Accurate Static Branch Prediction by Value Range Propagation

Jason R. C. Patterson
School of Computing Science Queensland University of Technology

Group 14: Reena Dhankani, Junhwan Kim, Zachary Papanastasopoulos, Yuxi Xie

Motivation

Approaches to obtaining branch prediction data:

- Heuristics based on nesting and coding styles
  - Based on assumptions
  - Poor accuracy
- Real execution profiling
  - Require programmer intervention
  - Time consuming
  - Can be impractical / inapplicable
- Source code hints supplied by the programmer
  - Require programmer intervention
  - Inconvenient
  - Potentially more inaccurate

Goal: Predict branching behavior as accurately as execution profiling and as fast as static analysis, without programmer intervention

Introduction

Constant Propagation

- Identify expressions that are constant
- Evaluate constant expressions at compile time
- Propagate information around the control flow graph

Value Range Propagation

- Value ranges are propagated rather than constants
- Determine weighted range of values an expression can have during program execution
- Examine weighted value range

Key Contribution

- Objective of branch prediction is to determine likelihood of taking each conditional branch
- Heuristic methods mark each branch as “taken” or “not taken”
- Useful but not ideal

Value range propagation produces probabilities.

- Out edges of each conditional branch are marked with probabilities
- More accurately assess degree of speculation

Component #1: Value Range Updates

Value Range: P[L : U : S]

1 [4:4:0] = "100% chance of being 4"
1 [0:8:2] = "100% chance of being a value in {0, 2, 4, 6, 8}"

Operations r3 = r1 + r2

r1: { 0.7[32:256:1], **0.3[3:21:3]** }
+ r2: { **0.6[16:100:4]**, **0.4[8:8:0]** }

r3: { 0.42[48: 356: 1], 0.28[40: 264: 1]
0.18[19: 121: 1], **0.12[11:29:3]** }

0.3 [3 : 21 : **3**]
+ 0.6 [16 : 100 : **4**]

0.18 [19 : 121 : **1**]

30% chance r1 in {3, 6, 9, 12, 15, 18, 21}
60% chance r2 in {16, 20, 24, ……, 96, 100}

18% chance r3 in {19, **20, 21**, ……, 120, 121}

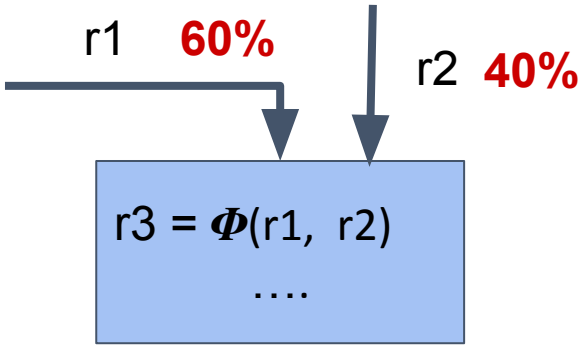
Largest common divisor

Information Loss! The output is generalized/ over-estimated

Component #1: Value Range Updates

Phi nodes
r3 = Φ(r1, r2)

r1: { 1[32:42:1]}
r2: { 0.6[0:10:2], 0.4[30:30:0] }

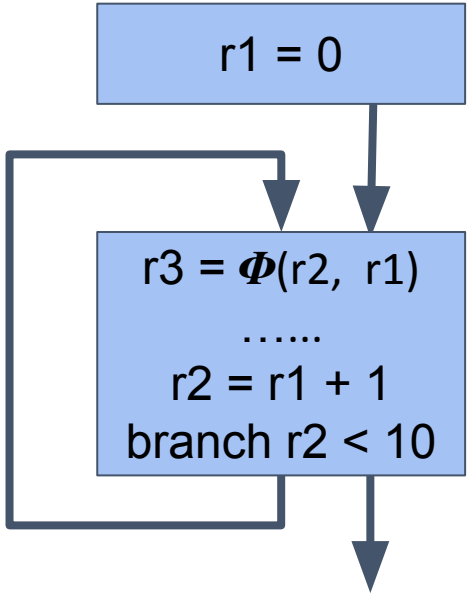


r3 has 60% chance to be r1 and 40% chance to be r2

r3: { **0.6***1[32:42:1], **0.4***0.6[0:10:2], **0.4***0.4[30:30:0] }
= { **0.6**[32:42:1], **0.24**[0:10:2], **0.16**[30:30:0] }

Component #2: Loop Carried Expressions

- When at least one in-edge to a phi-node is a backedge
- Derivation of how the variable may be computed from its previous value and the values of other variables.
- In practice, use templates to to identify simple inductive derivations such as loop counters



Template:

