Translation Validation of Embedded System Specifications using Equivalence Checking

Kunal Banerjee
Supervisors: Prof. C Mandal, Prof. D Sarkar

Dept of Computer Sc & Engg
IIT Kharagpur

Outline

1. Background
2. A formal model and related verification method
3. The method of symbolic value propagation
4. Array Data Dependence Graphs (ADDGs)
5. Future Work

Outline

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: Venit et al., Prelude to Programming: Concepts and Design)

We are not always happy with the programs we write.

Objectives of program optimization:

- To speed-up the computation
- To use less resource, eg. memory, power, etc.

So, we need a **compiler**.

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: Venit et al., Prelude to Programming: Concepts and Design)

We are not always happy with the programs we write.

Objectives of program optimization:

- To speed-up the computation
- To use less resource, eg. memory, power, etc.

So, we need a **compiler**.

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: Venit et al., Prelude to Programming: Concepts and Design)

We are not always happy with the programs we write.

Objectives of program optimization:

- To speed-up the computation
- To use less resource, eg. memory, power, etc.

So, we need a **compiler**.

Can you trust your compiler?

Erroneous loop reversal

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

```
sum = 0;
for (i=N; i>=0; i--) {
  sum = sum + a[i];
} /* a[N] gets accessed */
```

Program: An organized list of instructions that, when executed, causes the computer to behave in a $\boxed{\text{predetermined manner}}$.

A faulty compiler can alter the meaning of a program.

Can you trust your compiler?

Erroneous loop reversal

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

```
sum = 0;
for (i=N; i>=0; i--) {
  sum = sum + a[i];
} /* a[N] gets accessed */
```

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.

A faulty compiler can alter the meaning of a program.

What is the remedy?

- Verified Compiler – All optimized programs will be *correct by construction*.
  Example: CompCert, INRIA
  Limitations:
  - Very hard to formally verify all passes of a compiler.
  - Undecidability of the general problem of program verification restricts the scope of the input language supported by the verified compiler.
- Translation Validation – Each individual translation is followed by a validation phase which verifies that the target code produced correctly implements the source code.
  (This is what we do, i.e., equivalence checking of programs.)

What is the remedy?

- Verified Compiler – All optimized programs will be *correct by construction*.
  Example: CompCert, INRIA
  Limitations:
  - Very hard to formally verify all passes of a compiler.
  - Undecidability of the general problem of program verification restricts the scope of the input language supported by the verified compiler.
- Translation Validation – Each individual translation is followed by a validation phase which verifies that the target code produced correctly implements the source code.
  (This is what we do, i.e., equivalence checking of programs.)

What is the remedy?

- Verified Compiler – All optimized programs will be *correct by construction*.
  Example: CompCert, INRIA
  Limitations:
  - Very hard to formally verify all passes of a compiler.
  - Undecidability of the general problem of program verification restricts the scope of the input language supported by the verified compiler.
- Translation Validation – Each individual translation is followed by a validation phase which verifies that the target code produced correctly implements the source code.
  (This is what we do, i.e., equivalence checking of programs.)

How to check equivalence of programs?

The general problem is undecidable.

McCarthy 91 function

```
int M ( int n ) {
if (n > 100)
  return (n - 10);
else
  return M( M (n + 11) );
}
```

```
int M ( int n ) {
if (n > 100)
  return (n - 10);
else
  return 91;
}
```

Comparing two programs in *totality* is impossible – we should break them into *smaller* chunks.

Granularity of the chunks

Instruction level

```
x = a + b;
y = x - a;
z = y + b;
```

```
x = a + b;
y = b;
z = 2 * b;
```

Granularity of the chunks

Instruction level

```
x = a + b; ✓
y = x - a;
z = y + b;
```

```
x = a + b; ✓
y = b;
z = 2 * b;
```

Granularity of the chunks

Instruction level

`x = a + b;` ✓	`x = a + b;` ✓
`y = x - a;` ×	`y = b;` ×
`z = y + b;`	`z = 2 * b;`

So, instruction level checking can be misleading – let's try at basic block level.

Granularity of the chunks (contd.)

Basic Block level

```
x = a + b;
y = x - a;
z = y + b;
do {
  v = v + x;
  w = y * z;
} while( c1 );
```

```
x = a + b;
y = b;
z = 2 * b;
do {
  v = v + x;
} while( c1 );
w = y * z;
```

Granularity of the chunks (contd.)

