Sampling-based Runtime Verification

Borzoo Bonakdarpour, Samaneh Navabpour and Sebastian Fischmeister

Presented by: Bo Yang

Outline

- Background
- Problem description
- Solutions presented in the paper
- Experimental Results
- Conclusion and critiques

Background

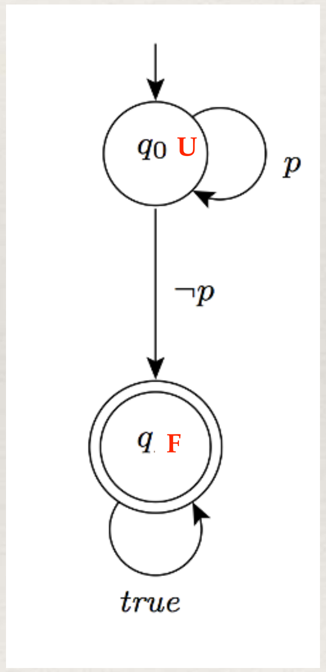
- Runtime Verification
  - differs from **theorem proving, model checking**
  - differs from **testing**
  - is to check whether a **run** of a system under scrutiny satisfies or violates a given correctness property
  - by synthesising the property to the **monitor**
  - The monitor observes and analyses the runs

Background

```
x = 0
y = 0
while (cond) {
    if (cond’) x = 0;
    x++;
    y++;
}
```

Property: Always (x <= 10)
safety property and monitorable

Let p denote x<=10



Synthesised Monitor

Possible runs:

```
x = 0 -> x = 1 -> x = 2 -> … > x = 10 -> x = 11 -> x = 0 -> …
  U        U        U     ..U..    U         F
x = 0 -> x = 1 -> x = 2 -> … > x = 10 -> x = 0 -> x = 1 -> …
  U        U        U     ..U..    U        U        U      …
```

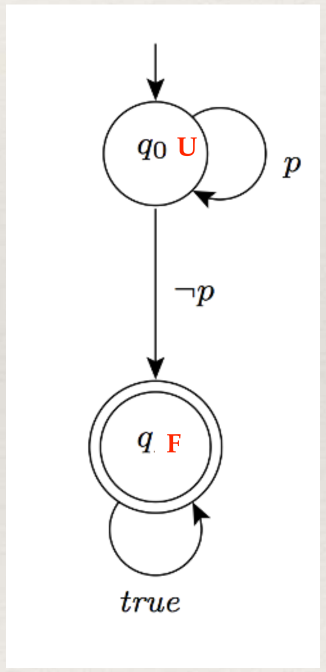
Background

```
x = 0
y = 0
while (cond) {
    if (cond') x = 0;
    x++;
    y++;
}
```

