Loop Acceleration for C Programs [3] [4]

M.K. Srivas
Chennai Mathematical Institute

December 14, 2017

[3] Joint Work with Charles M.B. and M. Praveen

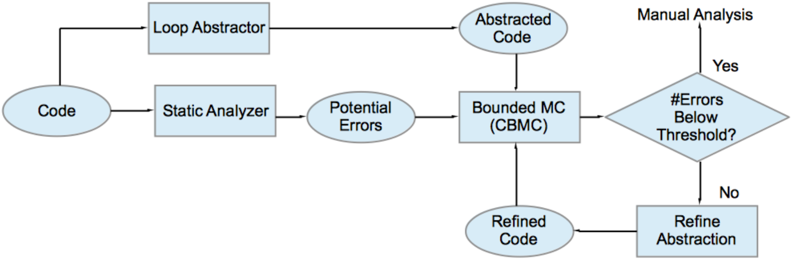
[4] Work supported in part by TCS Research

Outline

- Motivation and Context of Work
- Problem Definition: Loop Acceleration and our Restrictions
- Reduction of Problem to SMT
- Extension to handle overflows

Context of Work

- A Formal Verification Flow with Loop Abstraction



Abstract Loop Acceleration: Illustration

- An example where abstraction is adequate to prove property

```
int x, y, z = 0;
while (x <0x0fffffff) {
  x++;
  y = y+2;
  z = x+y;
}
assert(!(y % 2);
```

⇒

```
int x, y = 0;

int k = *;
Int x0, y0 = 0;
assume (x <0x0fffffff);
x = x0 + k*1;
y = y0 + k*2;
z = *;
assume (!(x <0x0fffffff));

assert(!(y % 2);
```

- P. Darke, et. al [DATE2015] [FM2015]

Abstract Loop Acceleration: Illustration

- An example where abstraction is inadequate to prove property

```
int x = 0;
while (x <0x0fffffff) {
  if (x < 0xfff0) x++;
  else x +=2;
}
assert(!(x % 2);
```

```
int x = 0;

int k, k1, k2 = *;
int x0 = 0;
assume (x < 0x0fffffff);
assume (k == k1+k2);
x = x0 + k1*1 k2*2;
assume (!(x <0x0fffffff));

assert(!(x % 2);
```

Abstract Loop Acceleration: Illustration

- An example where abstraction is inadequate to prove property

```
int x = 0;
while (x <0x0fffffff) {
  if (x < 0xfff0) x++;
  else x +=2;
}
assert(!(x % 2));
```

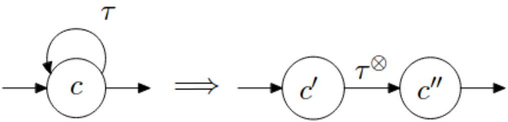
```
int x = 0;

int k, k1, k2 = *;
int x0 = 0;
assume (x < 0x0fffffff);
assume (k == k1+k2);
x = x0 + k1*1 k2*2;
assume (!(x <0x0fffffff));

assert(!(x % 2));
```

- Eliminates about 50% of spurious errors
- Was able to get 70% - 80% of SVCOMP in loop (safe) category

Main Sources of Imprecision in LABMC Acceleration

- Conditional control flow in the body
- Presence Overflows
- Other forms of assignments: Nonlinear closed forms
- Nested loops

Loop Acceleration: The General Problem



Is as hard as the Model Checking problem!!

What is the sweet spot for effort investment?

Our Problem Definition: Precise Linear Acceleration

- INPUT: A single-loop code with restricted linear expressions

```
int x = x0;
while (x <0x0fffffff) {
  if (x < 0xfff0) x++;
  else x +=2;
}
assert(!(x % 2);
```

Our Problem Definition: Precise Linear Acceleration

- INPUT: A single-loop code with restricted linear expressions

```
int x = x0;
while (x <0x0fffffff) {
  if (x < 0xfff0) x++;
  else x +=2;
}
assert(!(x % 2);
```

- Every var assignment is of the form $x = x \pm \delta$
- Loop condition and input assumption is a conjunction of linear inequalities
- Conditional expressions in loop body is a single linear inequality