Basic Block level

```
x = a + b; ✓
y = x - a; ✓
z = y + b; ✓
do {
  v = v + x;
  w = y * z;
} while( c1 );
```

```
x = a + b; ✓
y = b;     ✓
z = 2 * b; ✓
do {
  v = v + x;
} while( c1 );
w = y * z;
```

Granularity of the chunks (contd.)

Basic Block level

```
x = a + b;  ✓
y = x - a;  ✓
z = y + b;  ✓
do {
  v = v + x;  ×
  w = y * z;  ×
} while( c1 );
```

```
x = a + b;  ✓
y = b;      ✓
z = 2 * b;  ✓
do {
  v = v + x;  ×
} while( c1 );
w = y * z;
```

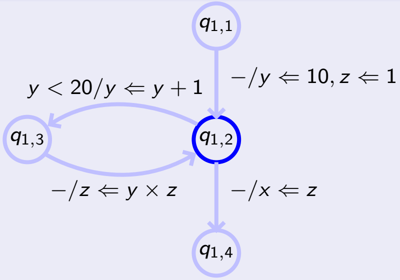
So, checking individual basic blocks is not enough.

Program as a combination of paths

Break a program into smaller chunks — cut loops.

Representing a program using CDFG

```
y := 10;
z := 1;
while ( y < 20 ) {
  y := y + 1;
  z := y × z;
}
x := z;
```



All computations of the program can be viewed as a concatenation of paths.

Example: $p_1.p_3$, $p_1.p_2.p_3$, $p_1.p_2.p_2.p_3$, $p_1.(p_2)^*.p_3$

Outline

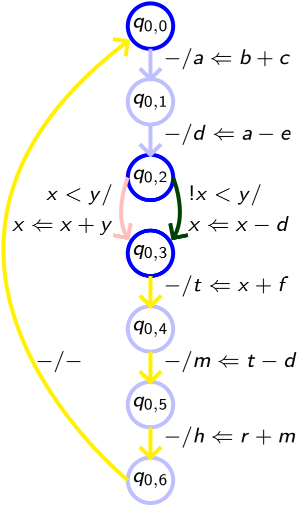
Finite State Machine with Datapath (FSMD)

FSMDs effectively capture both the control flow and the associated data processing of a behaviour.

The FSMD model is a seven tuple $F = \langle Q, q_0, I, V, O, f, h \rangle$:

- Q: Finite set of control states
- q_0: Reset state, i.e. $q_0 \in Q$
- I: Set of input variables
- V: Set of storage variables
- O: Set of output variables
- f: State transition function, i.e. $Q \times 2^S \rightarrow Q$
- h: Update function of the output and the storage variables, i.e. $Q \times 2^S \rightarrow U$
  - U represents a set of storage or output assignments
  - S is a set of arithmetic relations between arithmetic expressions

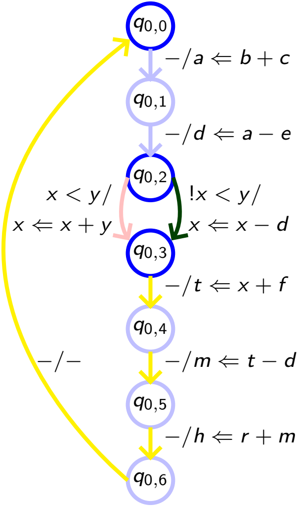
Equivalence checking of FSMDs: A basic example



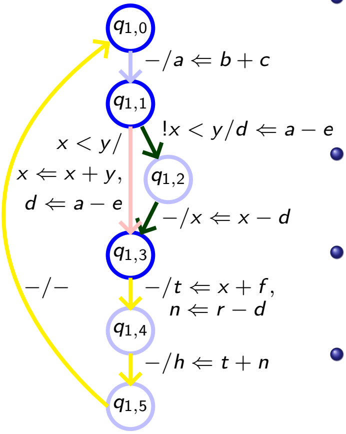
(a)M_0

- Any computation in an FSMD can be represented by a concatenation of its computation paths
- A path is an alternating sequence of states and transitions, starting and ending at cutpoints
- Identification of suitable cutpoints and the path segments between them leads to a finite path cover P_0 in M_0
- For an FSMD, the reset state and all states with multiple incoming/outgoing transitions can be considered as the cutpoints
- Length and number of computations of an FSMD can both be *infinite*
- Since any computation corresponds to a concatenation of paths, it is enough to establish path equivalences