Property: Always (x <= 10)
safety property and monitorable

Let p denote x<=10

x is the variable of interest



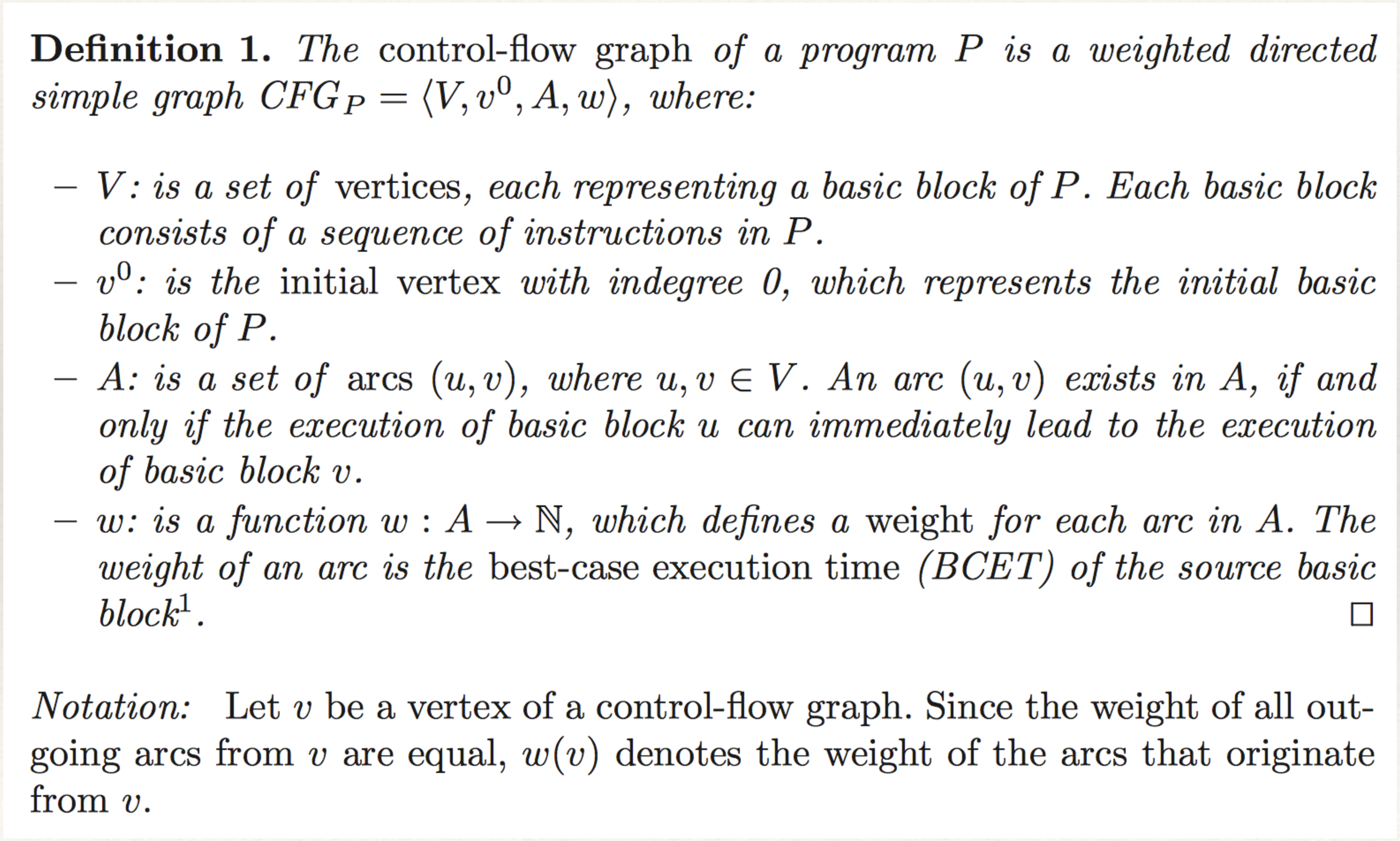
Synthesised Monitor

Possible runs:

```
x = 0 -> x = 1 -> x = 2 -> … > x = 10 -> x = 11 -> x = 0 -> …
  U        U        U     ..U..    U          F
x = 0 -> x = 1 -> x = 2 -> … > x = 10 -> x = 0 -> x = 1 -> …
  U        U        U     ..U..    U          U        U      …
```

Background

Control Flow Graph



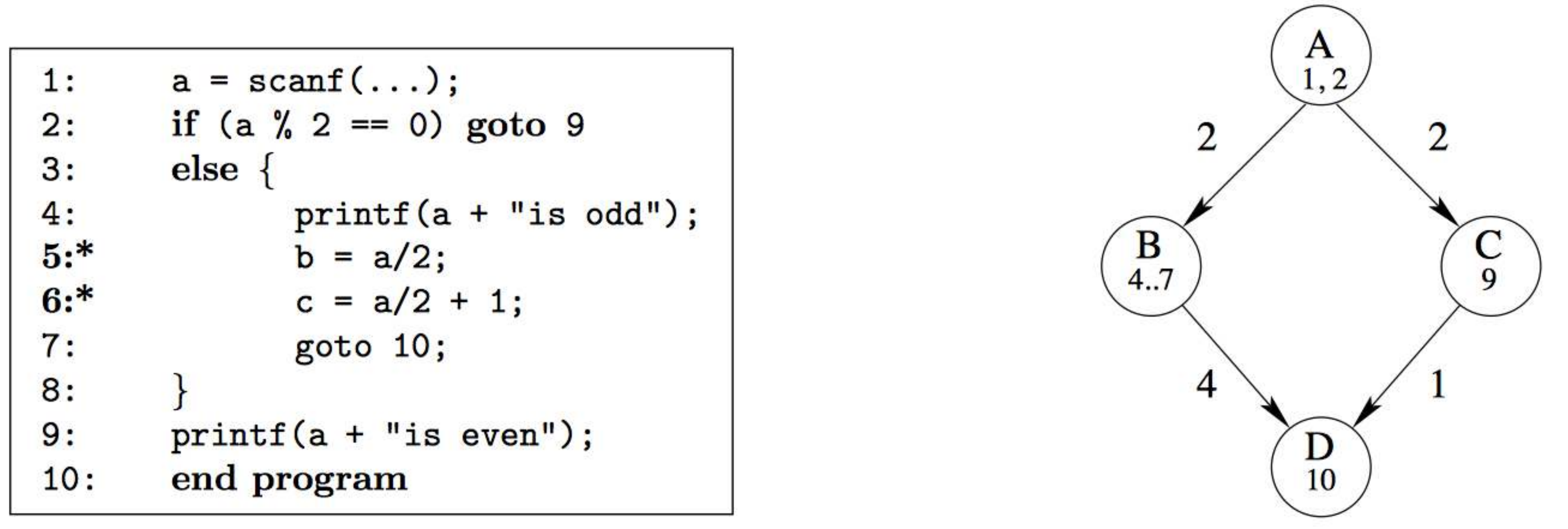
Definition 1. *The* control-flow graph *of a program P is a weighted directed simple graph $CFG_P = \langle V, v^0, A, w \rangle$, where:*

- *V: is a set of* vertices, *each representing a basic block of P. Each basic block consists of a sequence of instructions in P.*
- *v^0: is the* initial vertex *with indegree 0, which represents the initial basic block of P.*
- *A: is a set of* arcs *(u, v), where $u, v \in V$. An arc (u, v) exists in A, if and only if the execution of basic block u can immediately lead to the execution of basic block v.*
- *w: is a function $w : A \rightarrow \mathbb{N}$, which defines a* weight *for each arc in A. The weight of an arc is the* best-case execution time *(BCET) of the source basic block[1].* $\square$

Notation: Let v be a vertex of a control-flow graph. Since the weight of all outgoing arcs from v are equal, $w(v)$ denotes the weight of the arcs that originate from v.

