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

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
**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 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**

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

```c
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

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Instruction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = a + b; )</td>
<td>( x = a + b; )</td>
</tr>
<tr>
<td>( y = x - a; )</td>
<td>( y = b; )</td>
</tr>
<tr>
<td>( z = y + b; )</td>
<td>( z = 2 \times b; )</td>
</tr>
</tbody>
</table>
Granularity of the chunks

**Instruction level**

| x = a + b; ✓ | x = a + b; ✓ |
| y = x - a; | y = b; |
| z = y + b; | z = 2 * b; |
Granularity of the chunks

<table>
<thead>
<tr>
<th>Instruction level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = a + b;) ✓</td>
<td>(x = a + b;) ✓</td>
</tr>
<tr>
<td>(y = x - a;) ×</td>
<td>(y = b;) ×</td>
</tr>
<tr>
<td>(z = y + b;)</td>
<td>(z = 2 \times b;)</td>
</tr>
</tbody>
</table>

So, instruction level checking can be misleading – let’s try at basic block level.
## Granularity of the chunks (contd.)

### Basic Block level

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

```plaintext
x = a + b;
y = b;
z = 2 * b;
do {
    v = v + x;
} while( c1 );
w = y * z;
```
Granularity of the chunks (contd.)

<table>
<thead>
<tr>
<th>Basic Block level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = a + b$; ✓</td>
</tr>
<tr>
<td>$y = x - a$; ✓</td>
</tr>
<tr>
<td>$z = y + b$; ✓</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$v = v + x$;</td>
</tr>
<tr>
<td>$w = y * z$;</td>
</tr>
<tr>
<td>} while( c1 );</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = a + b$; ✓</td>
</tr>
<tr>
<td>$y = b$; ✓</td>
</tr>
<tr>
<td>$z = 2 * b$; ✓</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$v = v + x$;</td>
</tr>
<tr>
<td>} while( c1 );</td>
</tr>
<tr>
<td>$w = y * z$;</td>
</tr>
</tbody>
</table>
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 );

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

\[
y := 10; \\
z := 1; \\
while ( y < 20 ) \{ \\
\quad y := y + 1; \\
\quad z := y \times z; \\
\} \\
x := z;
\]

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

Example: \( p_1 \cdot p_3, p_1 \cdot p_2 \cdot p_3, p_1 \cdot p_2 \cdot p_2 \cdot p_3, p_1 \cdot (p_2)^* \cdot p_3 \)
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
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
A formal model and related verification method

Equivalence checking of FSMDs: A basic example

- 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 \textit{infinite}
- Since any computation corresponds to a concatenation of paths, it is enough to establish path equivalences
A formal model and related verification method

Equivalence checking of FSMDs: A basic example

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.

\[
\begin{align*}
q_0,0 & \xrightarrow{-/a \Leftarrow b + c} q_0,1 \\
q_0,1 & \xrightarrow{-/d \Leftarrow a - e} q_0,2 \\
q_0,2 & \xrightarrow{x < y/d \Leftarrow a - e} q_0,3 \\
q_0,3 & \xrightarrow{x < y} q_0,4 \\
q_0,4 & \xrightarrow{-/t \Leftarrow x + f} q_0,5 \\
q_0,5 & \xrightarrow{-/m \Leftarrow t - d} q_0,6 \\
q_1,0 & \xrightarrow{-/a \Leftarrow b + c} q_1,1 \\
q_1,1 & \xrightarrow{x < y/d \Leftarrow a - e} q_1,2 \\
q_1,2 & \xrightarrow{x < y} q_1,3 \\
q_1,3 & \xrightarrow{-/t \Leftarrow x + f, \; n \Leftarrow r - d} q_1,4 \\
q_1,4 & \xrightarrow{-/h \Leftarrow r + m} q_1,5 \\
q_1,5 & \xrightarrow{-/h \Leftarrow t + n} q_1,6
\end{align*}
\]

{ $q_0,0 \xLeftarrow{x<y} q_0,3 \simeq q_1,0 \xLeftarrow{x<y} q_1,3$, $q_0,0 \xLeftarrow{x<y} q_1,0 \simeq q_1,3$, $q_0,3 \xLeftarrow{x<y} q_0,6 \simeq q_0,0 \simeq q_1,3 \Rightarrow q_1,0$ }
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

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
The method of symbolic value propagation

An example of value propagation

\[ \langle \ldots, v, \ldots \rangle \xrightarrow{q_0,s} \langle \ldots, v, \ldots \rangle \]

\[ -/v \leftarrow f(x) \]

\[ \langle \ldots, f(x), \ldots \rangle \xrightarrow{q_0,t} \]

\[ \langle \ldots, g(y), \ldots \rangle \xrightarrow{q_1,t} \]

\[ \langle \ldots, g(y), \ldots \rangle \xrightarrow{q_1,t} \]

(a) \( M_0 \)

(b) \( M_1 \)
