inv}$ holds and the loop L iterates at least once, then the Hoare triple $\{\text{Pre}\} L \{\text{Post}\}$ holds.

Proof.

The proof proceeds by induction on values of LoopCounter (say i).

Given:

$$\text{Pre} \Rightarrow \text{Inv} \quad (1)$$

Base Case:

Prove $\{\text{Pre}\} \text{ L}_{\text{body}} \{\text{Post}_i\}$ holds, where $i = 0$

$$\{\text{Inv}\} \text{ L}_{\text{body}} \{\text{Inv} \wedge \text{Post}_i\} \quad (\because T2) \quad (2)$$

$$\{\text{Pre}\} \text{ L}_{\text{body}} \{\text{Inv} \wedge \text{Post}_i\} \quad (\because \text{From (1) \& (2)}) \quad (3)$$

$$\{\text{Pre}\} \text{ L}_{\text{body}} \{\text{Post}_i\} \quad (\because \text{Inv} \wedge \{\text{Post}_i\} \Rightarrow \{\text{Post}_i\}) \quad (4)$$

Induction Hypothesis:

$$\{\text{Pre}\} (\text{L}_{\text{body}})^{i'} \left\{ \bigwedge_{i': 0 \leq i' < i} \text{Post}_{i'} \right\} \quad (\because T3) \quad (5)$$

Induction:

Assuming hypothesis, prove $\{\text{Pre}\} (L_{\text{body}})^{i'} \{\bigwedge_{i':0 \leq i' \leq i} \text{Post}_{i'}\}$ holds.

$$\{\text{Inv} \wedge \bigwedge_{i':0 \leq i' < i} \text{Post}_{i'}\} L_{\text{body}} \{\text{Inv} \wedge \text{Post}_i\} \quad (\because T2) \quad (6)$$

$$\{\text{Inv} \wedge \bigwedge_{i':0 \leq i' < i} \text{Post}_{i'}\} L_{\text{body}} \{\text{Post}_i\} \quad (\because \text{Inv} \wedge \text{Post}_i \Rightarrow \text{Post}_i) \quad (7)$$

At the end of the i^{th} iteration of the loop L the following Hoare triple holds:

$$\{\text{Pre}\} (L_{\text{body}})^{i'} \{\bigwedge_{i':0 \leq i' \leq i} \text{Post}_{i'}\} \quad (\because \text{From (5) \& (7)}) \quad (8)$$

$$\bigwedge_i \text{Post}_i \equiv \text{Post} \quad (\because T1) \quad (9)$$

$$\{\text{Pre}\} L \{\text{Post}\} \quad (\because \text{From (8) \& (9)}) \quad \square$$

Sequentially Composed Loops

```
void copynswap(int N)
{
    int i, tmp;
    int a[], b[], acopy[];

    for (i = 0; i < N; i++) {
        acopy[i] = a[i];
    }

    for (i = 0; i < N; i++) {
        tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }

    for (i = 0; i < N; i++) {
        assert(b[i] == acopy[i]);
    }
}
```

```
void copynswap(int N)
{
    int i, tmp;
    int a[], b[], acopy[];

    for (i = 0; i < N; i++) {
        acopy[i] = a[i];
    }

    for (i = 0; i < N; i++) {
        tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }

    for (i = 0; i < N; i++) {
        assert(b[i] == acopy[i]);
    }
}
```

Mid-conditions

- Invariants between sequentially composed loops
- Hard to generate precise invariants
- Identify *candidate* mid-conditions using annotation assistants

```
void copynswap(int N)
{
    int i, tmp;
    int a[], b[], acopy[];

    for (i = 0; i < N; i++) {
        acopy[i] = a[i];
    }

    for (i = 0; i < N; i++) {
        tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }

    for (i = 0; i < N; i++) {
        assert(b[i] == acopy[i]);
    }
}
```

Mid-conditions

- Invariants between sequentially composed loops
- Hard to generate precise invariants
- Identify *candidate* mid-conditions using annotation assistants

Candidate mid-conditions

- $\forall i(a[i] = \text{acopy}[i])$
- $\forall i(a[i] \neq b[i])$

Sequentially Composed Loops

```
void copynswap(int N)
{
    int i, tmp;
    int a[], b[], acopy[];

    for (i = 0; i < N; i++) {
        acopy[i] = a[i];
    }

    for (i = 0; i < N; i++) {
        tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }

    for (i = 0; i < N; i++) {
        assert(b[i] == acopy[i]);
    }
}
```