Background

Control Flow Graph: an example



```
1:     a = scanf(...);
2:     if (a % 2 == 0) goto 9
3:     else {
4:             printf(a + "is odd");
5:*            b = a/2;
6:*            c = a/2 + 1;
7:             goto 10;
8:     }
9:     printf(a + "is even");
10:    end program
```

(a) A simple C program

(b) Control-flow graph

Fig. 1. A C program and its control-flow graph.

Problem Description

- When to invoke the monitor?
  - Every time value of the variables of interest change or every fixed interval (‘event triggered’ or ‘time triggered’)
  - Event triggered vs. Time triggered
    - ‘Event triggered’ introduces unpredictable overhead, bursts of interruptions, possible probe effect
    - ‘Time triggered’ results in predictable overhead, likely longer overall executing time
  - In CPS application domain, predictability is highly desired. ‘Time triggered’ monitor is a periodic task. Easy to schedule!

Problem Description

- How to decide the sampling period
    - The naive way is to choose the minimum time that the variables of interest change their values
    - Increase the sampling period by memorising the values change history
        - Need auxiliary memory. Trade off between auxiliary memory and sampling period. Find the optimality!
        - Increasing overhead

Solutions

- First, construct the critical CFG (Control Flow Graph) from CFG

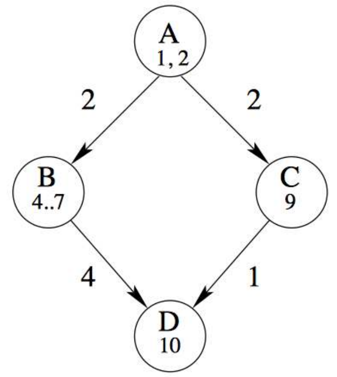
  Step1: Each critical instruction is in a basic block that contains no other instructions.

  Step2: Uncritical vertices play no role in determining the sampling period. Collapse uncritical vertices.

Assume each instruction takes 1 time unit to finish execution

```
1:     a = scanf(...);
2:     if (a % 2 == 0) goto 9
3:     else {
4:           printf(a + "is odd");
5:*          b = a/2;
6:*          c = a/2 + 1;
7:           goto 10;
8:     }
9:     printf(a + "is even");
10:    end program
```

(a) A simple C program



(b) Control-flow graph

Fig. 1. A C program and its control-flow graph.

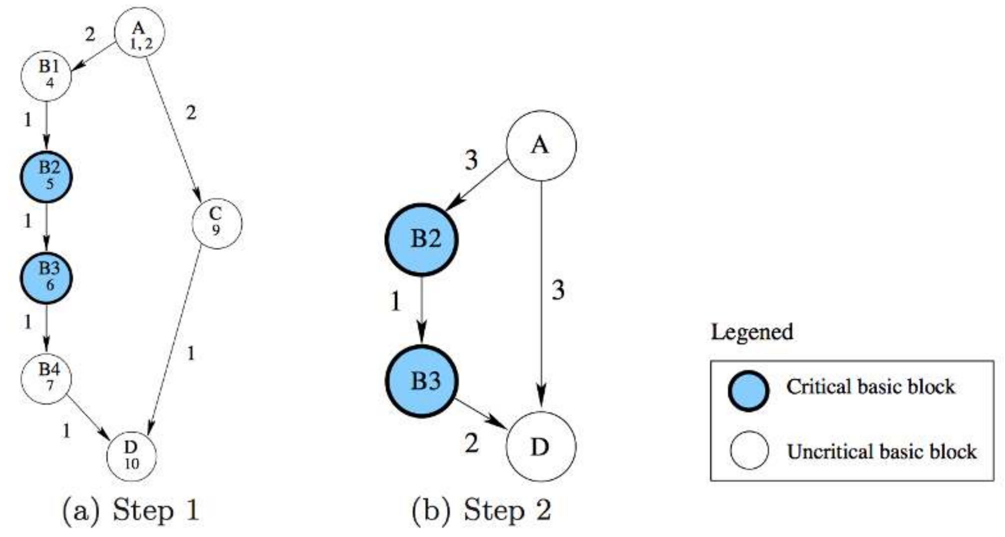


Fig. 2. Obtaining a critical CFG and calculating the sampling period.

Solution 1—Applying MSP

- Minimum Sample Period (MSP) = min $\{w(v1,v2) \mid (v1, v2) \in A \wedge v1$ is a critical vertex$\}$
- For our example, MSP = 1
- Applying MSP, no property violations can be overlooked.
- But the overhead is increased, overall execution time is increased a lot.