Equivalence checking of FSMDs: A basic example



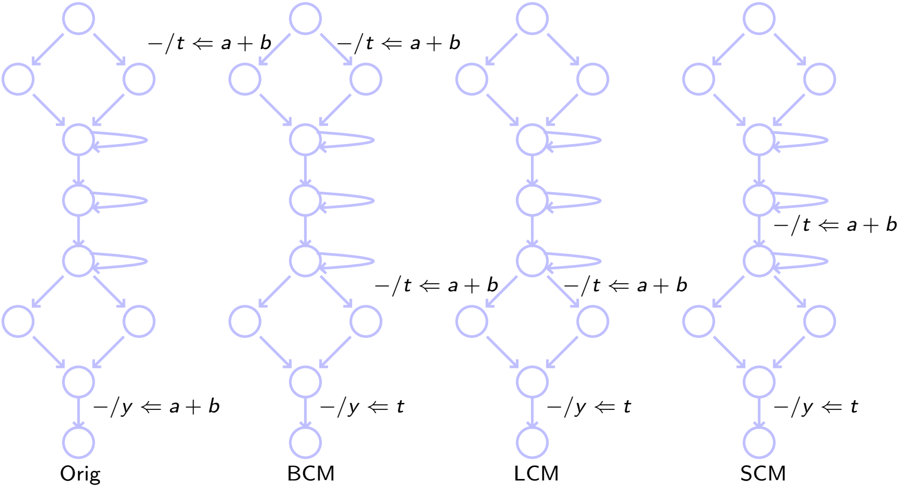
(a) M_0



(b) M_1

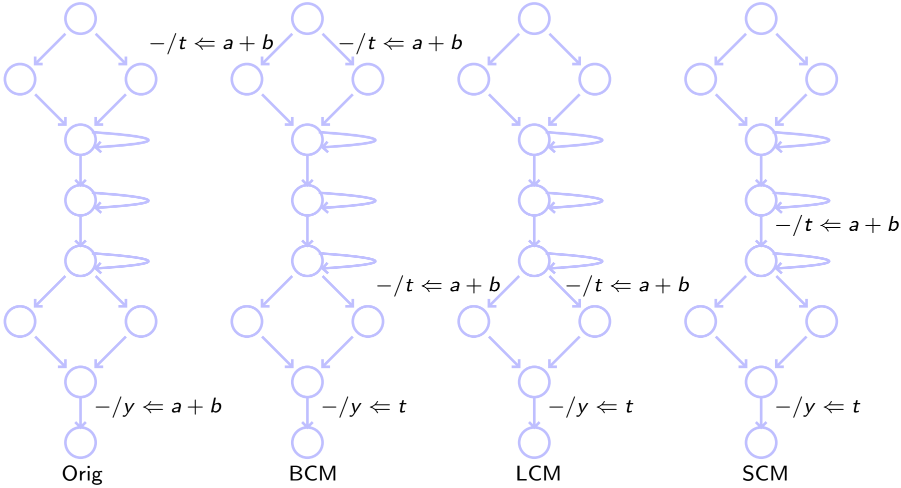
- Two FSMDs M_0 and M_1 are equivalent if for every path in P_0 there is an equivalent path in P_1 and vice versa
- Code transformations can make this job difficult
- Paths may be extended, and the path covers are updated accordingly
- $\{q_{0,0} \xRightarrow{x<y} q_{0,3} \simeq q_{1,0} \xRightarrow{x<y} q_{1,3},\ q_{0,0} \xRightarrow{!x<y} q_{0,3} \simeq q_{1,0} \xRightarrow{!x<y} q_{1,3},\ q_{0,3} \Longrightarrow q_{0,0} \simeq q_{1,3} \Longrightarrow q_{1,0}\}$

A major challenge: Code motions across loops



A path, by definition, cannot be extended beyond a loop.

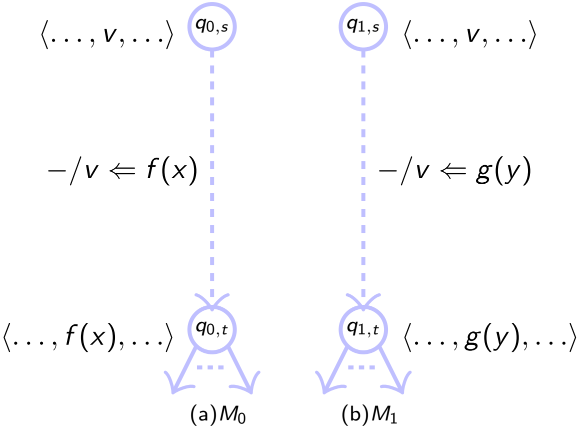
A major challenge: Code motions across loops



A path, by definition, cannot be extended beyond a loop.