Mid-conditions

- Invariants between sequentially composed loops
- Hard to generate precise invariants
- Identify *candidate* mid-conditions using annotation assistants
- *Prove* them using Tiling

Candidate mid-conditions

- $\forall i(a[i] = \text{acopy}[i])$
- $\forall i(a[i] \neq b[i])$

Proved mid-conditions

- $\forall i(a[i] = \text{acopy}[i])$

Nested Loops

```
void nested(int N)
{
    int i, j, VAL=2, arr[];

    if(N % 5 != 0) { return; }

    assume(N % 5 == 0);
    for(i = 1; i <= N/5; i++)
    {
        for(j = 1; j <= 5; j++)
        {
            if(j >= VAL)
                arr[i*5 - j] = j;
            else
                arr[i*5 - j] = 0;
        }
    }

    for(i = 0; i < N; i++)
        assert(arr[i] >= VAL || arr[i] == 0 );
}
```

Nested Loops

```
void nested(int N)
{
    int i, j, VAL=2, arr[];

    if(N % 5 != 0) { return; }

    assume(N % 5 == 0);
    for(i = 1; i <= N/5; i++)
    {
        for(j = 1; j <= 5; j++)
        {
            if(j >= VAL)
                arr[i*5 - j] = j;
            else
                arr[i*5 - j] = 0;
        }
    }

    for(i = 0; i < N; i++)
        assert(arr[i] >= VAL || arr[i] == 0 );
}
```

Technique continues to work

- Analysis applies to segments
- Segments are paths between loop heads
- Tiles generated for each segment
- Candidate invariants generated at each loop head
- Conditions T1, T2, T3 checked for each segment

Tiler Tool Diagram

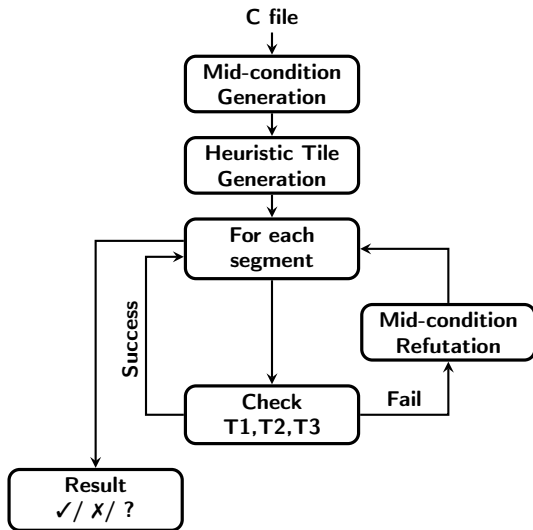


Figure : Tiler Tool Diagram

- Built on top of LLVM/CLANG infrastructure in C++
- Mid-condition generation
 - ▶ Daikon learns candidate scalar invariants from concrete traces
 - ▶ Lift these to quantified invariants
- Heuristic tile generation
 - ▶ Determine indices in terms of loop counters
 - ▶ Get a closed form expression in terms of index expressions
 - ▶ Remove possible overlaps
- Checking conditions T1, T2 and T3
 - ▶ Z3 for checking the validity of T1
 - ▶ CBMC for checking the validity of T2 and T3

- 60 benchmarks from industry and academia
- Performance compared with tools
 - ▶ SMACK+Corral - Bounded model checker
 - ▶ Booster - Acceleration based verification for arrays
 - ▶ Vaphor - Distinguished cell abstraction for arrays
- Memory limit - 1GB
- Time limit - 900s