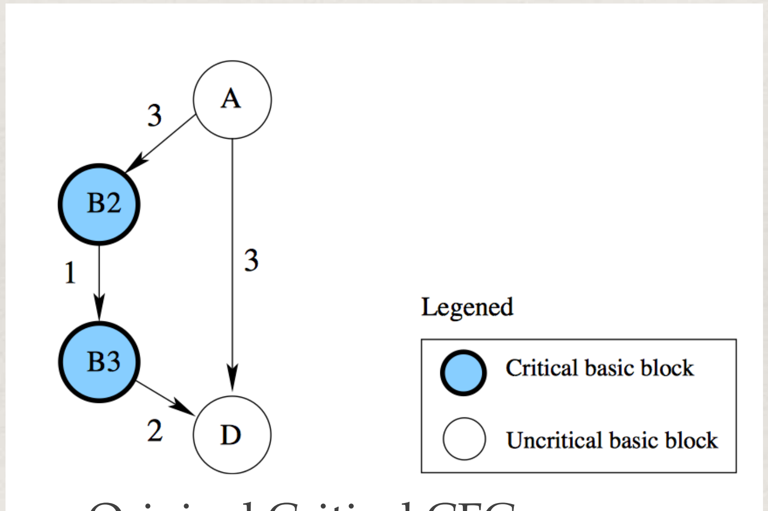
Solution2—Increasing SP

- How to build a history of critical state changes between two samples?
    - Add instruments to related blocks, like $a' \longleftarrow a$ (a is a variable) meaning saving a to memory location a'.
    - Instrumenting is a technique used in software debugging, monitoring without affecting the program's correctness.

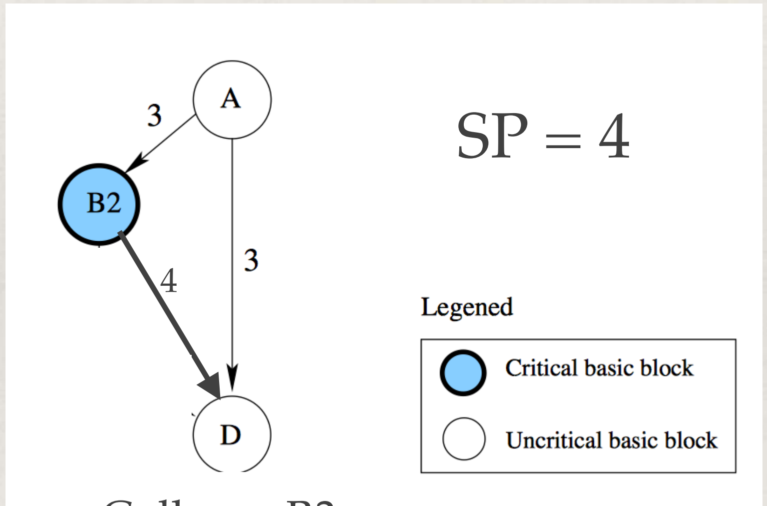
Solution2—Increasing SP

- Collapse critical blocks.

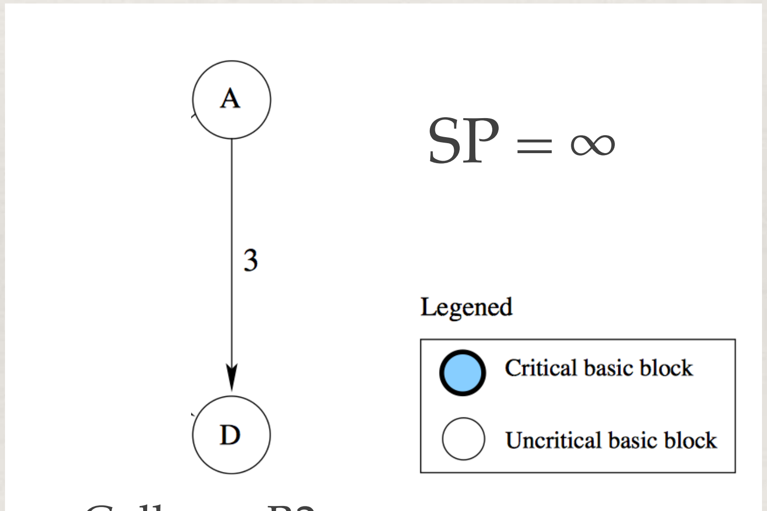
$\langle B2 \rangle = \langle B2 \rangle . \langle B3 \rangle .\ a' \longleftarrow a$



