Verifying the Floating-Point Computation Equivalence of Manually and Automatically Differentiated Code

—

1st International Workshop on Software Correctness for HPC Applications, Nov 12, 2017, CO, USA

Markus Schordan[1], Jan Hückelheim[2], Pei-Hung Lin[1], Harshitha Menon[1]

[1] Lawrence Livermore National Laboratory, CA, USA
[2] Imperial College London

—

External Audience (Unlimited)
LLNL-PRES-741134

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Why reason about floating point equivalence

Two Programs - syntactically different - semantically equal

- Compiler optimizations - optimized vs original program
- Manually optimized code vs original code
- Tools for optimizing programs (.e.g., cache locality)
  - Earlier version of our approach [ISoLA'14]
- Difficulties with floating-point computations:
  - Law of associativity does not hold for floating-point numbers
  - Truncation, Rounding, etc.
  - Source code vs assembly code (generated fp-instructions)
- Problem: in general undecidable

Testcase: Two AD Programs

Algorithmic Differentiation (AD)

- Derivatives of mathematical functions ingredients for many methods
- sensitivity analysis, gradient-based opt., uncertainty quantification
- AD is accurate to machine precision and can be very efficient

Tapenade Generated Code vs Manually Written Code

Source Code	TapAD	ManAD
Loops	yes	no
2-dim arrays	yes	no
1-dim arrays	yes	yes
Addition Op	13	5
Subtraction Op	16	13
Multiplication Op	28	28
Division Op	2	2

See paper for complete source codes of TapAD and ManAD.

Quality metric function in mesh adaption benchmark FeasNewt.

Equivalence Proof Technique

Two Programs - P_1, P_2

1. Evaluate partially: $(P_1, P_2) \rightarrow (P_1', P_2')$
   – Evaluate only integer and pointer operations
2. Rewrite: $(P_1', P_2') \rightarrow (P_1'', P_2'')$
   – Rewrite fp-expressions with fp-semantics preserving rules)
3. Match: $P_1'' \overset{?}{\leftrightarrow} P_2''$
   – Match P_1'' and P_2'' syntactically at source level
4. Assembly code statistics: $(P_1, P_2) \rightarrow (A_1, A_2)$
   – Determine whether certain instructions are generated by compiler in A_1, A_2.

Partial Evaluation (Example Fragment)

```
#define nbdirs 1
for (nd = 0; nd < nbdirs; ++nd) {
  for (ii1 = 0; ii1 < 4; ++ii1)
      matrb[ii1][nd] = 0.0;
  matrb[0][nd] = matrb[0][nd] + 2.0*matr[0]*fb[nd];
  matrb[0][nd] = matrb[0][nd] + matr[3]*gb[nd];
}
```

```
matrb[0][0] = 0.0;
matrb[1][0] = 0.0;
matrb[2][0] = 0.0;
matrb[3][0] = 0.0;
matrb[0][0] = matrb[0][0] + 2.0*matr[0]*fb[0];
matrb[0][0] = matrb[0][0] + matr[3]*gb[0];
```

Current limitation: must produce one execution path, result otherwise unknown.

Rewrite Rules Preserve Exact FP-Semantics

1. $v_i = E_v; \ldots; E = \ldots v_i \ldots \Longrightarrow E = \ldots E_v \ldots$
2. $E_1 + -E_2 \Longrightarrow E_1 - E_2$
3. $-E_1 + E_2 \Longrightarrow E_2 - E_1$
4. $E_1 - (-E_2) \Longrightarrow E_1 + E_2$
5. $-E_1 - E_2 \Longrightarrow -(E_1 + E_2)$
6. $E * (-1.0) \Longrightarrow -E$
7. $(-1.0) * E \Longrightarrow -E$
8. $0.0 - E \Longrightarrow -E$
9. $E + 0.0 \Longrightarrow E$
10. $0.0 + E \Longrightarrow E$
11. $E * 1.0 \Longrightarrow E$
12. $1.0 * E \Longrightarrow E$
13. commutative fp-semantics-aware sort

Rules	(1)	(2) (3)	(4)	(5)	(6) (7)	(8)	(9) (10)	(11) (12)	(13)	Tot
TapAD	70	2	0	2	0	4	12	20	174	284
ManAD	45	0	0	2	10	0	0	0	174	231

Rewrite System: Rules 1-12 are guaranteed to terminate (every rule reduces size of program)

Rule 13: last rule. All (sub)expressions are of the same floating-point type

Equivalence Check

Source Code

- All rewrite rules are fp-semantics-preserving
- Check whether both programs have become syntactically identical
- This implies semantic equivalence at the source level

Assembly Code

- Compute assembly instruction statistics (flow-insensitive)
- Ensure only certain instructions are generated by compiler

Assembly Instruction Counts

	Intel Compiler					
	-O0			-O3		
	TapAD	ManAD	HNEQ	TapAD	ManAD	HNEQ
Arithmetic instructions						
ADDSD xmm1, xmm2/m64	13	8	8	7	10	10
SUBSD xmm1, xmm2/m64	15	13	13	11	11	11
MULSD xmm1, xmm2/m64	28	25	21	27	25	21
DIVSD xmm1, xmm2/m64	2	2	2	2	2	2
Load instructions						
movsd xmm1, m64	90	73	69	4	10	10
movaps xmm1, m64	0	0	0	3	0	0
Store instructions						
movsd m64, xmm1	40	23	23	9	20	20
movaps m64, xmm1	0	0	0	3	0	0
Move instructions						
movapd xmm1, xmm2	0	0	0	0	0	0
movaps xmm1, xmm2	0	0	0	18	15	14

no 64/80 bit conversion, no multiply-add fusion.

Conclusion

Equivalence Checking: Source and Assembly Level

1. Equivalence at the source code level of two programs
2. Only certain instructions used in corresponding assembly codes

---

Conclusion: two programs' computations bit-wise equivalent

If two programs cannot be proven to be equivalent the result is unknown.

Tool: CodeThorn (www.rosecompiler.org)