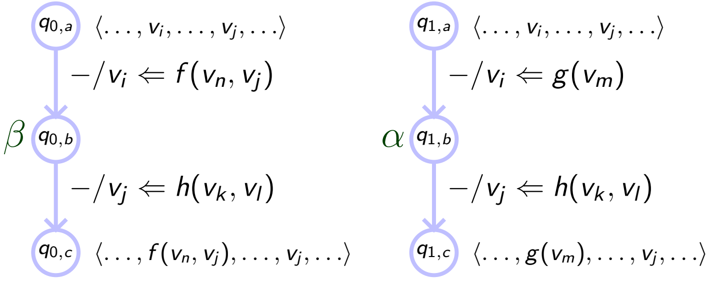
Outline

The method of symbolic value propagation



An example of value propagation

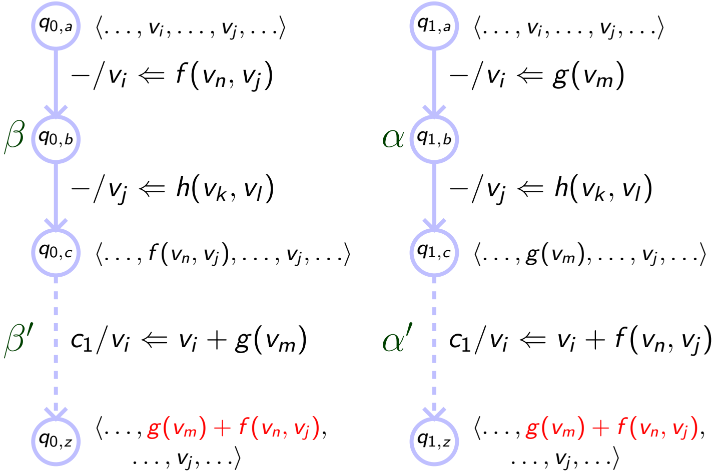
The method of value propagation



(a) M_0 (b) M_1

An example of value propagation with dependency between propagated values

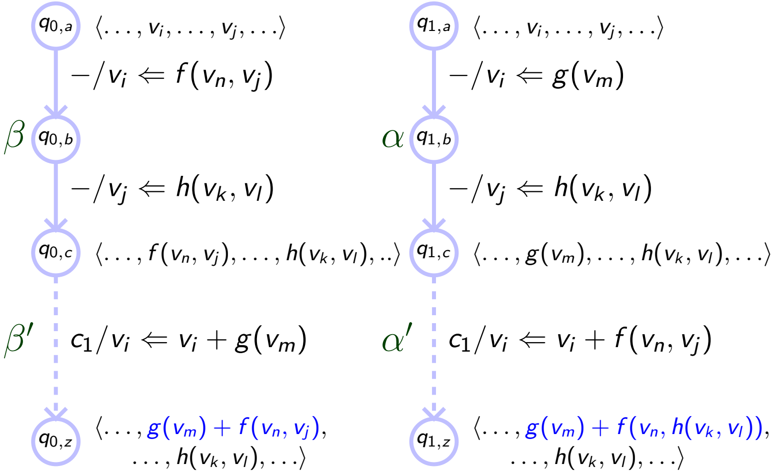
The method of value propagation



(a) M_0 (b) M_1

An *erroneous* decision taken

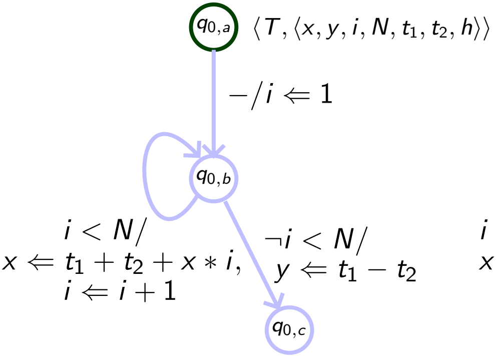
The method of value propagation



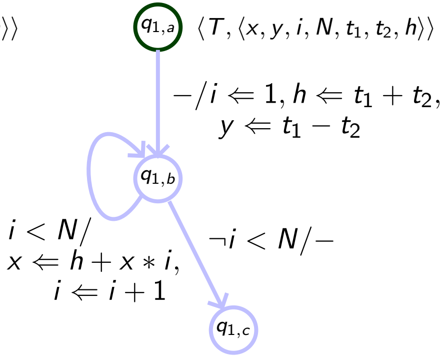
(a) M_0 (b) M_1

Correct decision taken

Equivalence checking of FSMDs using value propagation



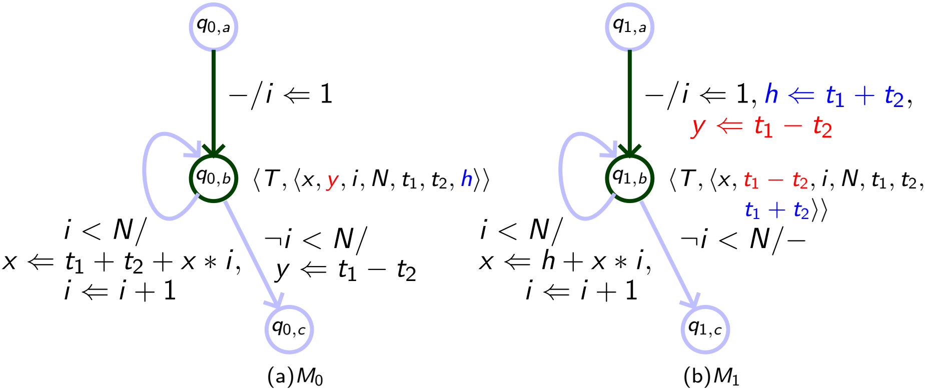
(a) M_0



(b) M_1

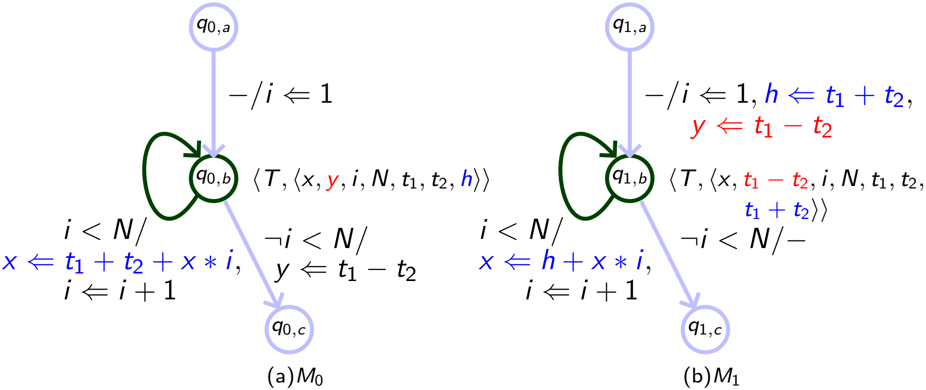
At the reset states