Benchmark	#L	Tiler	S+C	Booster	Vaphor
cpyrev.c	2	✓3.8	†	✓3.1	✓5.4
cpynswp.c	2	✓4.2	†	✓12.4	✓1.38
cpynswp2.c	3	✓10.2	†	✓198	✓7.2*
maxinarr.c	1	✓0.51	†	✓0.01	✓0.11
mininarr.c	1	✓0.53	†	✓0.02	✓0.13
poly1.c	1	TO	†	✓15.7	TO
poly2.c	2	? 6.44	†	? 19.5	TO
tcpy.c	1	? 0.65	†	TO	✓25.1
rew.c	1	✓0.48	†	✓0.01	TO
skipped.c	1	✓1.24	†	TO	TO
rewrev.c	1	✓0.39	†	TO	TO
pr4.c	1	✓0.68	†	TO	TO
pr5.c	1	✓1.32	†	TO	TO
pnr4.c	1	✓0.86	†	TO	TO
pnr5.c	1	✓1.98	†	TO	TO
mbpr4.c	4	✓12.75	†	TO	TO
mbpr5.c	5	✓18.08	†	TO	TO
nr4.c	1-1	✓2.43*	†	TO	TO
nr5.c	1-1	✓2.90*	†	TO	TO
copy9u.c	9	✗0.16	✗4.48	✗0.44	✗30.8
skippedu.c	1	✗0.81	✗2.94	✗0.02	TO

Benchmark	#L	Tiler	S+C	Booster	Vaphor
cpyrev.c	2	✓3.8	†	✓3.1	✓5.4
cpynswp.c	2	✓4.2	†	✓12.4	✓1.38
cpynswp2.c	3	✓10.2	†	✓198	✓7.2*
maxinarr.c	1	✓0.51	†	✓0.01	✓0.11
mininarr.c	1	✓0.53	†	✓0.02	✓0.13
poly1.c	1	TO	†	✓15.7	TO
poly2.c	2	? 6.44	†	? 19.5	TO
tcpy.c	1	? 0.65	†	TO	✓25.1
rew.c	1	✓0.48	†	✓0.01	TO
skipped.c	1	✓1.24	†	TO	TO
rewrev.c	1	✓0.39	†	TO	TO
pr4.c	1	✓0.68	†	TO	TO
pr5.c	1	✓1.32	†	TO	TO
pnr4.c	1	✓0.86	†	TO	TO
pnr5.c	1	✓1.98	†	TO	TO
mbpr4.c	4	✓12.75	†	TO	TO
mbpr5.c	5	✓18.08	†	TO	TO
nr4.c	1-1	✓2.43*	†	TO	TO
nr5.c	1-1	✓2.90*	†	TO	TO
copy9u.c	9	✗0.16	✗4.48	✗0.44	✗30.8
skippedu.c	1	✗0.81	✗2.94	✗0.02	TO

Benchmark	#L	Tiler	S+C	Booster	Vaphor
cpyrev.c	2	✓3.8	†	✓3.1	✓5.4
cpynswp.c	2	✓4.2	†	✓12.4	✓1.38
cpynswp2.c	3	✓10.2	†	✓198	✓7.2*
maxinarr.c	1	✓0.51	†	✓0.01	✓0.11
mininarr.c	1	✓0.53	†	✓0.02	✓0.13
poly1.c	1	TO	†	✓15.7	TO
poly2.c	2	? 6.44	†	? 19.5	TO
tcpy.c	1	? 0.65	†	TO	✓25.1
rew.c	1	✓0.48	†	✓0.01	TO
skipped.c	1	✓1.24	†	TO	TO
rewrev.c	1	✓0.39	†	TO	TO
pr4.c	1	✓0.68	†	TO	TO
pr5.c	1	✓1.32	†	TO	TO
pnr4.c	1	✓0.86	†	TO	TO
pnr5.c	1	✓1.98	†	TO	TO
mbpr4.c	4	✓12.75	†	TO	TO
mbpr5.c	5	✓18.08	†	TO	TO
nr4.c	1-1	✓2.43*	†	TO	TO
nr5.c	1-1	✓2.90*	†	TO	TO
copy9u.c	9	✗0.16	✗4.48	✗0.44	✗30.8
skippedu.c	1	✗0.81	✗2.94	✗0.02	TO

