AUTOMATED ALIGNMENT FOR EQUIVALENCE CHECKING

Ameer Hamza and Grigory Fedyukovich

Florida State University, Tallahassee, FL, USA

Introduction

- Compilers need verification
- Need to prove equivalence of the source and the target
- *Lockstep composition* (requires the same number of steps) is a common approach
- Lockstep composition does not work for complex optimizations
- *Our idea*: preprocees (align) the source and the target such that lockstep composition is applicable

Vectorization Example

Given:

1. Source program
2. Target (vectorized) program
3. A precondition (on inputs): `a=c, b=d, M=N`
4. A postcondition (on outputs): `a=c, b=d, M=N`

Goal:

- Check that the programs are equivalent w.r.t. pre/post-condition

```
assume (M > 0);
int i = 0;
while(i < M*4-1) {
  if (b[0] > 0)
  a[i] = a[i+1] + b[i];
  i++;
}
```

```
assume (N > 0);
if (d[0] > 0) {
  c[0] = c[1] + d[0];
  c[1] = c[2] + d[1];
}
int j = 2;
while(j < N*4-2) {
  if (d[0] > 0) {
    c[j]   = c[j+1] + d[j];
    c[j+1] = c[j+2] + d[j+1];
    c[j+2] = c[j+3] + d[j+2];
    c[j+3] = c[j+4] + d[j+3];
  }
  j += 4;
}
if (d[0] > 0)
c[N*4-2] = c[N*4-1] + d[N*4-2];
```

Programs are not lockstep-composable because:

- Number of iterations are not the same
- Target contains some code before and after the loop

Solution:

1. Create a batch of 4 iterations inside the source loop
2. Move 2 iterations before the loop, 1 iteration after the loop in the source
3. Create a lockstep-composition and reduce to safety verification.

Relational Verification

Given:

1. Two transition systems (i.e., single-loop prorams)
2. A relational precondition pre
3. A relational postcondition $post$
4. **Challenge:** Systems are not necessarily in lockstep

Goal:

- Check if $post$ holds after both programs begin with pre and terminate

Relational Verification $\cong$ Lockstep Composition + Safety of Product

For *Equivalence Checking*,
pre : pairwise equality of inputs $post$: pairwise equality of outputs

Our Approach

Given:

1. Source program
2. 3-phased target program
   - a *pre-phase*: represents few initial iterations
   - a *main-phase*: represents the iterating part of the transition system
   - a *post-phase*: represents few final iterations
3. A precondition pre
4. A postcondition $post$

