Information Extraction from Real-time Applications at Run Time

Sebastian Fischmeister
University of Waterloo

esg.uwaterloo.ca

UNIVERSITY OF WATERLOO

Outline

- Setting the stage
- Motivate the need for information extraction
- Time-aware instrumentation
- Time-triggered runtime verification
- Conclusions

SETTING THE STAGE: REAL-TIME SAFETY-CRITICAL EMBEDDED SOFTWARE

Embedded Systems Everywhere

Embedded Software Everywhere

System	Lines of Code
Darlington Shutdown System	40 000
Mars Science Laboratory	4 000 000
Boeing 787	6 500 000
Current luxury car	100 000 000

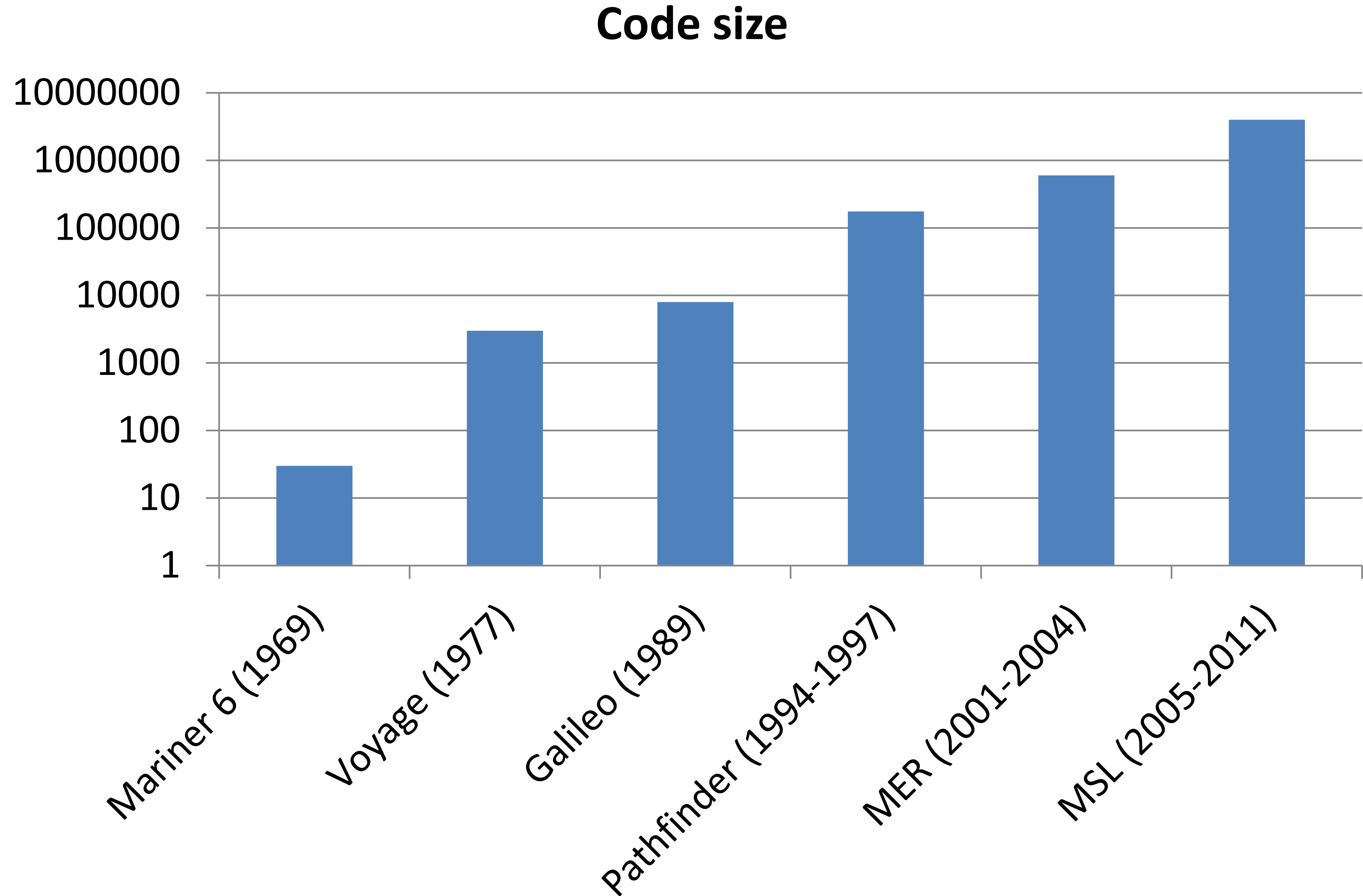
Safety-critical Real-time Systems

Physics doesn't wait for you.
The right value too late still causes errors.