Equivalence checking of FSMDs using value propagation



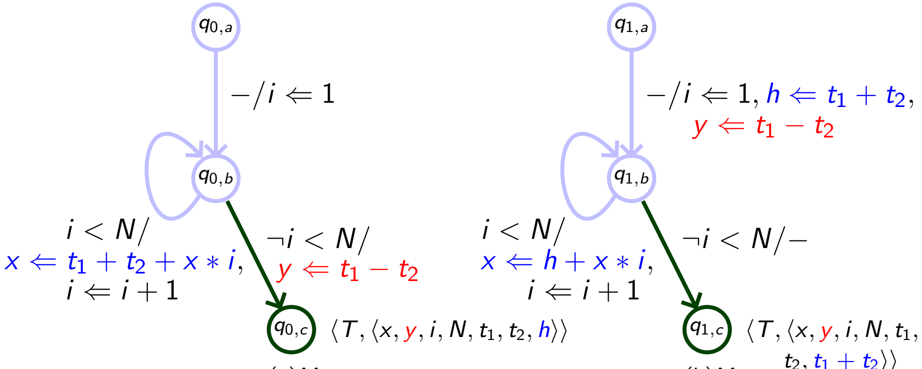
At the beginning of the loops

Equivalence checking of FSMDs using value propagation



At the end of the loops

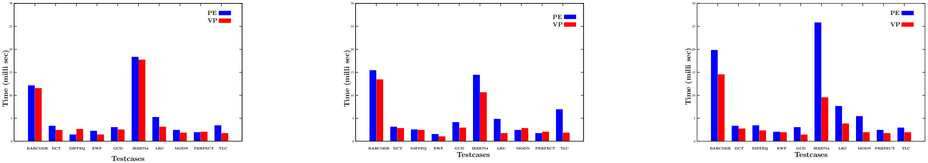
Equivalence checking of FSMDs using value propagation



(a) M_0 (b) M_1

At the end states

Experimental Results



(a) BB-based (b) Path-based (c) SPARK

C. Mandal, and R. M. Zimmer, “A Genetic Algorithm for the Synthesis of Structured Data Paths,” VLSI Design (2000)

R. Camposano, “Path-based Scheduling for Synthesis,” TCAD (1991)

S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, “SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations,” VLSI Design (2003)

Experimental Results (contd.)