(old value) = (constant)
(new value) = (old value) + (constant)
branch (new value) (operator) (constant)

r1 = 0
r2= r1 + 1
branch r2 < 10

r3: { 1[0 : 10 : 1] }

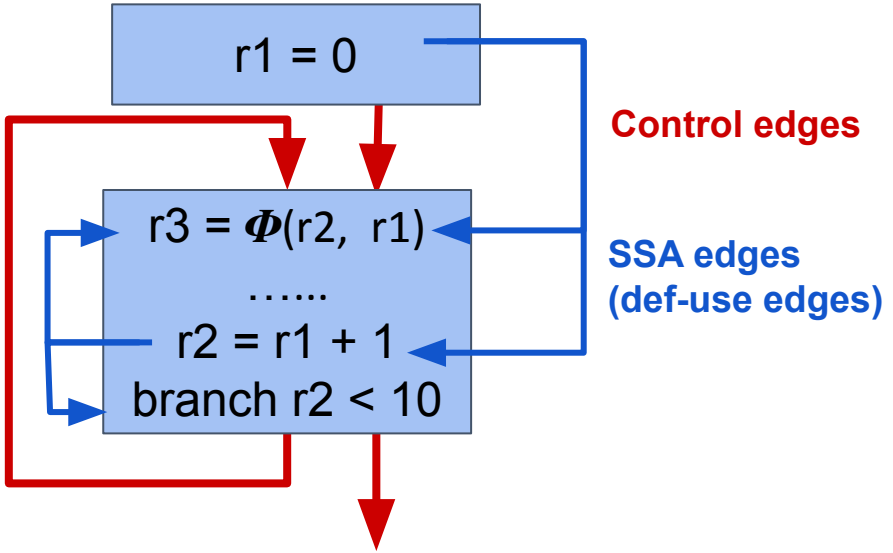
Component #3: CFG and Queues

- Control flow graph in **SSA form**

- **FlowWorkList:**
  - Add **control edges** to successor basic blocks when the current branch probability is updated

- **SSAWorkList:**
  - Add **def-use edges** if the value range of the def is updated

→ ensures all values ranges (and thus the branch probabilities) are updated until convergence



VRP Algorithm Details

```
FlowWorkList = {All out-edges from the start node}
SSAWorkList = ∅
value range probabilities = 0
value ranges = T
Probability of out-edges from the start node = 1

While (True):
    If (!FlowWorkList && !SSAWorkList): break
    Take next item i from either one of the worklists
```

VRP Algorithm Details

```
While (True):
    If (!FlowWorkList && !SSAWorkList): break
    Take next item i from either one of the worklists
    If (i is control flow edge):
        Visit target BB n that i points to
        If (visited[n]):
            Evaluate only the phi nodes in n
        Else:
            Evaluate every expression in n
        visited[n] = 1
    Else If (i is SSA edge):
        Evaluate the expression that i points to
```

VRP Algorithm Details

```
Evaluate(Expression e):
    If (e is loop carried && !impossible_to_derive[e]):
        Attempt to derive value range for e
        If (success):
            loop_derived[e] = 1
            If (new value range != old value range):
                Add the SSA edge starting at that expression e
                to the SSAWorkList
        Else:
            impossible_to_derive[e] = 1
```

VRP Algorithm Details

```
Evaluate(Expression e):
    ...
    Else If (e is phi-node && !loop_derived[e]):
        Evaluate e by merging value ranges according to the
        branch probability of each e's in-edges

        If (new value range != old value range):
            Add the SSA edge starting at that expression e to
            the SSAWorkList
```

VRP Algorithm Details

```
Evaluate(Expression e):
    ...
    Else If (e is an expression && !loop_derived[e]):
        Evaluate e by performing value range operations

        If (new value range != old value range):
            Add the SSA edge starting at that expression e to
            the SSAWorkList
```

VRP Algorithm Details

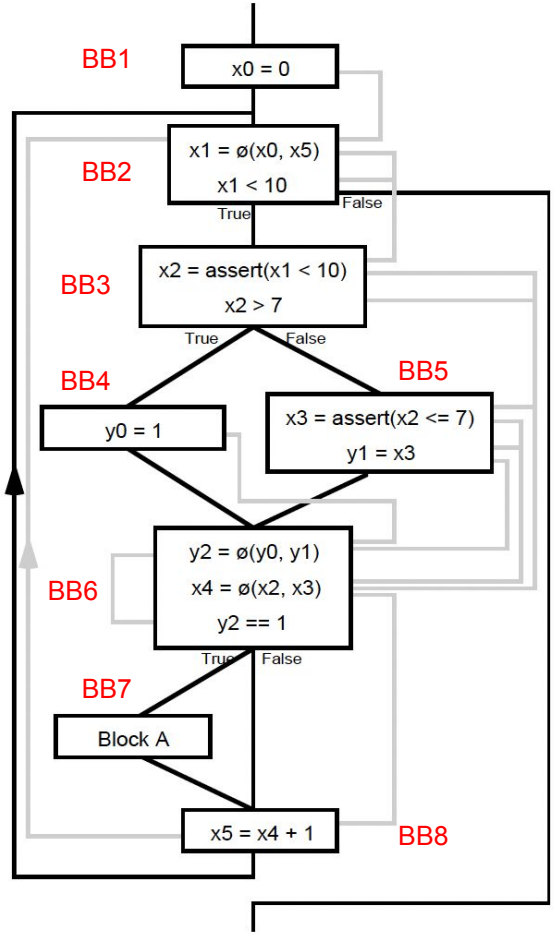
```
Evaluate(Expression e):
    ...
    Else If (e is a conditional branch && !loop_derived[e]):
        Determine the probability p of taking the branch by
        examining the relevant variable's value range

        If (new p != old p):
            Mark the out-edges with new probabilities
            Add out-edges to the FlowWorkList
```