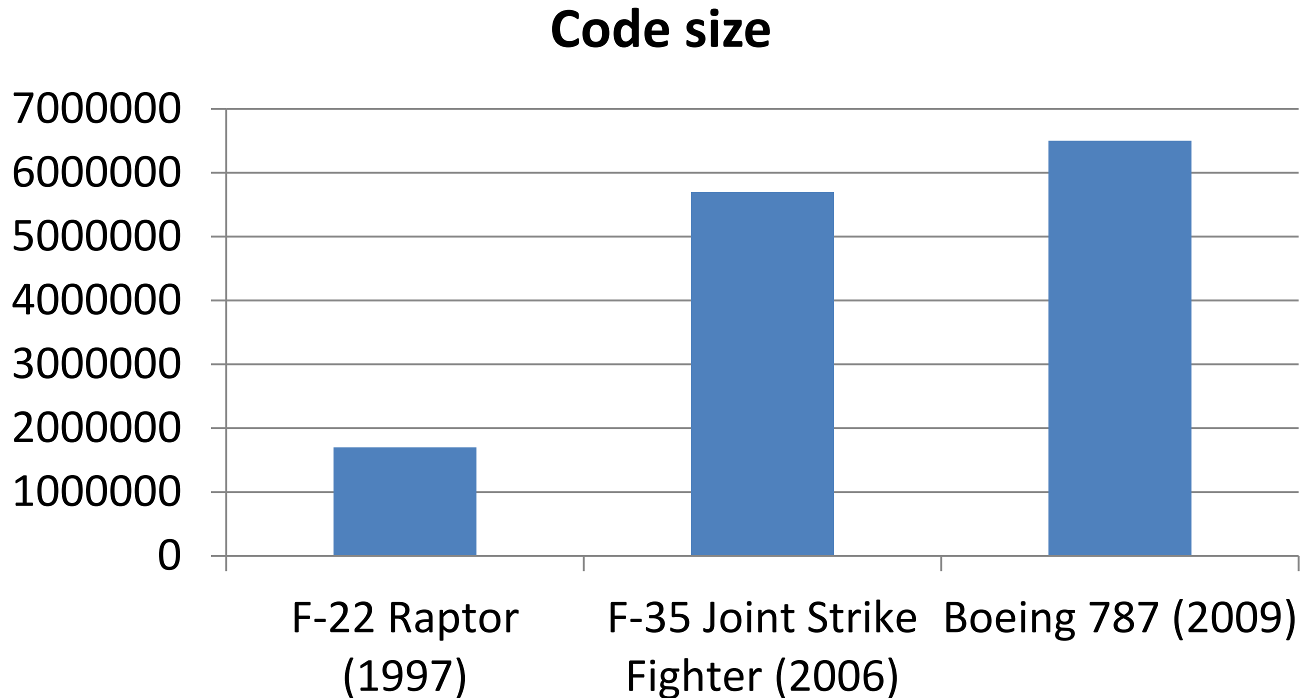
THE NEED FOR INFORMATION EXTRACTION

- Software is getting big
- We can’t comprehend it
- Bugs are real

Software is Getting Big



Software is Getting Big



- GM car in 1981: 50 000 LOC
- GM car in 2011: 100 000 000 LOC
- Next generation car: 300 000 000 LOC