Benchmarks	Original FSMD		Transformed FSMD		#Variable		#across	Maximum	Time (ms)	
	#state	#path	#state	#path	com	uncom	loops	mismatch	PE	VP
BARCODE-1	33	54	25	56	17	0	0	3	20.1	16.2
DCT-1	16	1	8	1	41	6	0	6	6.3	3.6
DIFFEQ-1	15	3	9	3	19	3	0	4	5.0	2.6
EWF-1	34	1	26	1	40	1	0	1	4.2	3.6
LCM-1	8	11	4	8	7	2	1	4	–	2.5
IEEE754-1	55	59	44	50	32	3	4	3	–	17.7
LRU-1	33	39	32	38	19	0	2	2	–	4.0
MODN-1	8	9	8	9	10	2	0	3	5.6	2.5
PERFECT-1	6	7	4	6	8	2	2	2	–	0.9
QRS-1	53	35	24	35	25	15	3	19	–	15.9
TLC-1	13	20	7	16	13	1	0	2	9.1	4.1

A major challenge: Loop transformations for arrays

Loop transformations are used extensively to gain speed-ups (parallelization), save memory usage, reduce power, etc.

Loop Fusion

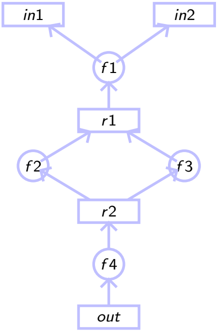
```
for (i=0; i<=7; i++) {
  for (j=0; j<=7; j++) {
    a[i+1][j+1] = F(in);
} }

for (i=0; i<=7; i++) {
  for (j=0; j<=7; j++) {
    b[i][j] = c[i][j];
} }
```

```
for (l1=0; l1<=3; l1++) {
  for (l2=0; l2<=3; l2++) {
    for (l3=0; l3<=1; l3++) {
      for (l4=0; l4<=1; l4++) {
        i = 2*l1 + l3;
        j = 2*l2 + l4;
        a[i+1][j+1] = F(in);
        b[i][j] = c[i][j];
} } } }
```

For array operations, **equivalence of index spaces** has to be ensured as well.

Outline

Array Data Dependence Graphs (ADDGs)



ADDG

- Array data dependence graph (ADDG) model can capture array intensive programs [Shashidhar et al., DATE 2005]
- ADDGs have been used to verify static affine programs
- Equivalence checking of ADDGs can verify loop transformations as well as arithmetic transformations

Two equivalent array-handling programs

Loop fusion and arithmetic simplification

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```

```
for ( i = 1; i <= N; i++ ) {
  z[i] = 2 * a[i];
}
```

```
for ( i = 1; i <= 100; i++ ) { out[i-1] = in[i+1]; }
```

Jargons:

Iteration domain: Domain of the index variable. $\{i \mid 1 \leq i \leq 100\}$

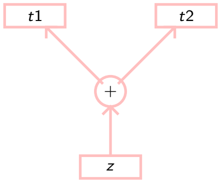
Definition domain: Domain of the (lhs) variable getting defined. $\{i \mid 0 \leq i \leq 99\}$

Operand domain: Domain of the operand variable. $\{i \mid 2 \leq i \leq 101\}$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

$_I M_z = \{k \rightarrow k+1 \mid 0 \leq k \leq N-1\} = {}_I M_{t1} = {}_I M_{t2}$

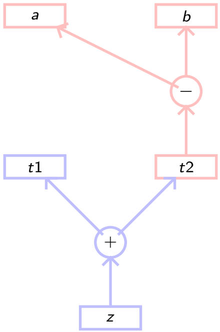
$_z M_{t1} = {}_I M_z{}^{-1} \diamond {}_I M_{t1} = \{k \rightarrow k \mid 1 \leq k \leq N\} = {}_z M_{t2}$