Original Critical CFG



Collapse B3



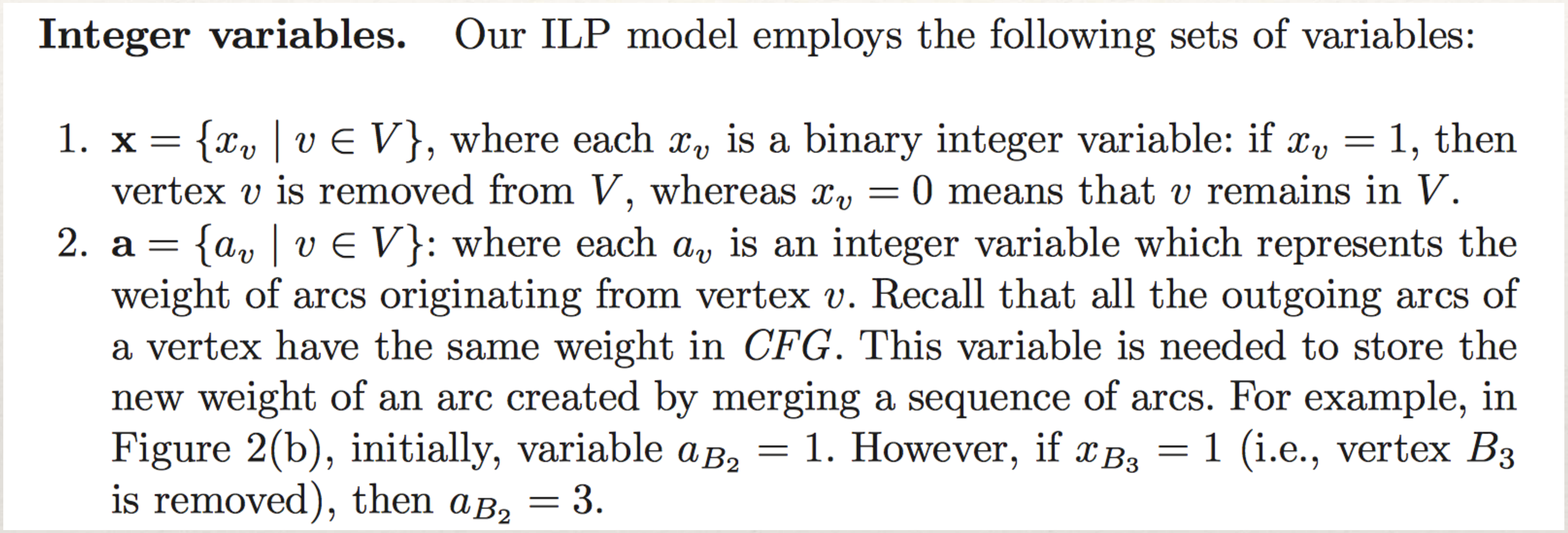
Collapse B2

Solution2—Increasing SP

- Optimization problem: minimizing auxiliary memory and maximising sampling period
  - Reduced to a decision problem: Does there exist a set $U \subseteq V$, such that after applying collapsing all $u \in U$, we obtain a critical control-flow graph $CFG' = \langle V', v^0, A', w' \rangle$, where $|U| \le Y$ and for all arcs $(u,v) \in A'$, $w'(u,v) \ge X$? (NP-Complete)
  - Mapped to ILP (integer linear programming). Minimize the number of instrumentations while satisfying SP requirement.
  - ILP solvers solve such problems: minimize some term while subjected to some constraints. (e.g. $2*x+y >= 4 \wedge x+2*y>=2$, minimize $x+y$)

Solution2—Increasing SP

The corresponding ILP model

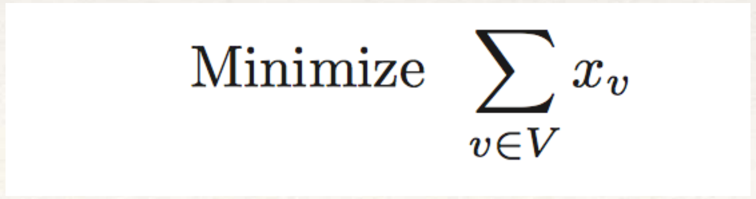


Integer variables. Our ILP model employs the following sets of variables:

1. $\mathbf{x} = \{x_v \mid v \in V\}$, where each x_v is a binary integer variable: if $x_v = 1$, then vertex v is removed from V, whereas $x_v = 0$ means that v remains in V.
2. $\mathbf{a} = \{a_v \mid v \in V\}$: where each a_v is an integer variable which represents the weight of arcs originating from vertex v. Recall that all the outgoing arcs of a vertex have the same weight in *CFG*. This variable is needed to store the new weight of an arc created by merging a sequence of arcs. For example, in Figure 2(b), initially, variable $a_{B_2} = 1$. However, if $x_{B_3} = 1$ (i.e., vertex B_3 is removed), then $a_{B_2} = 3$.

……..