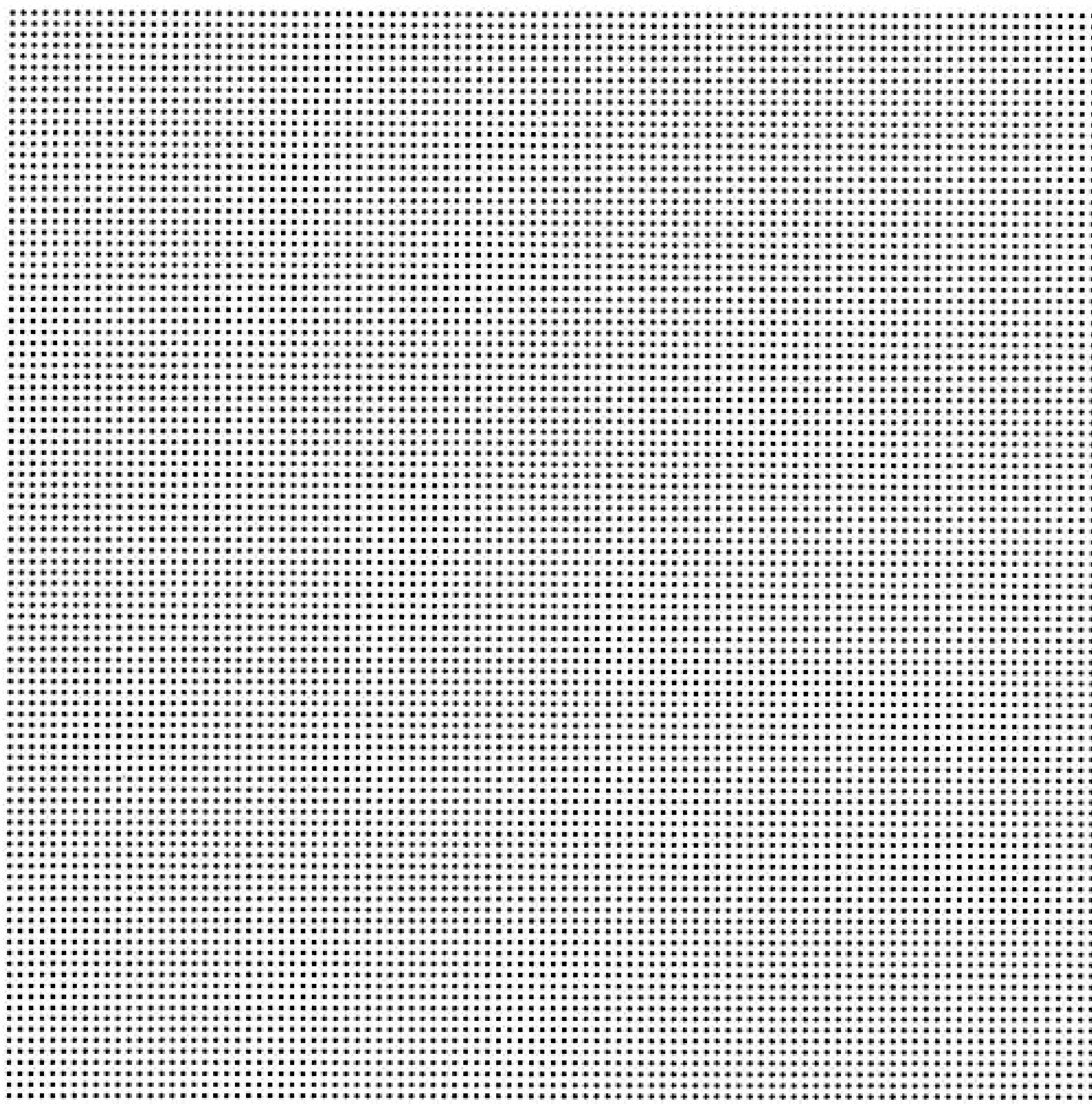
We Cannot Comprehend Software

- **Software is where the innovation is happening!**
  Features sell, apps everywhere

- **Software size and complexity is the challenge!**

Illustrating one root cause:
Bridge from Tokyo
to Vancouver

10 000 dots



100K?

10M?

~100M Pixels

Courtesy of Bob Mallard.