Problem Definition: Precise Linear Acceleration

- OUTPUT: Precise acceleration for the loop as follows
- A sequence of guarded block assignments
- $(C1(X) \Rightarrow X = X0 + k_1 * \Delta_1) \wedge$
  $(C2(X) \Rightarrow X = X0 + k_2 * \Delta_2) \wedge ...$

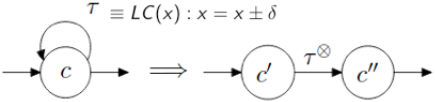
Problem Definition: Precise Linear Acceleration

- OUTPUT: Precise acceleration for the loop as follows
- A sequence of guarded block assignments
- $(C1(X) \Rightarrow X = X0 + k_1 * \Delta_1) \wedge$
  $(C2(X) \Rightarrow X = X0 + k_2 * \Delta_2) \wedge \ldots$

```
int x = 0;
int k, k1, k2 = *;
int x0 = 0;
assume (x < 0x0fffffff);

assume (x < 0xfff0); //BlocksPath1
 assume (k1 == 0xfff0 – x0 + 1);
  x = x0 + k1*1;

assume (!(x < 0xfff0)); //BlocksPath2
 assume (2*k2 == 0x0fffffff – x+2);
 x = x + k2*2;

assume (!(x <0x0fffffff));
assert(!(x % 2);
```

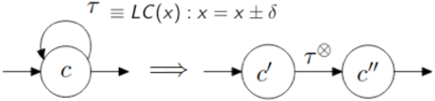
Problem Reduction to Quantified SMT Solving: No Conditionals

```
int x = x0;
while (LC(x))
   x = x ± δ;
```



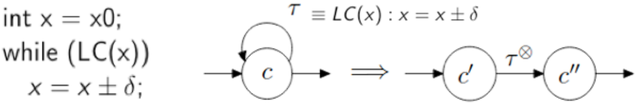
Problem Reduction to Quantified SMT Solving: No Conditionals

```
int x = x0;
while (LC(x))
   x = x ± δ;
```



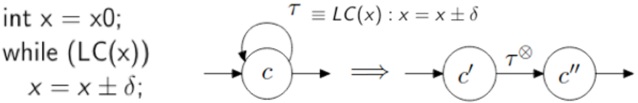
- Closed Form for value of x after k iterations: $\tau^k = x0 \pm k * \delta$

Problem Reduction to Quantified SMT Solving: No Conditionals



- Closed Form for value of x after k iterations: $\tau^k = x0 \pm k * \delta$
- Transitive Closure τ^* (at termination):
  $x0 \pm k * \delta$ (least) skolem solution to k s.t.
  - $\forall x_0 \exists k : ((LC(x0) \pm k * \delta) \wedge \neg(LC(x0) \pm (k+1) * \delta))$
  - Loop is non-terminating if no solution exists

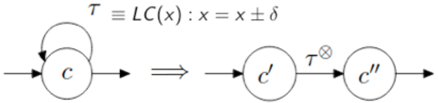
Problem Reduction to Quantified SMT Solving: No Conditionals



- Closed Form for value of x after k iterations: $\tau^k = x0 \pm k * \delta$
- Transitive Closure τ^* (at termination):
  $x0 \pm k * \delta$ (least) skolem solution to k s.t.
  - $\forall x_0 \exists k : ((LC(x0) \pm k * \delta) \wedge \neg(LC(x0) \pm (k+1) * \delta))$
  - Loop is non-terminating if no solution exists
- Naturally extends to multiple variables

Problem Reduction to Quantified SMT Solving: No Conditionals

```
int x = x0;
while (LC(x))
  x = x ± δ;
```