The goal is to



$$\text{Minimize} \sum_{v \in V} x_v$$

Experimental Results

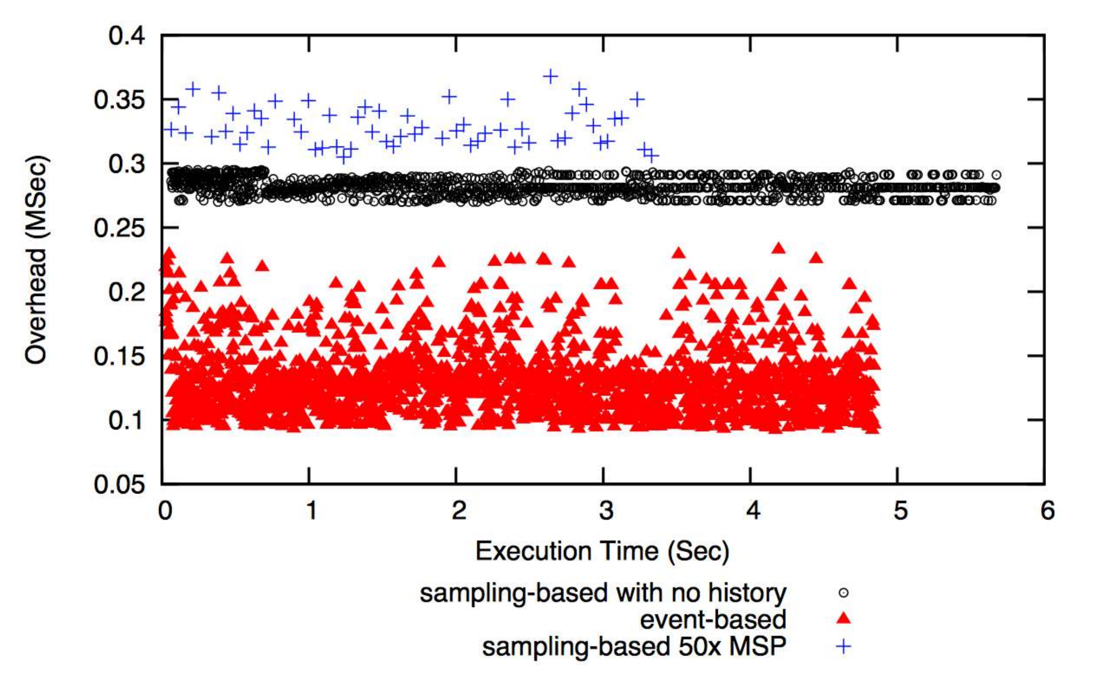


Fig. 4. Experimental results for Dijktra ($50 * MSP$ sampling period).

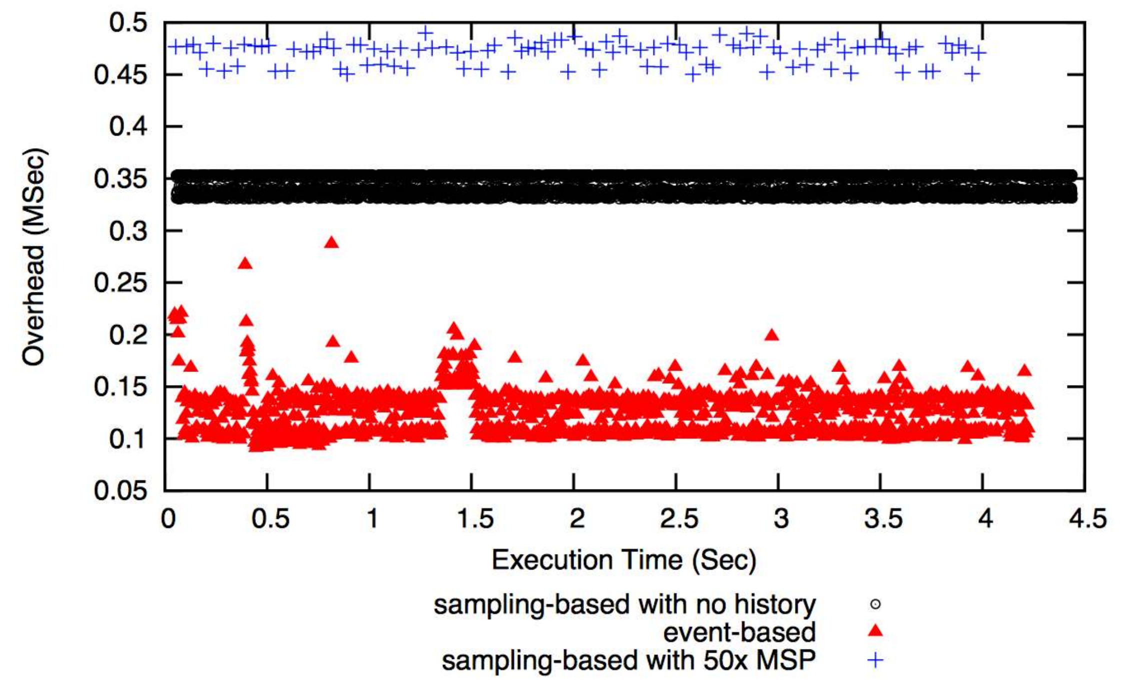


Fig. 5. Experimental results for Blowfish ($50 * MSP$ sampling period).

For event-triggered, overhead is subjected to bursts while the overhead for sampling-based monitors remains consistent and bounded.

Increased overhead for sampling-based monitors. May or may not increase the overall execution time.

Adding history causes variability in data extraction overhead, but still better than event-triggered.