The method of value propagation

An example of value propagation with dependency between propagated values
The method of value propagation

The method of symbolic value propagation

The method of value propagation

\[ q_{0,a} \langle \ldots, v_i, \ldots, v_j, \ldots \rangle \]
\[-/v_i \leftarrow f(v_n, v_j) \]
\[ \beta \]
\[ q_{0,b} \]
\[-/v_j \leftarrow h(v_k, v_l) \]
\[ q_{0,c} \langle \ldots, f(v_n, v_j), \ldots, v_j, \ldots \rangle \]
\[ \beta' \]
\[ c_1/v_i \leftarrow v_i + g(v_m) \]
\[ q_{0,z} \langle \ldots, g(v_m) + f(v_n, v_j), \ldots, v_j, \ldots \rangle \]
\[ (a) M_0 \]

\[ q_{1,a} \langle \ldots, v_i, \ldots, v_j, \ldots \rangle \]
\[-/v_i \leftarrow g(v_m) \]
\[ \alpha \]
\[ q_{1,b} \]
\[-/v_j \leftarrow h(v_k, v_l) \]
\[ q_{1,c} \langle \ldots, g(v_m), \ldots, v_j, \ldots \rangle \]
\[ \alpha' \]
\[ c_1/v_i \leftarrow v_i + f(v_n, v_j) \]
\[ q_{1,z} \langle \ldots, g(v_m) + f(v_n, v_j), \ldots, v_j, \ldots \rangle \]
\[ (b) M_1 \]

An erroneous decision taken

Kunal Banerjee (IIT Kharagpur)
The method of symbolic value propagation

The method of value propagation

$q_0, a \langle \ldots, v_i, \ldots, v_j, \ldots \rangle$

$- / v_i \leftarrow f(v_n, v_j)$

$\beta$ $q_0, b$

$- / v_j \leftarrow h(v_k, v_l)$

$q_0, c \langle \ldots, f(v_n, v_j), \ldots, h(v_k, v_l), \ldots \rangle$

$q_1, a \langle \ldots, v_i, \ldots, v_j, \ldots \rangle$

$- / v_i \leftarrow g(v_m)$

$\alpha$ $q_1, b$

$- / v_j \leftarrow h(v_k, v_l)$

$q_1, c \langle \ldots, g(v_m), \ldots, h(v_k, v_l), \ldots \rangle$

$\beta' c_1 / v_i \leftarrow v_i + g(v_m)$

$q_0, z \langle \ldots, g(v_m) + f(v_n, v_j), \ldots, h(v_k, v_l), \ldots \rangle$

$q_1, z \langle \ldots, g(v_m) + f(v_n, h(v_k, v_l)), \ldots, h(v_k, v_l), \ldots \rangle$

$\alpha' c_1 / v_i \leftarrow v_i + f(v_n, v_j)$

$(a) M_0$

$(b) M_1$

Correct decision taken
Equivalence checking of FSMDs using value propagation

At the reset states

(a) $M_0$

(b) $M_1$
The method of symbolic value propagation

Equivalence checking of FSMDs using value propagation

At the beginning of the loops

Kunal Banerjee  (IIT Kharagpur)
Equivalence checking of FSMDs using value propagation

\[ q_{0,a} \]
\[ -i \leftarrow 1 \]
\[ q_{0,b} \]
\[ \langle T, \langle x, y, i, N, t_1, t_2, h \rangle \rangle \]
\[ i < N \quad x \leftarrow t_1 + t_2 + x \times i, \quad i \leftarrow i + 1 \]
\[ q_{0,c} \]
\[ (a)M_0 \]

\[ q_{1,a} \]
\[ -i \leftarrow 1, \quad h \leftarrow t_1 + t_2, \quad y \leftarrow t_1 - t_2 \]
\[ q_{1,b} \]
\[ \langle T, \langle x, t_1 - t_2, i, N, t_1, t_2, t_1 + t_2 \rangle \rangle \]
\[ i < N \quad x \leftarrow h + x \times i, \quad i \leftarrow i + 1 \]
\[ q_{1,c} \]
\[ (b)M_1 \]

At the end of the loops
Equivalence checking of FSMDs using value propagation

At the end states
The method of symbolic value propagation

Experimental Results

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


Experimental Results (contd.)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Original FSMD</th>
<th>Transformed FSMD</th>
<th>#Variable</th>
<th>#across</th>
<th>Maximum</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#state</td>
<td>#path</td>
<td>#state</td>
<td>#path</td>
<td>com</td>
<td>uncom</td>
</tr>
<tr>
<td>BARCODE-1</td>
<td>33</td>
<td>54</td>
<td>25</td>
<td>56</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>DCT-1</td>
<td>16</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>41</td>
<td>6</td>
</tr>
<tr>
<td>DIFFEQ-1</td>
<td>15</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>EWF-1</td>
<td>34</td>
<td>1</td>
<td>26</td>
<td>1</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>LCM-1</td>
<td>8</td>
<td>11</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>IEEE754-1</td>
<td>55</td>
<td>59</td>
<td>44</td>
<td>50</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>LRU-1</td>
<td>33</td>
<td>39</td>
<td>32</td>
<td>38</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>MODN-1</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>PERFECT-1</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>QRS-1</td>
<td>53</td>
<td>35</td>
<td>24</td>
<td>35</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>TLC-1</td>
<td>13</td>
<td>20</td>
<td>7</td>
<td>16</td>
<td>13</td>
<td>1</td>
</tr>
</tbody>
</table>
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**