Bugs are Real

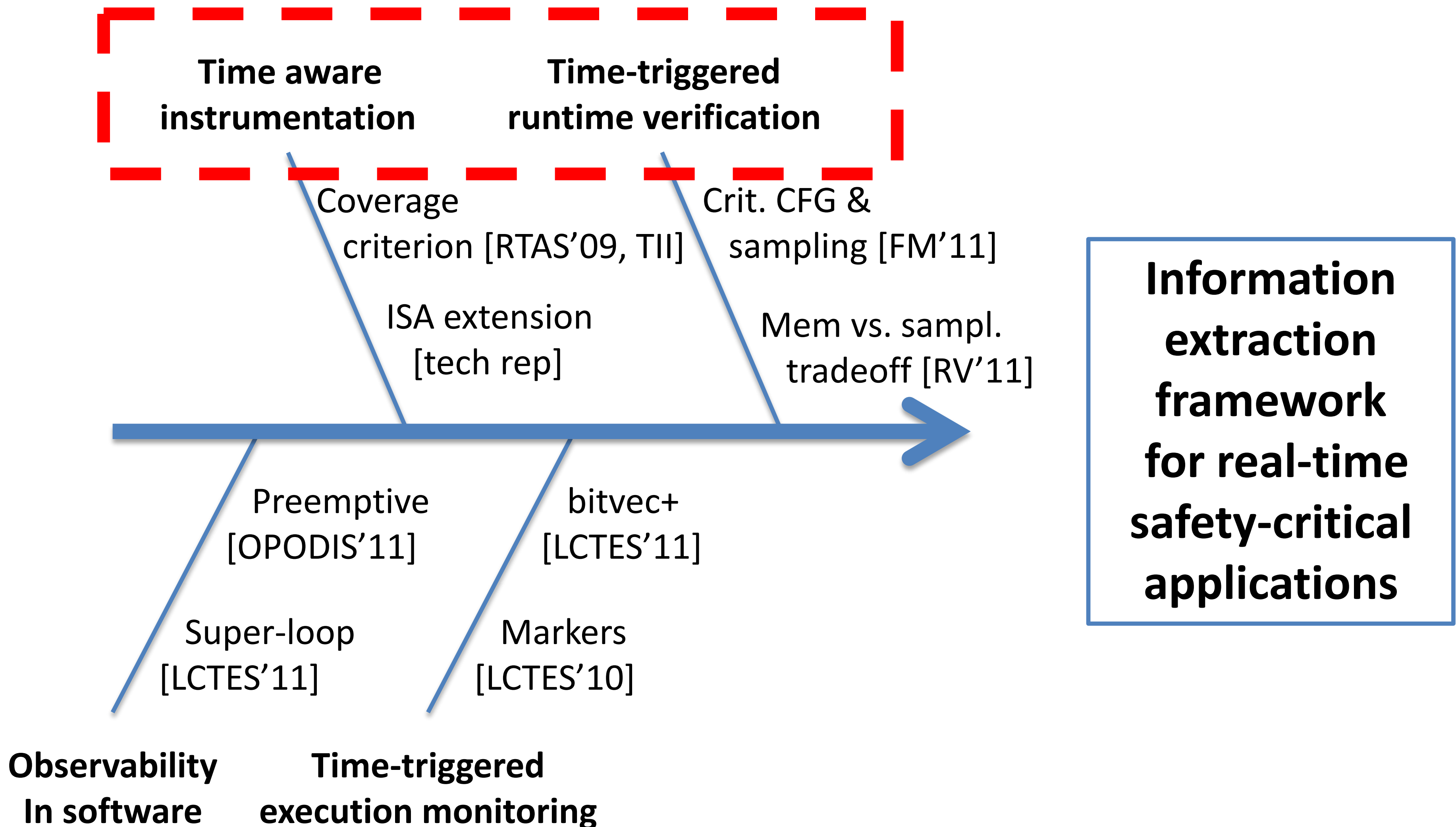
- 80% of the developer time is debugging
- 30-50% of the total cost is integration testing and debugging

Source	*Language*	*Failure per KLOC*	*Formal methods used?*
Siemens operating system	Assembly	6-15	No
NAG scientific libraries	Fortran	3.00	No
CDIS air-traffic-control support	C	0.81	Yes
Lloyd's language parser	C	1.40	Yes
IBM Cleanroom development	**Various**	**3.40**	**Partly**
IBM normal development	Various	30.0	No
Satellite planning study	Fortran	6-16	No
Unisys communication software	Ada	2-9	No