Experimental Results

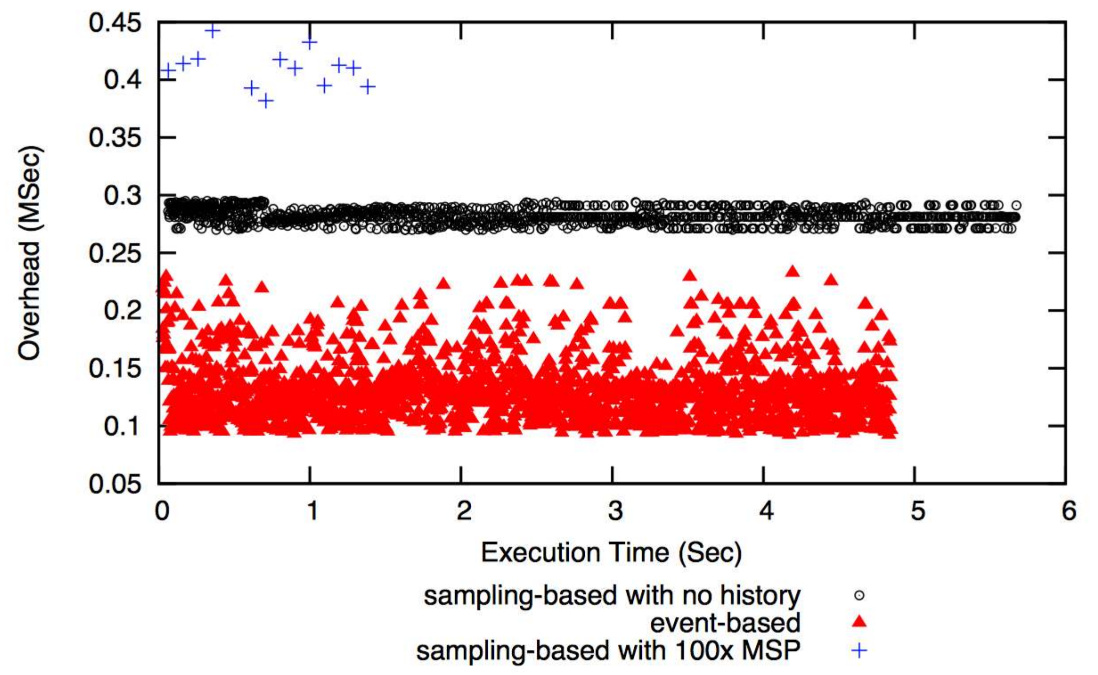


Fig. 6. Experimental results for Dijktra (100 $*$ *MSP* sampling period).

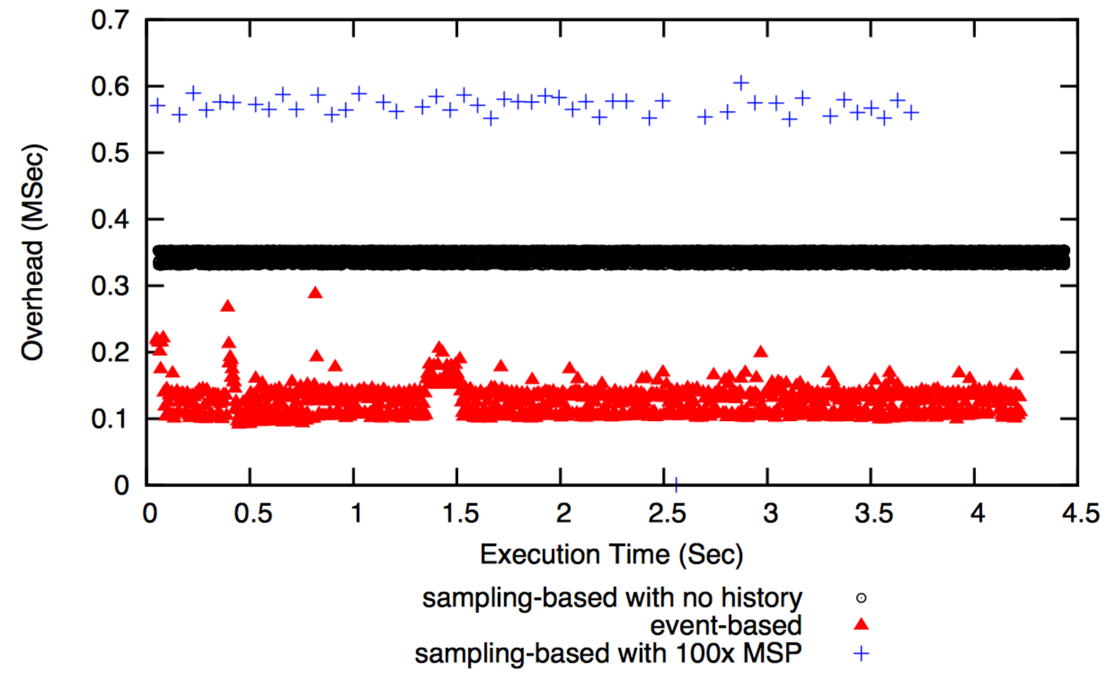


Fig. 7. Experimental results for Blowfish (100 $*$ *MSP* sampling period).

Due to the fact that more and larger variables are stored in the history between two samples for Blowfish, the reduction in execution time of Blowfish is less than Dijkstra.

The number of variables stored in the history from one sample to another does not significantly change in Blowfish, thus the overhead variability in Blowfish is smaller than that in Dijktra.