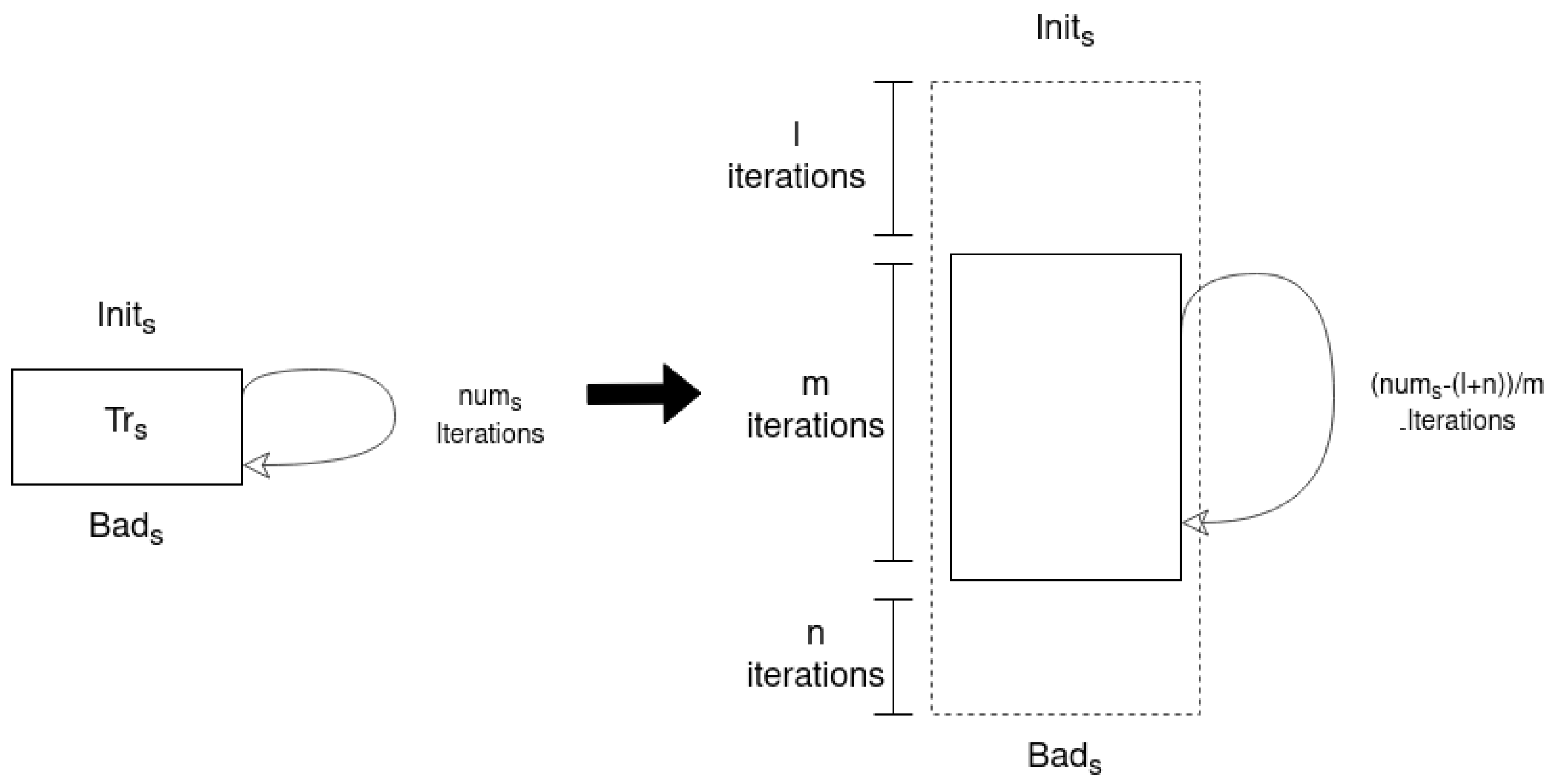
Algorithm:

- Rearrange iterations in the source, given three parameters ℓ, m, n
  - move ℓ iterations before the loop
  - create a batch of m iterations inside the loop
  - move n iterations after the loop
- Create a product of two programs in lockstep composition, w.r.t.
  - a relational precondition π: strongest postcondition of pre and initial iterations in both programs
  - a relational postcondition ϕ: weakest precondition of $post$ and final iterations in both programs

Integers ℓ, m, n can be computed by:

- Estimation of the number of iterations in both programs symbolically
- Solving a first-order logic formula relating number of iterations, a model to which provides info about ℓ, m, n values

Updating the Source (schema)



Reduction to Safety Verification

Given:

1. A product program in lockstep composition
2. Relational pre/post-conditions π and ϕ

Goal:

- Check that if initially π holds, then ϕ holds in the end (i.e., find *safe inductive invariants*); or find a counter-example
- If ϕ holds, report EQUIVALENT, othewise report NONEQUIVALENT

Evaluation

- ALIEN tool implemented on top of invariant synthesizer FREQHORN [1]
- Evaluated on 79 benchmarks from the Test Suite of Vectorization Compilers (TSVC) [2]
- All benchmarks contain a single loop, and (possibly, several) array(s).
- ALIEN solved 79 out of 80 benchmarks.
- Minimum time: 0.74s, maximum time: 12.35s, and average time: 2.32s.

References

[1] Grigory Fedyukovich et al. “Quantified Invariants via Syntax-Guided Synthesis”. In: *CAV, Part I*. Vol. 11561. LNCS. Springer, 2019, pp. 259–277.

[2] Saeed Maleki et al. “An Evaluation of Vectorizing Compilers”. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 372–382.