```plaintext
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];
    }
}
```

```plaintext
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

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
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**

```plaintext
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 <= 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. \( \{ i \mid 2 \leq i \leq 101 \} \)
Array Data Dependence Graphs (ADDGs)

Construction of ADDG-1

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

```c
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];
}
```

$IM_z = \{ k \rightarrow k + 1 \mid 0 \leq k \leq N - 1 \} = IM_{t1} = IM_{t2}$

$zM_{t1} = IM_z^{-1} \odot IM_{t1} = \{ k \rightarrow k \mid 1 \leq k \leq N \} = zM_{t2}$

$r_\alpha : z = t1 + t2$
Construction of ADDG-1

Array Data Dependence Graphs (ADDGs)

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];
}
```

$\begin{align*}
t_2 M_a &= \{ j \rightarrow j \mid 1 \leq j \leq N \} = t_2 M_b \\
z M_{t1} &= \{ k \rightarrow k \mid 1 \leq k \leq N \} \\
z M_a &= \{ j \rightarrow j \mid 1 \leq j \leq N \} = z M_b \\
r_\alpha : z &= t_1 + (a - b)
\end{align*}$
Construction of ADDG-1

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

```c
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];
}
```

\[ t_1 M_a = \{ i \rightarrow i \mid 1 \leq i \leq N \} = t_1 M_b \]

\[ z M_a = \{ k \rightarrow k \mid 1 \leq k \leq N \} = z M_b \]

\[ r_\alpha : z = (a + b) + (a - b) = 2 \times a - \text{simplification possible since domains match} \]
**Construction of ADDG-2**

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

\[ IM_z = \{ i \rightarrow i \mid 1 \leq i \leq N \} = IM_a \]
\[ zM_a = \{ i \rightarrow i \mid 1 \leq i \leq N \} \]
\[ r_\beta : z = 2 \ast a \]
Two ADDGs are said to be equivalent if their characteristic formulae – $r_\alpha$ and $r_\beta$, and corresponding mappings between the output arrays wrt input array(s) – $z M^\alpha_a$ and $z M^\beta_a$, match. Hence, these two ADDGs are declared equivalent.
### Experimental Results

<table>
<thead>
<tr>
<th>Cases</th>
<th>C lines</th>
<th>loops</th>
<th>arrays</th>
<th>slices</th>
<th>Exec time (sec)</th>
<th>Exec time (sec) - ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nests</td>
<td>src trans</td>
<td>src trans</td>
<td>src trans</td>
<td>src trans</td>
<td>eqv</td>
</tr>
<tr>
<td>SOB1</td>
<td>2</td>
<td>27</td>
<td>19</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>SOB2</td>
<td>2</td>
<td>27</td>
<td>27</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>WAVE</td>
<td>1</td>
<td>17</td>
<td>17</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>LAP1</td>
<td>2</td>
<td>12</td>
<td>21</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>LAP2</td>
<td>2</td>
<td>12</td>
<td>14</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LAP3</td>
<td>2</td>
<td>12</td>
<td>28</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>ACR1</td>
<td>1</td>
<td>14</td>
<td>20</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>ACR2</td>
<td>1</td>
<td>24</td>
<td>14</td>
<td>4</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>SOR</td>
<td>2</td>
<td>26</td>
<td>22</td>
<td>8</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>LIN1</td>
<td>2</td>
<td>13</td>
<td>13</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>LIN2</td>
<td>2</td>
<td>13</td>
<td>16</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>LOWP</td>
<td>2</td>
<td>13</td>
<td>28</td>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

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
**Handling recurrences**

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

Presence of recurrences leads to cycles in the ADDG and hence a closed form representation of $r_{\alpha}$ 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++) {
}
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?

```c
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?

```c
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?

```c
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


**C1** K Banerjee, C Mandal, D Sarkar, “Extending the Scope of Translation Validation by Augmenting Path Based Equivalence Checkers with SMT Solvers,” VDAT, 2014.


ADDG


**C3** K Banerjee, “An Equivalence Checking Mechanism for Handling Recurrences in Array-Intensive Programs,” POPL (student poster), (accepted).
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*).


**PRES+** (a parallel model of computation)


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