Execution Example

Value Range:

x0	{0[T]}
x1	{0[T]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

```
FlowWorkList = {start→BB1}
SSAWorkList = ∅
```



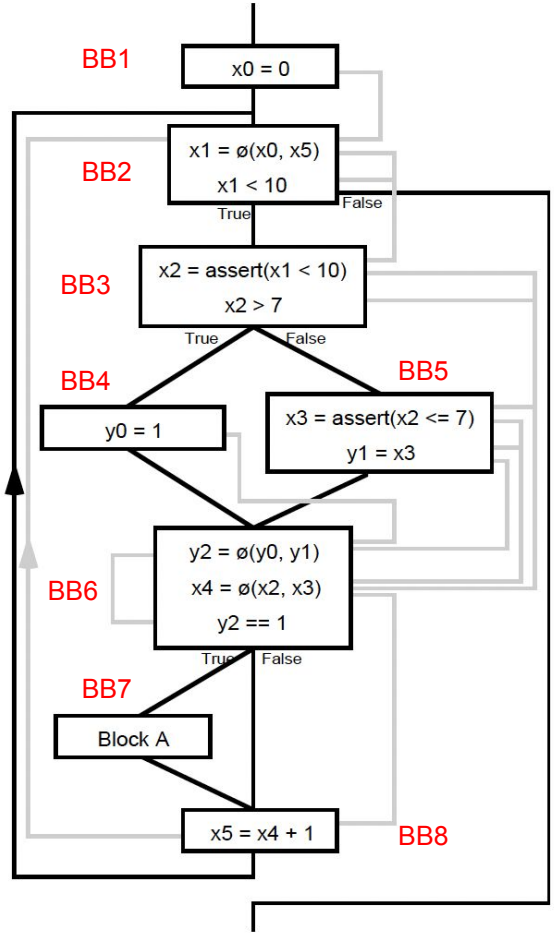
Execution Example

Value Range:

x0	{0[T]}
x1	{0[T]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

```
FlowWorkList = ∅
SSAWorkList = ∅
```

```
Item i = start→BB1
visited[BB1] != 1
Evaluate all expressions in BB1
```



Execution Example

Value Range:

x0	**{1[0:0:0]}**
x1	{0[T]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

FlowWorkList = ∅

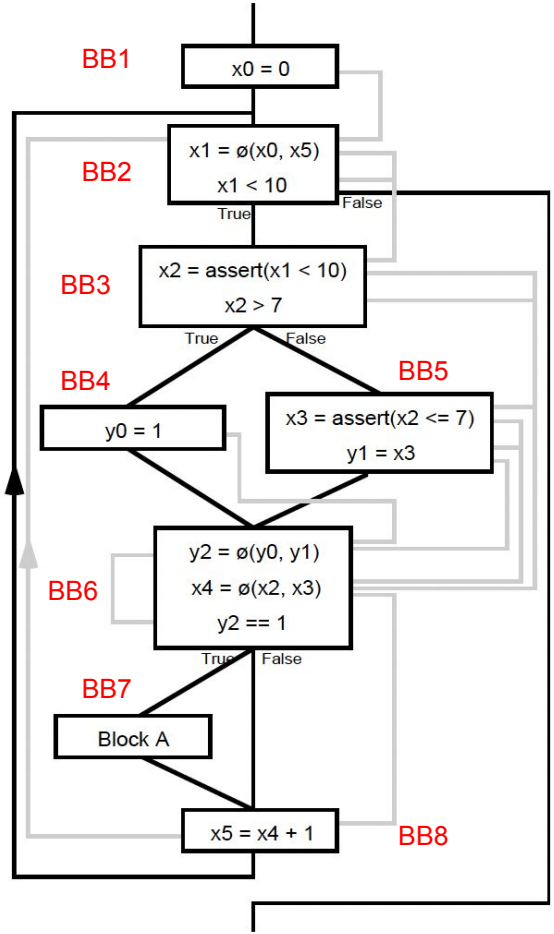
SSAWorkList = **{x1=Φ(x0, x5)}**

Item i = start→BB1

visited[BB1] != 1

Evaluate all expressions in BB1

    Evaluate x0=0

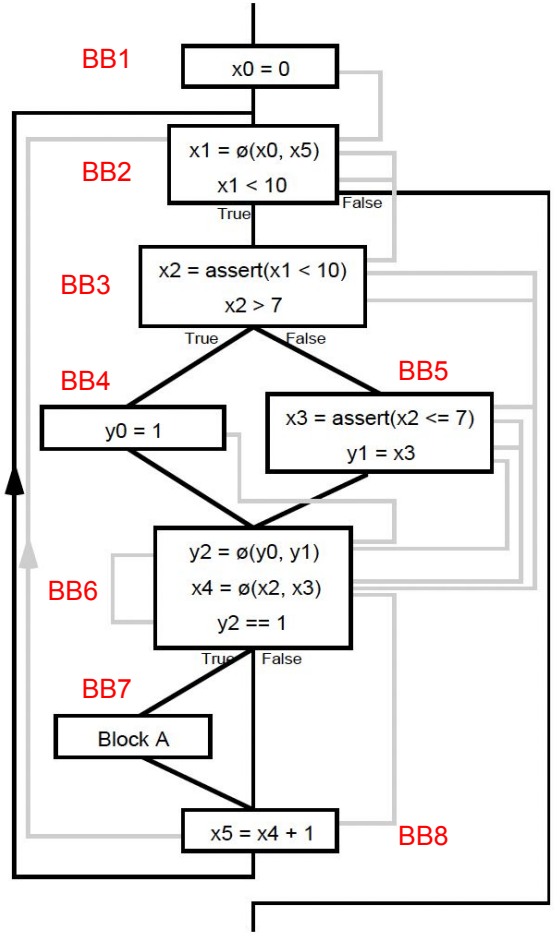


Execution Example

Value Range:

x0	{1[0:0:0]}
x1	{0[T]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

```
FlowWorkList = {BB1→BB2}
SSAWorkList = {x1=Φ(x0, x5)}

Item i = start→BB1
visited[BB1] != 1
Evaluate all expressions in BB1
    Evaluate fall-through branch
```

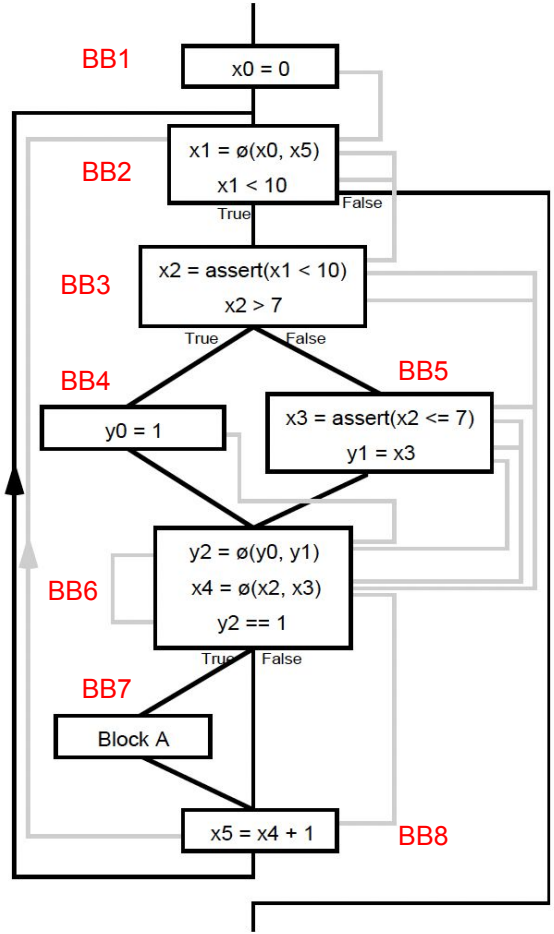


Execution Example

Value Range:

x0	{1[0:0:0]}
x1	{0[T]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

```
FlowWorkList = {BB1→BB2}
SSAWorkList = {x1=Φ(x0, x5)}
```