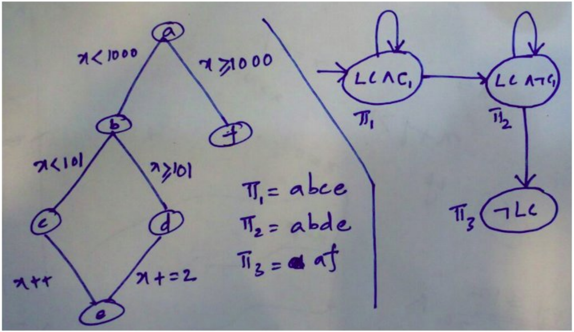


- Closed Form for value of x after k iterations: $\tau^k = x0 \pm k * \delta$
- Transitive Closure τ^* (at termination):
  $x0 \pm k * \delta$ (least) skolem solution to k s.t.
  - $\forall x_0 \exists k : ((LC(x0) \pm k * \delta) \wedge \neg(LC(x0) \pm (k+1) * \delta))$
  - Loop is non-terminating if no solution exists
- Naturally extends to multiple variables
- Synthesis problem decidable (for linear LC) as this belongs to *accelerable loops* [S. Bardin et al]
- Solution for τ^* expressible as a linear function of x_0 and δ

Loops With Conditionals: Example1

```
#include "assert.h"
int main(){
    int x=0;
    while(x<1000){
        if(x<101)
            x++;
        else
            x+=2;
    }
    assert(!x%2)
}
```

Fig1: Program with control flow

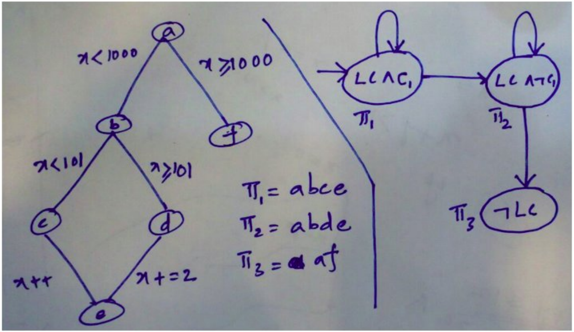


- *Loop-head CFG*: Each transition is a single iteration of loop

Loops With Conditionals: Example1

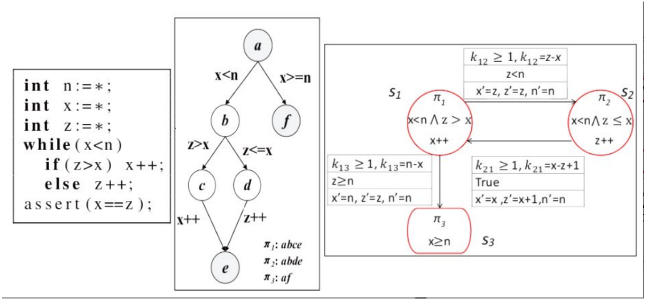
```
#include "assert.h"
int main(){
    int x=0;
    while(x<1000){
        if(x<101)
            x++;
        else
            x+=2;
    }
    assert(!x%2)
}
```

Fig1: Program with control flow



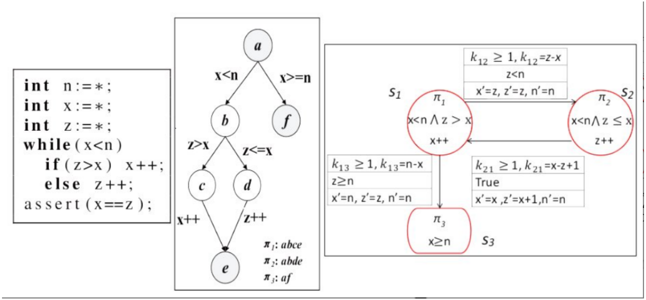
- *Loop-head CFG*: Each transition is a single iteration of loop
- The closed form of τ^* is defined by the set of all feasible *looping-path patterns* ($\langle \pi_{i_1}, \pi_{i_2}, ..., \pi_{i_n} \rangle$) that account for all legal terminating execution traces of loop
- For this example: $\pi_1{}^*\pi_2^*\pi_3$ defines the entire feasible set

Loops With Conditionals: Example2



- CFG has a cycle besides self loops (in S_1 and S_2)

Loops With Conditionals: Example2



- CFG has a cycle besides self loops (in S_1 and S_2)
- The following looping-path patterns define all possible loop execution traces
  - $\pi_1^*(\pi_2\pi_1)^*\pi_3$
  - $\pi_2^*(\pi_1\pi_2)^+\pi_1\pi_3$
  - π_3

Problem Reduction to SMT: Loops with Conditionals

INPUT: Restricted Linear Program:

$X = X_0$;
assume ($\mathrm{Init}(X_0)$);
while ($\mathrm{LC}(X_0)$ {
    if $c_1(X)$ $X = X \pm \Delta_1$;
    else if $c_2(X)$ .....
    else $c_n(X)$ $X = X \pm \Delta_n$;
}

Problem Reduction to SMT: Loops with Conditionals

INPUT: Restricted Linear Program:

$X = X_0$;
assume (Init(X_0));
while (LC(X_0) {
    if $c_1(X)$ $X = X \pm \Delta_1$;
    else if $c_2(X)$ .....
    else $c_n(X)$ $X = X \pm \Delta_n$;
}

Step1 Synthesis of Loop-head CFG: Eliminate all (definitive) infeasible transitions

- $UNSAT(LC(X^{pre}) \wedge X^{post} = X \pm \Delta_i \wedge LC(X^{post}) \wedge BoolCombo_i(c_1(X^{pre}), ..., c_n(X^{pre})) \wedge BoolCombo_j(c_1(X^{post}), ..., c_n(X^{post})))$

Problem Reduction to SMT: Loops with Conditionals

INPUT: Restricted Linear Program:

$X = X_0$;
assume $(\text{Init}(X_0))$;
while $(\text{LC}(X_0)$ $\{$
 if $c_1(X)$ $X = X \pm \Delta_1$;
 else if $c_2(X)$ .....
 else $c_n(X)$ $X = X \pm \Delta_n$;
$\}$

Step1 Synthesis of Loop-head CFG: Eliminate all (definitive) infeasible transitions

- $UNSAT(LC(X^{pre}) \wedge X^{post} = X \pm \Delta_i \wedge LC(X^{post}) \wedge BoolCombo_i(c_1(X^{pre}), ..., c_n(X^{pre})) \wedge BoolCombo_j(c_1(X^{post}), ..., c_n(X^{post})))$

Step2 Generate a complete set of all looping-path patterns in the CGF b/w every pair of start and terminal states.

- Can be done performing a depth-first-search of CFG

Problem Reduction to SMT: Step 3

- For each feasible *looping-path* pattern ($< \psi_1^*, \psi_2^*, ..., \psi_l^* >$ generated in Step-2), synthesize $k_{i_0}, k_{i_1}, ...k_{i_l}$ s.t. the following is valid:
  - $\forall X_0 \exists k_{i_0}, k_{i_1}, ...k_{i_l} : Init(X_0) \wedge$
    $(c_{i_0}(X_0) \wedge X_1 = X_0 + k_{i_0} * \Delta_0 \wedge \neg c_{i_0}(X_1)) \wedge$
    .....
    $(c_{i_l}(X_{l-1}) \wedge X_1 = X_{l-1} + k_{i_l} * \Delta_l \wedge LC(X_l))$

Problem Reduction to SMT: Step 3

- For each feasible *looping-path* pattern ($< \psi_1^*, \psi_2^*, ..., \psi_l^* >$ generated in Step-2), synthesize $k_{i_0}, k_{i_1}, ...k_{i_l}$ s.t. the following is valid:
  - $\forall X_0 \exists k_{i_0}, k_{i_1}, ...k_{i_l} : Init(X_0) \wedge$
    $(c_{i_0}(X_0) \wedge X_1 = X_0 + k_{i_0} * \Delta_0 \wedge \neg c_{i_0}(X_1)) \wedge$
    .....
    $(c_{i_l}(X_{l-1}) \wedge X_1 = X_{l-1} + k_{i_l} * \Delta_l \wedge LC(X_l))$
- To get a closed form, we need to solve for each k_i as a function of X_{i-1}.

Problem Reduction to SMT: Step 3

- For each feasible *looping-path* pattern ($< \psi_1^*, \psi_2^*, ..., \psi_l^* >$ generated in Step-2), synthesize $k_{i_0}, k_{i_1}, ...k_{i_l}$ s.t. the following is valid:
  - $\forall X_0 \exists k_{i_0}, k_{i_1}, ...k_{i_l} : Init(X_0) \wedge$
    $(c_{i_0}(X_0) \wedge X_1 = X_0 + k_{i_0} * \Delta_0 \wedge \neg c_{i_0}(X_1)) \wedge$
    .....
    $(c_{i_l}(X_{l-1}) \wedge X_1 = X_{l-1} + k_{i_l} * \Delta_l \wedge LC(X_l))$
- To get a closed form, we need to solve for each k_i as a function of X_{i-1}.
- **Claim**: A linear solution is guaranteed, if one exists (SAT)
  - Conditions are all conjunctive linear inequalities $\Rightarrow$ monotonic