Tiler in Action

Benchmark	#L	Tiler	S+C	Booster	Vaphor
cpyrev.c	2	✓3.8	†	✓3.1	✓5.4
cpynswp.c	2	✓4.2	†	✓12.4	✓1.38
cpynswp2.c	3	✓10.2	†	✓198	✓7.2*
maxinarr.c	1	✓0.51	†	✓0.01	✓0.11
mininarr.c	1	✓0.53	†	✓0.02	✓0.13
poly1.c	1	TO	†	✓15.7	TO
poly2.c	2	? 6.44	†	? 19.5	TO
tcpy.c	1	? 0.65	†	TO	✓25.1
rew.c	1	✓0.48	†	✓0.01	TO
skipped.c	1	✓1.24	†	TO	TO
rewrev.c	1	✓0.39	†	TO	TO
pr4.c	1	✓0.68	†	TO	TO
pr5.c	1	✓1.32	†	TO	TO
pnr4.c	1	✓0.86	†	TO	TO
pnr5.c	1	✓1.98	†	TO	TO
mbpr4.c	4	✓12.75	†	TO	TO
mbpr5.c	5	✓18.08	†	TO	TO
nr4.c	1-1	✓2.43*	†	TO	TO
nr5.c	1-1	✓2.90*	†	TO	TO
copy9u.c	9	✗0.16	✗4.48	✗0.44	✗30.8
skippedu.c	1	✗0.81	✗2.94	✗0.02	TO

Benchmark	#L	Tiler	S+C	Booster	Vaphor
cpynrev.c	2	✓3.8	†	✓3.1	✓5.4
cpynswp.c	2	✓4.2	†	✓12.4	✓1.38
cpynswp2.c	3	✓10.2	†	✓198	✓7.2*
maxinarr.c	1	✓0.51	†	✓0.01	✓0.11
mininarr.c	1	✓0.53	†	✓0.02	✓0.13
poly1.c	1	TO	†	✓15.7	TO
poly2.c	2	? 6.44	†	? 19.5	TO
tcpy.c	1	? 0.65	†	TO	✓25.1
rew.c	1	✓0.48	†	✓0.01	TO
skipped.c	1	✓1.24	†	TO	TO
rewrev.c	1	✓0.39	†	TO	TO
pr4.c	1	✓0.68	†	TO	TO
pr5.c	1	✓1.32	†	TO	TO
pnr4.c	1	✓0.86	†	TO	TO
pnr5.c	1	✓1.98	†	TO	TO
mbpr4.c	4	✓12.75	†	TO	TO
mbpr5.c	5	✓18.08	†	TO	TO
nr4.c	1-1	✓2.43*	†	TO	TO
nr5.c	1-1	✓2.90*	†	TO	TO
copy9u.c	9	✗0.16	✗4.48	✗0.44	✗30.8
skippedu.c	1	✗0.81	✗2.94	✗0.02	TO

Tiler Limitations

```
void tcpy(int N)
{
    int i, a[N], reverse[N];

    if(N % 2 != 0)
    { return; }

    assume(N % 2 == 0);
    for (i = 0; i < N/2; i++)
    {
        reverse[i] = a[N-i-1];
        reverse[N-i-1] = a[i];
    }

    for(i = 0; i < N; i++)
    {
        assert(a[i] == reverse[N-i-1]);
    }
}
```

```
void poly2(int N)
{
    int i, a[N];

    for(i=0; i<N; i++)
    {
        a[i] = i*i + 2;
    }

    for(i=0; i<N; i++)
    {
        a[i] = a[i] - 2;
    }

    for(i=0; i<N; i++)
    {
        assert(a[i] == i*i);
    }
}
```

- Abstract Interpretation based
 - ▶ Jiangchao Liu and Xavier Rival. “Abstraction of Arrays Based on Non Contiguous Partitions”. In: *VMCAI'15*
 - ▶ Patrick Cousot, Radhia Cousot, and Francesco Logozzo. “A parametric segmentation functor for fully automatic and scalable array content analysis”. In: *POPL'11*
 - ▶ Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. “Lifting abstract interpreters to quantified logical domains”. In: *POPL'08*
- Abstraction based
 - ▶ David Monniaux and Laure Gonnord. “Cell Morphing: From Array Programs to Array-Free Horn Clauses”. In: *SAS'16*
 - ▶ Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. “Booster: An Acceleration-Based Verification Framework for Array Programs”. In: *ATVA'14*
- Without explicit partitioning
 - ▶ Isil Dillig, Thomas Dillig, and Alex Aiken. “Fluid Updates: Beyond Strong vs. Weak Updates”. In: *ESOP'10*

- Presented a novel verification technique that
 - ▶ proves universally quantified assertions over arrays
 - ▶ decomposes reasoning about arrays using *tiles*
 - ▶ is property driven, compositional and efficient
- Future directions
 - ▶ Automated synthesis of *tiles*
 - ▶ Combining the strengths of Booster, Vaphor and Tiler
 - ▶ Integration of other candidate invariant generators like Houdini

Thank you