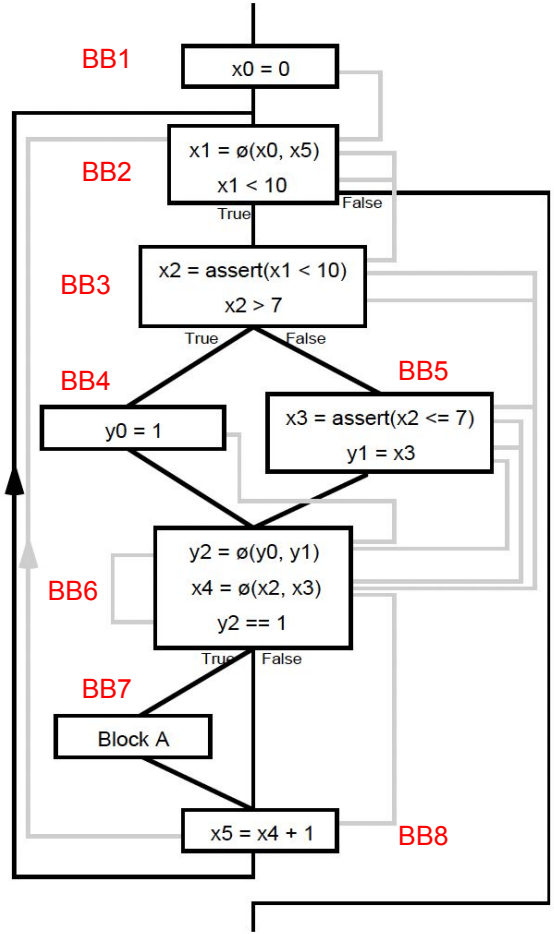


Execution Example

Value Range:

x0	{1[0:0:0]}
x1	{0[T]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

```
FlowWorkList = ∅
SSAWorkList = {x1=Φ(x0, x5)}

Item i = BB1→BB2
visited[BB2] != 1
Evaluate all expressions in BB2
```

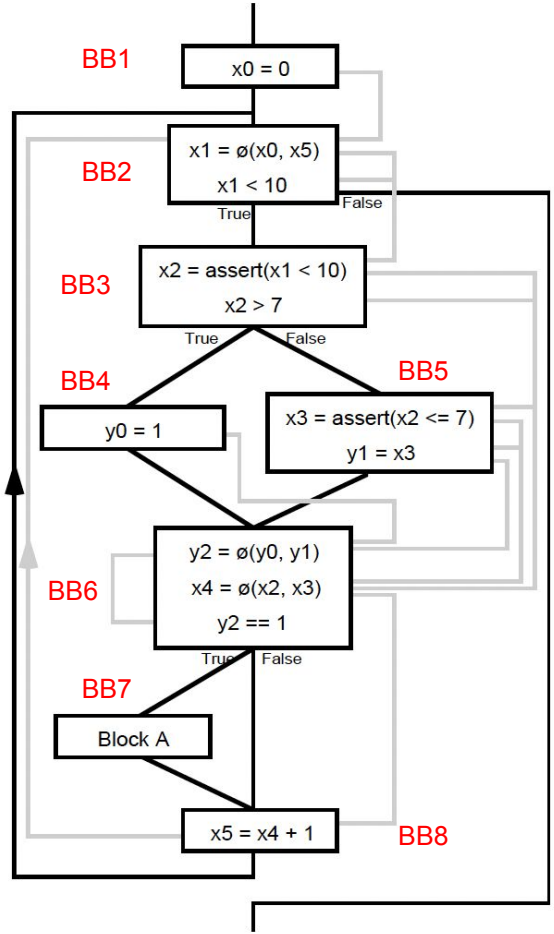


Execution Example

Value Range:

x0	{1[0:0:0]}
x1	{0[T]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

```
FlowWorkList = ∅
SSAWorkList = {x1=Φ(x0, x5)}

Item i = BB1→BB2
visited[BB2] != 1
Evaluate all expressions in BB2
    Evaluate x1=Φ(x0, x5)
    → loop carried
```



Execution Example

Value Range:

x0	{1[0:0:0]}
x1	**{1[0:10:1]}**
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

FlowWorkList = ∅

SSAWorkList = {x1=Φ(x0, x5), **x1<10, x2=assert(x1<10)**}

Item i = BB1→BB2

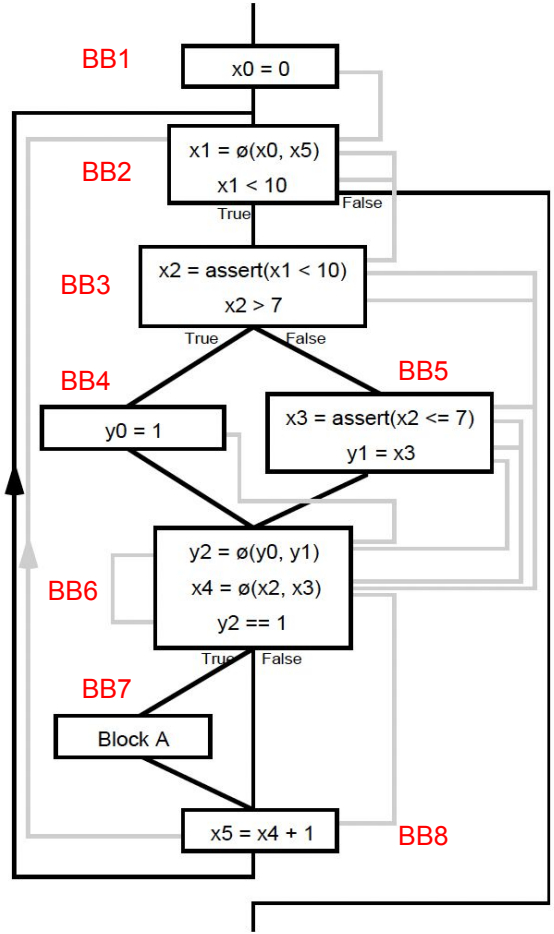
visited[BB2] != 1

Evaluate all expressions in BB2

Evaluate x1=Φ(x0, x5)

→ loop carried

Derive successful!