Problem Reduction to SMT: Step 3

- For each feasible *looping-path* pattern ($< \psi_1^*, \psi_2^*, ..., \psi_l^* >$ generated in Step-2), synthesize $k_{i_0}, k_{i_1}, ...k_{i_l}$ s.t. the following is valid:
  - $\forall X_0 \exists k_{i_0}, k_{i_1}, ...k_{i_l} : Init(X_0) \wedge$
    $(c_{i_0}(X_0) \wedge X_1 = X_0 + k_{i_0} * \Delta_0 \wedge \neg c_{i_0}(X_1)) \wedge$
    .....
    $(c_{i_l}(X_{l-1}) \wedge X_1 = X_{l-1} + k_{i_l} * \Delta_l \wedge LC(X_l))$
- To get a closed form, we need to solve for each k_i as a function of X_{i-1}.
- **Claim**: A linear solution is guaranteed, if one exists (SAT)
  - Conditions are all conjunctive linear inequalities $\Rightarrow$ monotonic
- Final Loop Summary: Disjunction closed forms for all patterns in the complete set (Step 2)

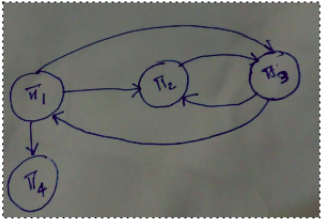
Restrictions on CFG for Decidability

- *Flattable CFG*: The loop-head CFG for our programs must be such that the set of all paths in the CFG can be defined by a finite set of looping patterns each of which is of the following fo
- $< \psi_1^*, \psi_2^*, ..., \psi_l^* >$, s.t. $\psi_i =< \pi_{i_0}^{c_1} ... \pi_{i_n}^{c_n} >$ for constants c
- A sufficient condition for Flattable CFG:
  - *Flat CFG*: Every state in the CFG must be part of at most once cycle
  - CFG is reducible to an equivalent flat CFG
- Claim: The acceleration problem for a restricted linear program (defined earlier) with a flattable CFG is decidable [Bardin, Finkel, et.al]

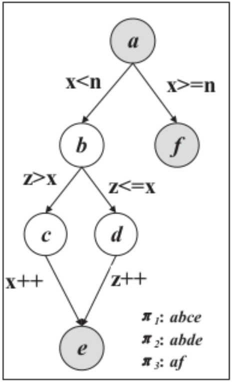
Loops with Non-periodic Cycles: A non-flattable CFG

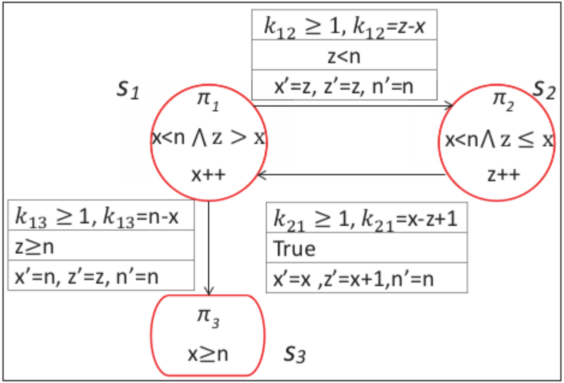
Below is the program where there are multiple interconnected circuits.

```
#include "assert.h"
int main(){
    int k=0;x=0;z=0;
    while(k<100)
        if(k<x)
            k=k+5;
        else
            if(z>x)
                x++;
            else
                z++
    assert(z==x);
}
```



Solving for the sub-loop iteration numbers: k_i





$\pi_1^*(\pi_2\pi_1) * \pi_3$

Solving for the sub-loop iteration numbers: k_i

In the previous example, consider $s_1 \to s_2$. Let k_{12} be the state counter. So,

$$X'_{k_{12}-1} : x' = x + k_{12} - 1, n' = n, z' = z$$

$$X'_{k_{12}} : x' = x + k_{12}, n' = n, z' = z$$

$$cond = (x+k_{12}-1 < n) \wedge \overline{(z > x + k_{12} - 1)} \wedge (x+k_{12} < n) \wedge \overline{(z \leq x + k_{12})}$$