$r_\alpha : z = t1 + t2$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

$_{t2}M_a = \{j \rightarrow j \mid 1 \leq j \leq N\} = {}_{t2}M_b$

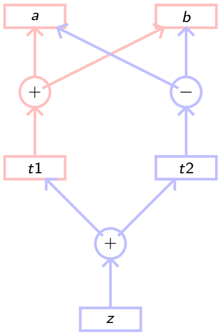
$_{z}M_{t1} = \{k \rightarrow k \mid 1 \leq k \leq N\}$ $\quad$ $_{z}M_a = \{j \rightarrow j \mid 1 \leq j \leq N\} = {}_{z}M_b$

$r_\alpha : z = t1 + (a - b)$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i]
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j]
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

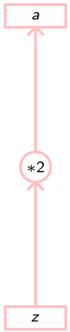
$_{t1}M_a = \{i \to i \mid 1 \leq i \leq N\} = {_{t1}M_b}$

$_{z}M_a = \{k \to k \mid 1 \leq k \leq N\} = {_{z}M_b}$

$r_\alpha : z = (a + b) + (a - b) = 2 * a$ – simplification possible since domains match

Construction of ADDG-2

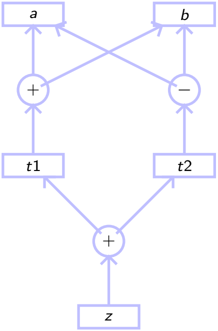
```
for ( i = 1; i <= N; i++ ) {
  z[i] = 2 * a[i];
}
```



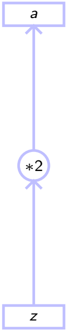
ADDG-2

$_{I}M_{z} = \{i \to i \mid 1 \leq i \leq N\} = {}_{I}M_{a}$
$_{z}M_{a} = \{i \to i \mid 1 \leq i \leq N\}$
$r_{\beta} : z = 2 * a$

Equivalence of ADDGs



ADDG-1



ADDG-2

Two ADDGs are said to be **equivalent** if their characteristic formulae – r_α and r_β, and corresponding mappings between the output arrays wrt input array(s) – $_zM_a^\alpha$ and $_zM_a^\beta$, match.
Hence, these two ADDGs are declared equivalent.

Experimental Results

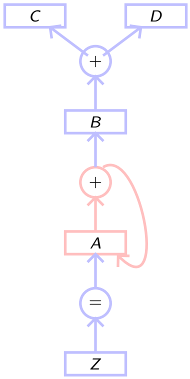
Cases	nests	C lines		loops		arrays		slices		Exec time (sec)			Exec time (sec) - ISA		
		src	trans	src	trans	src	trans	src	trans	eqv	not-eqv1	not-eqv2	eqv	not-eqv1	not-eqv2
SOB1	2	27	19	3	1	4	4	1	1	1.79	0.61	0.75	–	–	–
SOB2	2	27	27	3	3	4	4	1	1	1.85	0.90	0.62	–	–	–
WAVE	1	17	17	1	2	2	2	4	4	6.83	3.81	3.84	0.31	0.18	0.19
LAP1	2	12	21	1	3	2	4	1	1	2.79	0.57	0.65	–	–	–
LAP2	2	12	14	1	1	2	2	1	2	4.82	0.45	0.93	–	–	–
LAP3	2	12	28	1	4	2	4	1	2	9.25	1.14	4.84	0.28	0.19	0.25
ACR1	1	14	20	1	3	6	6	1	1	0.76	0.51	0.72	0.18	0.12	0.13
ACR2	1	24	14	4	1	6	6	2	1	0.98	0.46	0.39	–	–	–
SOR	2	26	22	8	6	11	11	1	1	1.08	0.61	0.62	0.18	0.20	0.17
LIN1	2	13	13	3	3	4	4	2	2	0.62	0.28	0.26	0.12	0.11	0.13
LIN2	2	13	16	3	4	4	4	2	3	0.74	0.20	0.33	0.13	0.12	0.13
LOWP	2	13	28	2	8	2	4	1	2	9.17	0.65	2.90	–	–	–

Verdoolaege et al., "Equivalence checking of static affine programs using widening to handle recurrences," TOPLAS (2012)

Outline

Handling recurrences

```
for ( i = 1; i < N; i++ ) {
  B[i] = C[i] + D[i];
}
for ( i = 1; i < N; i++ ) {
  A[i] = A[i-1] + B[i];
}
for ( i = 1; i < N; i++ ) {
  Z[i] = A[i];
}
```

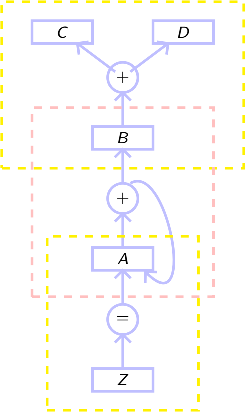


ADDG

Presence of recurrences leads to cycles in the ADDG and hence a closed form representation of r_α cannot be obtained.