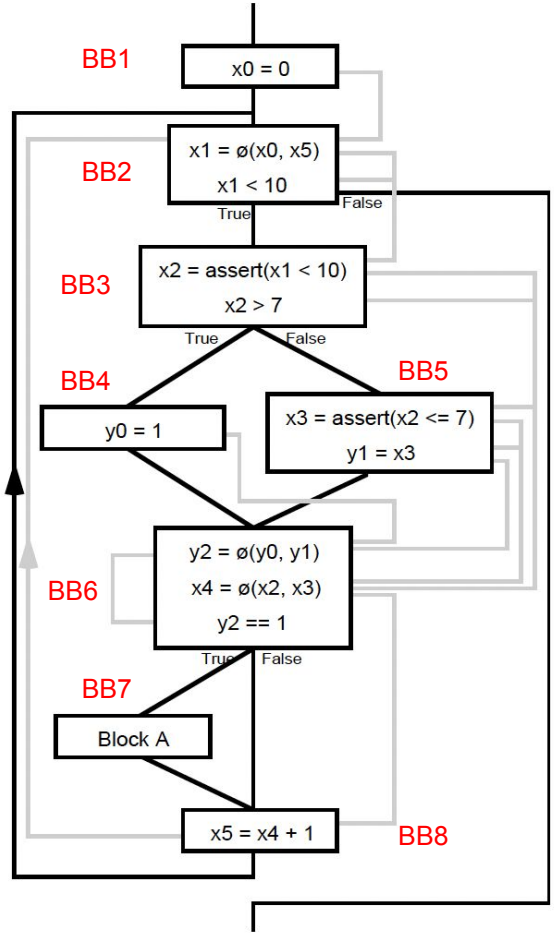
Execution Example

Value Range:

x0	{1[0:0:0]}
x1	{1[0:10:1]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

```
FlowWorkList = ∅
SSAWorkList = {x1=Φ(x0, x5),
x1<10, x2=assert(x1<10)}
```

```
Item i = BB1→BB2
visited[BB2] != 1
Evaluate all expressions in BB2
    Evaluate conditional branch
    x1 < 10?
```



Execution Example

Value Range:

x0	{1[0:0:0]}
x1	{1[0:10:1]}
x2	{0[T]}
x3	{0[T]}
x4	{0[T]}
x5	{0[T]}
y0	{0[T]}
y1	{0[T]}
y2	{0[T]}

Branch Prob:

x1<10 0.91

FlowWorkList = {**BB2→BB3, BB2→end**}

SSAWorkList = {x1=Φ(x0, x5), x1<10, x2=assert(x1<10)}

Item i = BB1→BB2

visited[BB2] != 1

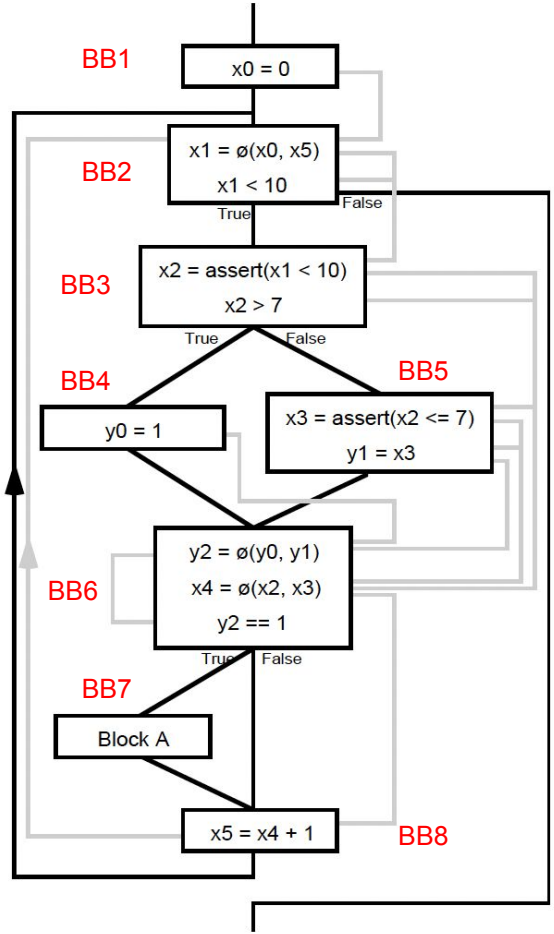
Evaluate all expressions in BB2

Evaluate conditional branch x1 < 10?

prob(BB2→BB3) = 0.91

prob(BB2→end) = 0.09

Continue following the algorithm...