Experimental Results

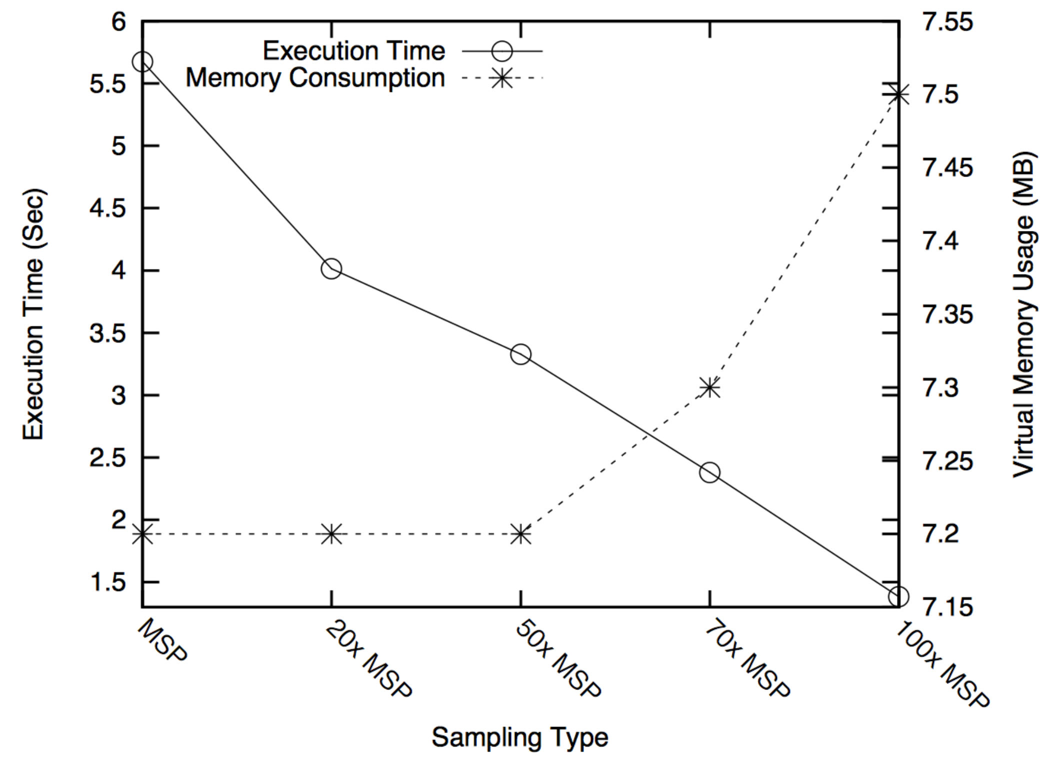


Fig. 8. Memory usage vs. execution time (Dijkstra).

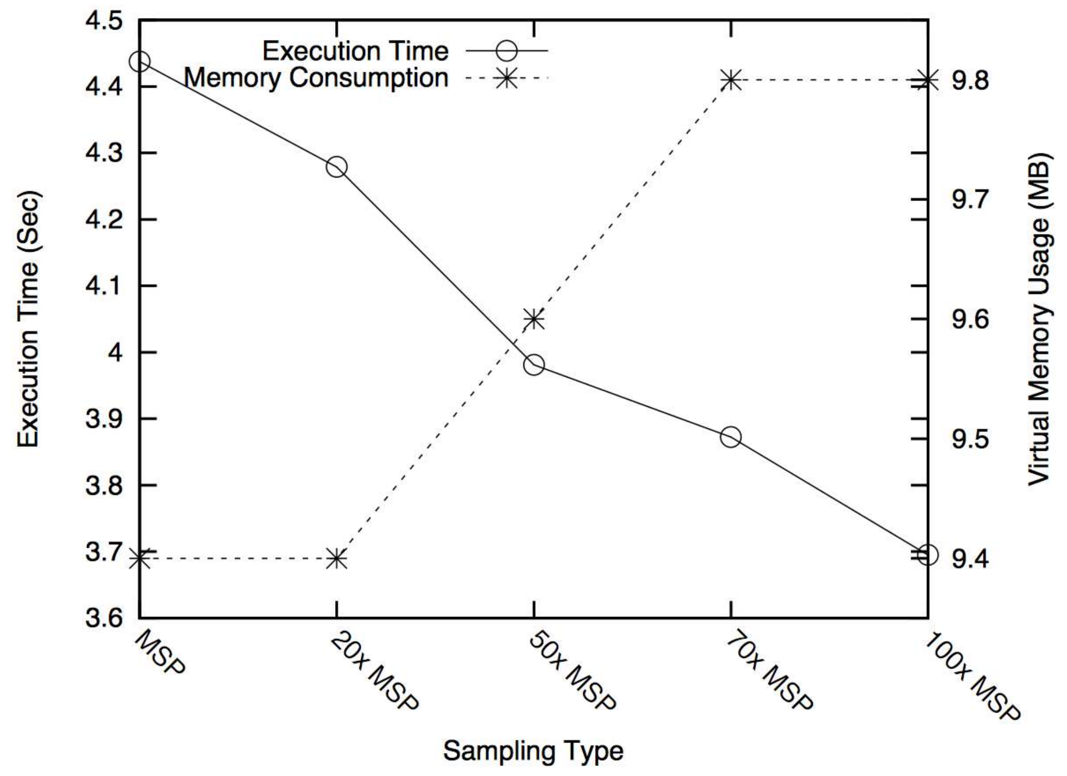


Fig. 9. Memory usage vs. execution time Blowfish.

To increase the sampling period, the system requires negligible extra memory.

Conclusions and Critiques

- Sampling-based approach for runtime verification is investigated, and defined in formal terms.
- Using minimum auxiliary memory to maximize the sampling period is NP-complete.
- The problem is encoded in ILP as a practical solution.
- Sampling-based monitoring provides a predictive overhead.

Critiques:

1. *Some programs may not be able to transformed to CFG.*
2. *MSP is pessimistic ?*
3. *Adding interments may result in missing deadline, thus constrained ability to increase SP.*
4. *Detection delay problem.*
5. *A heuristic algorithm is desired if the problem becomes too complicate for ILP solvers.*