So,

$$\phi_{12} : k_{12} = z - x$$

$$\psi_{12} : z < n$$

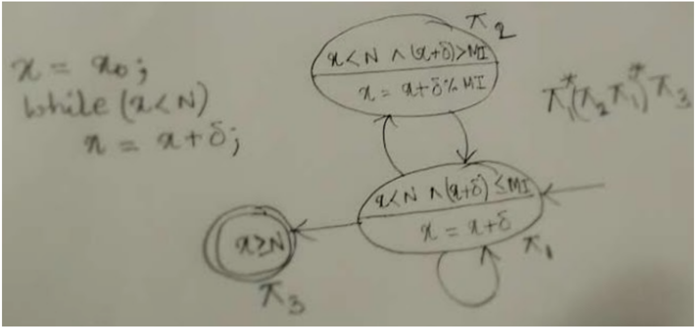
$$U_{12}(X, k_{12}) = \{z, z, n\}$$

The moment we are in s_1 let us say that the initial values of $X = \{x, z, n\}$. After doing the transition $s_1 \to s_2$, the values of the variables will be $\{z, z, n\}$

Some Related Work

- What kind of loops are effectively and precisely Accelerable?
  - Flat acceleration in symbolic model checking [Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Philippe Schnoebelen]
  - PROTEUS: Computing Disjunctive Loop Summary via Path Dependency Analysis. FSE'16 [Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le Xiaohong Li]
  - Simplifying Loop Invariant Generation Using Splitter Predicates. CAV'11 [Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken]

Challenges of Handling Overflows



Problem Reduction to SMT: Conditionals with Overflows

- $\exists l, k_{i_0}, k_{i_1}, ...k_{i_l} : Init(X_0) \wedge$

  $(LC(X_0) \wedge X_1 = (X_0 + k_0 * \Delta_0 \ mod \ MAXINT) \wedge \neg LC(X_1)) \wedge$

  .....

  $(LC(X_{l-1}) \wedge X_1 = (X_{l-1} + k_{l-1} * \Delta_l \ mod \ MAXINT) \wedge \neg LC(X_l))$

- Some Observations:

Problem Reduction to SMT: Conditionals with Overflows

- $\exists l, k_{i_0}, k_{i_1}, ...k_{i_l} : Init(X_0) \wedge$

  $$(LC(X_0) \wedge X_1 = (X_0 + k_0 * \Delta_0 \; mod \; MAXINT) \wedge \neg LC(X_1)) \wedge$$
  $$.....$$

  $$(LC(X_{l-1}) \wedge X_1 = (X_{l-1} + k_{l-1} * \Delta_l \; mod \; MAXINT) \wedge \neg LC(X_l))$$

- Some Observations:
  - Nonlinear SMT solving required due to mod operation
  - For a fixed bit-vector size and for given x_0, δ it is possible to solve for k_i by model enumeration

Problem Reduction to SMT: Conditionals with Overflows

- $\exists l, k_{i_0}, k_{i_1}, ...k_{i_l} : Init(X_0) \wedge$

  $(LC(X_0) \wedge X_1 = (X_0 + k_0 * \Delta_0 \ mod \ MAXINT) \wedge \neg LC(X_1)) \wedge$
  $.....$

  $(LC(X_{l-1}) \wedge X_1 = (X_{l-1} + k_{l-1} * \Delta_l \ mod \ MAXINT) \wedge \neg LC(X_l))$

- Some Observations:
  - Nonlinear SMT solving required due to mod operation
  - For a fixed bit-vector size and for given x_0, δ it is possible to solve for k_i by model enumeration
  - Can we exploit the cyclic algebraic properties of mod operation to eliminate or optimize enumeration?
    - Termination can be checked without enumeration
    - Synthesizing a closed form may require minimization search

Conclusions

- Precise loop acceleration is a hard problem
- Can recent advances in quantifier elimination and skolem function generation help?
- Are there other sub-classes that admit easy solutions?