Remedy – Separate DAGs from cycles

```
for ( i = 1; i < N; i++ ) {
  B[i] = C[i] + D[i];
}
for ( i = 1; i < N; i++ ) {
  A[i] = A[i-1] + B[i];
}
for ( i = 1; i < N; i++ ) {
  Z[i] = A[i];
}
```



ADDG

Try to establish equivalence of the *separated* ADDG portions.

Reasoning over a finite domain

What's the output?

```
if ( x+1 >= x )
  printf(``Hello'');
else
  printf(``World'');
```

What happens if `x` is the maximum representable integer?

- Output is `World` if modular arithmetic is followed
- Output is `Hello` if saturation arithmetic is followed
- C does not have a defined semantics for overflows, definitions of some other behaviours differ across different standards (ANSIC, C99)

Possible remedy: Bit-tracking.

Reasoning over a finite domain

What's the output?

```
if ( x+1 >= x )
  printf(``Hello'');
else
  printf(``World'');
```

What happens if `x` is the maximum representable integer?

- Output is `World` if modular arithmetic is followed
- Output is `Hello` if saturation arithmetic is followed
- C does not have a defined semantics for overflows, definitions of some other behaviours differ across different standards (ANSIC, C99)

Possible remedy: Bit-tracking.

Reasoning over a finite domain

What's the output?

```
if ( x+1 >= x )
  printf(``Hello'');
else
  printf(``World'');
```

What happens if `x` is the maximum representable integer?

- Output is `World` if modular arithmetic is followed
- Output is `Hello` if saturation arithmetic is followed
- C does not have a defined semantics for overflows, definitions of some other behaviours differ across different standards (ANSIC, C99)

Possible remedy: Bit-tracking.

A word of caution

gcc – Frequently Reported Bugs

There are many reasons why a reported bug doesn't get fixed. It might be difficult to fix, or fixing it might break compatibility. Often, reports get a low priority when there is a simple work-around. In particular, bugs caused by invalid code have a simple work-around: fix the code.
(source: `http://gcc.gnu.org/bugs/#known`)

Publications

Translation Validation

FSMD

- **J1** **K Banerjee**, D Sarkar, C Mandal, "Extending the FSMD Framework for Validating Code Motions of Array-Handling Programs," IEEE Trans on CAD of ICS, (accepted).
- **J2** **K Banerjee**, C Karfa, D Sarkar, C Mandal, "Verification of Code Motion Techniques using Value Propagation," IEEE Trans on CAD of ICS, 2014.
- **C1** **K Banerjee**, C Mandal, D Sarkar, " Extending the Scope of Translation Validation by Augmenting Path Based Equivalence Checkers with SMT Solvers," VDAT, 2014.
- **C2** **K Banerjee**, C Karfa, D Sarkar, C Mandal, "A Value Propagation Based Equivalence Checking Method for Verification of Code Motion Techniques," ISED, 2012.

ADDG

- **J3** C Karfa, **K Banerjee**, D Sarkar, C Mandal, "Verification of Loop and Arithmetic Transformations of Array-Intensive Behaviours," IEEE Trans on CAD of ICS, 2013.
- **C3** **K Banerjee**, "An Equivalence Checking Mechanism for Handling Recurrences in Array-Intensive Programs," POPL (student poster), (accepted).

Publications (contd.)

C4 C Karfa, **K Banerjee**, D Sarkar, C Mandal, "Experimentation with SMT Solvers and Theorem Provers for Verification of Loop and Arithmetic Transformations," I-CARE, 2013 (received *Best Paper Award*).

C5 C Karfa, **K Banerjee**, D Sarkar, C Mandal, "Equivalence Checking of Array-Intensive Programs," ISVLSI, 2011.

PRES+ (a parallel model of computation)

C6 S Bandyopadhyay, **K Banerjee**, D Sarkar, C Mandal, "Translation Validation for PRES+ Models of Parallel Behaviours via an FSMD Equivalence Checker," VDAT, 2012.

Other areas of my research interest:

- Automatic Program Correction and Evaluation
- Secure Hardware Design to Counter Power Analysis Attacks

Thank you!

☞ http://cse.iitkgp.ac.in/~kunban/
✉ kunalb@cse.iitkgp.ernet.in