Eventually we get to...

Value Range:

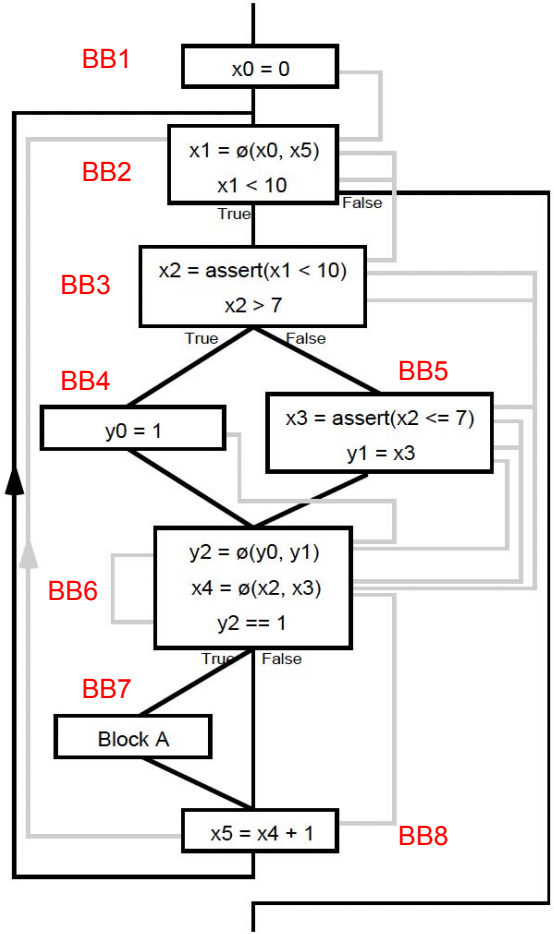
x0	{1[0:0:0]}
x1	{1[0:10:1]}
x2	{1[0:9:1]}
x3	{1[0:7:1]}
x4	{1[0:9:1]}
x5	{1[1:10:1]}
y0	{1[1:1:0]}
y1	{1[0:7:1]}
y2	{0.8[0:7:1], 0.2[1:1:0]}

Branch Prob:

x1<10	0.91
x2>7	0.20
y2==1	0.3

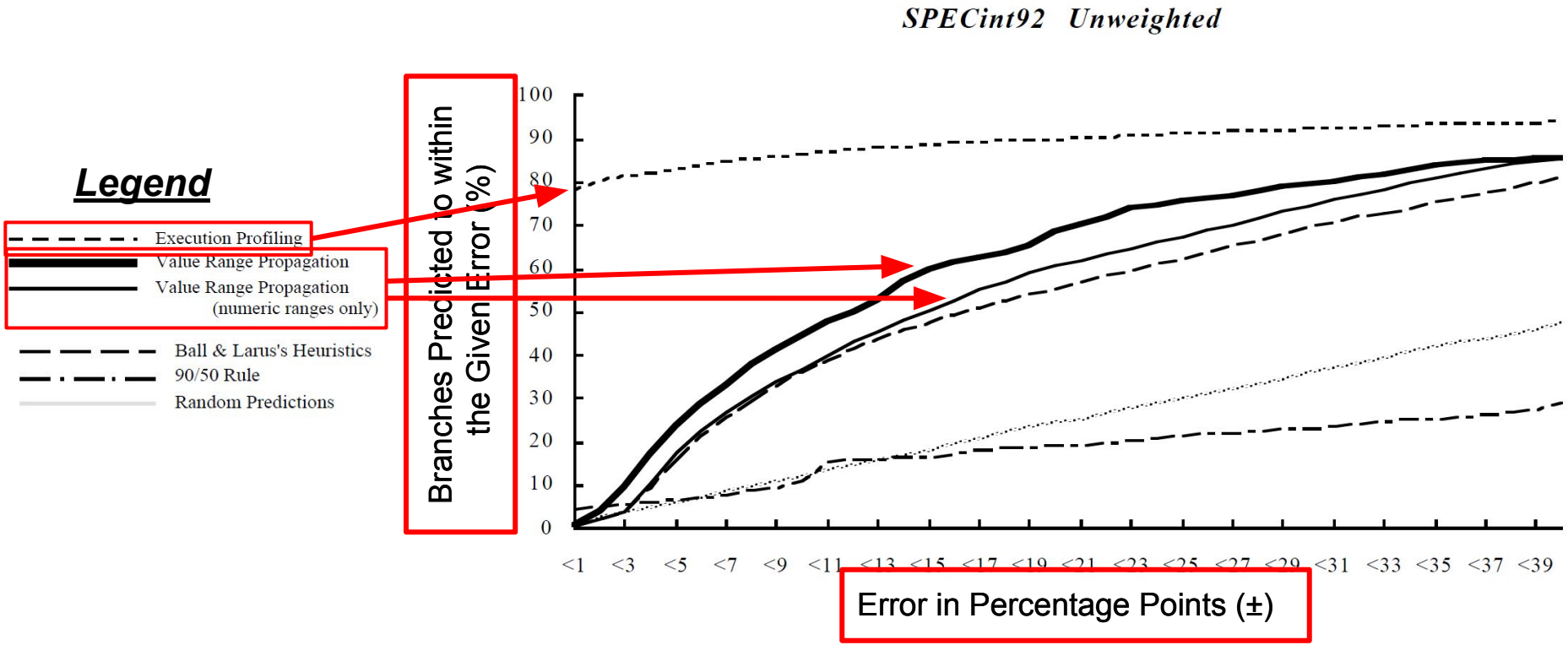
FlowWorkList = ∅

SSAWorkList = ∅



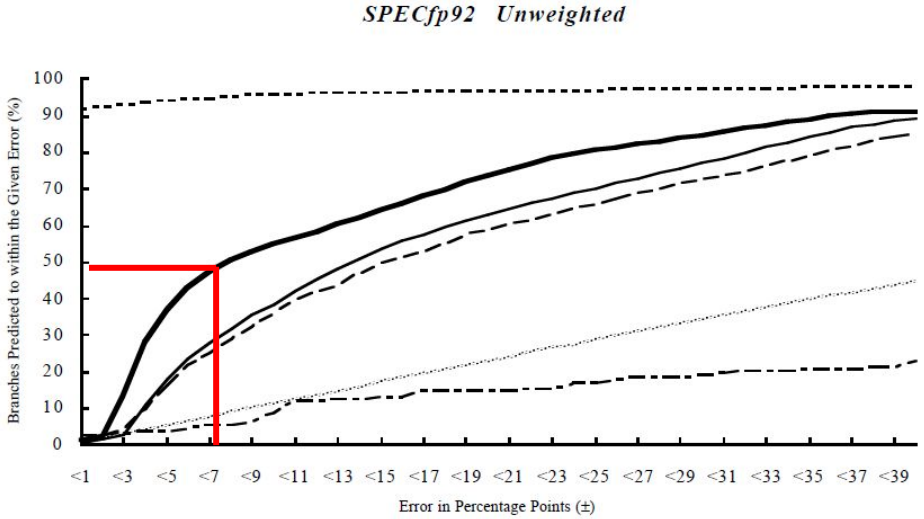
Results

Integer and pointer code benchmark

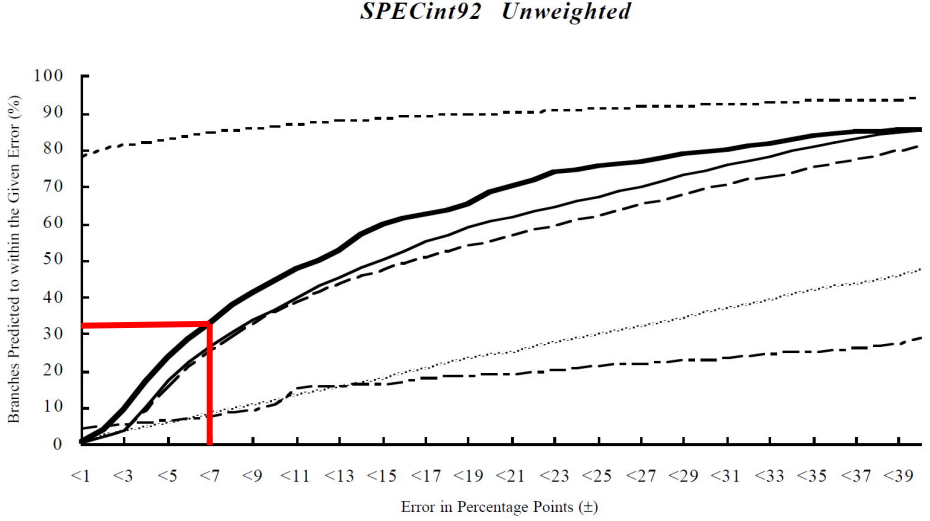


Results

Numeric code benchmark



Integer and pointer code benchmark



Strength vs. Weaknesses

Strengths

- Extension of constant propagation
- Branch prediction granularity

Weaknesses

- Algorithm extensibility
- Probability accuracy is significantly worse than profile data

Conclusion

- Static branch prediction algorithm
- Extends constant propagation
- Flexible and efficient value range representation

{ Probability[Lower bound : Upper bound : Stride], … }

- Performs better than previous static branch prediction heuristics