Information Extraction

- Information extraction helps the developer understand the program's behavior at run time:
  - Testing, debugging, tuning, monitoring, validating, certifying
- **Goals:** Easy, low cost, readily available, deployable, and **shouldn't break anything.**
- **Problem:** Existing approaches mostly consider logical correctness only, but **what about other properties? (e.g., timing)**

Vision & Path



TIME-AWARE INSTRUMENTATION

Fischmeister, S., and P. Lam, "Time-Aware Instrumentation of Embedded Software", IEEE Transactions on Industrial Informatics, vol. P, issue 99, pp. 1551–3203, August, 2010

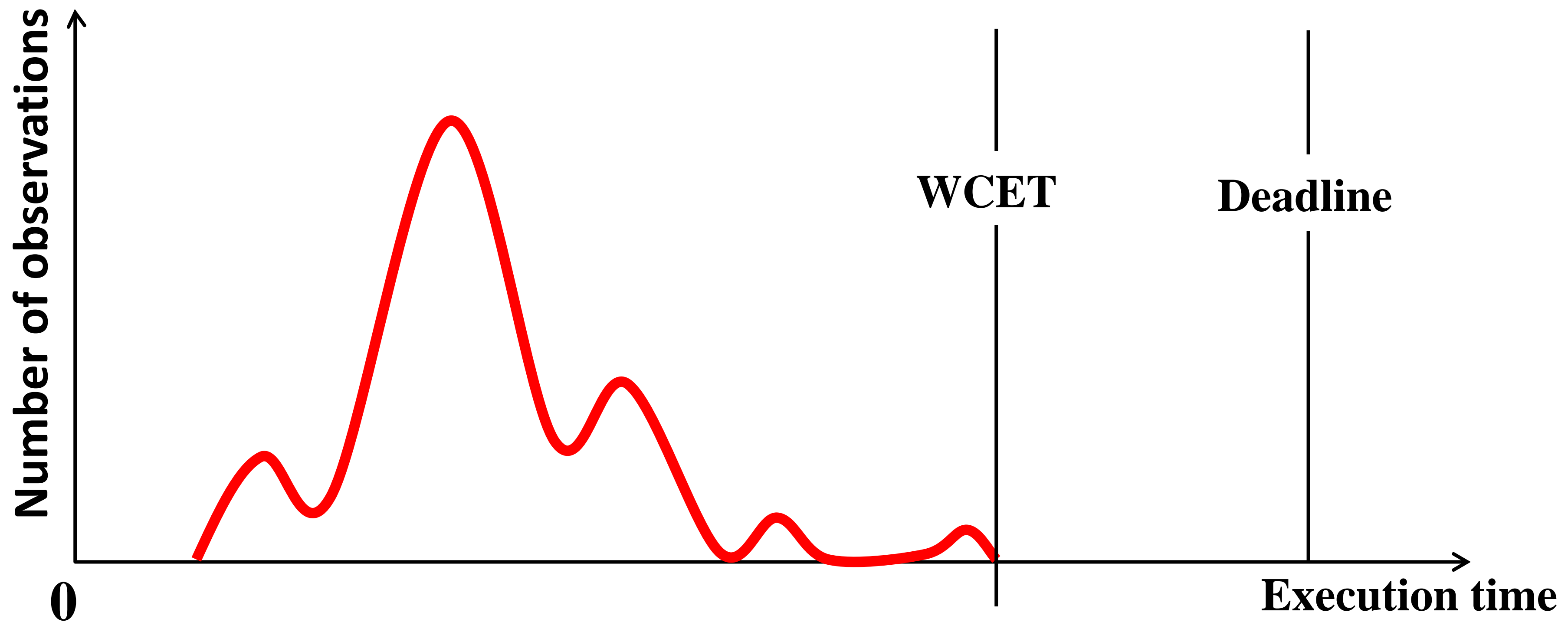
Fischmeister, S., and P. Lam, "On Time-Aware Instrumentation of Programs", Proc. of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), San Francisco, USA, pp. 305--314, 2009.

Problem

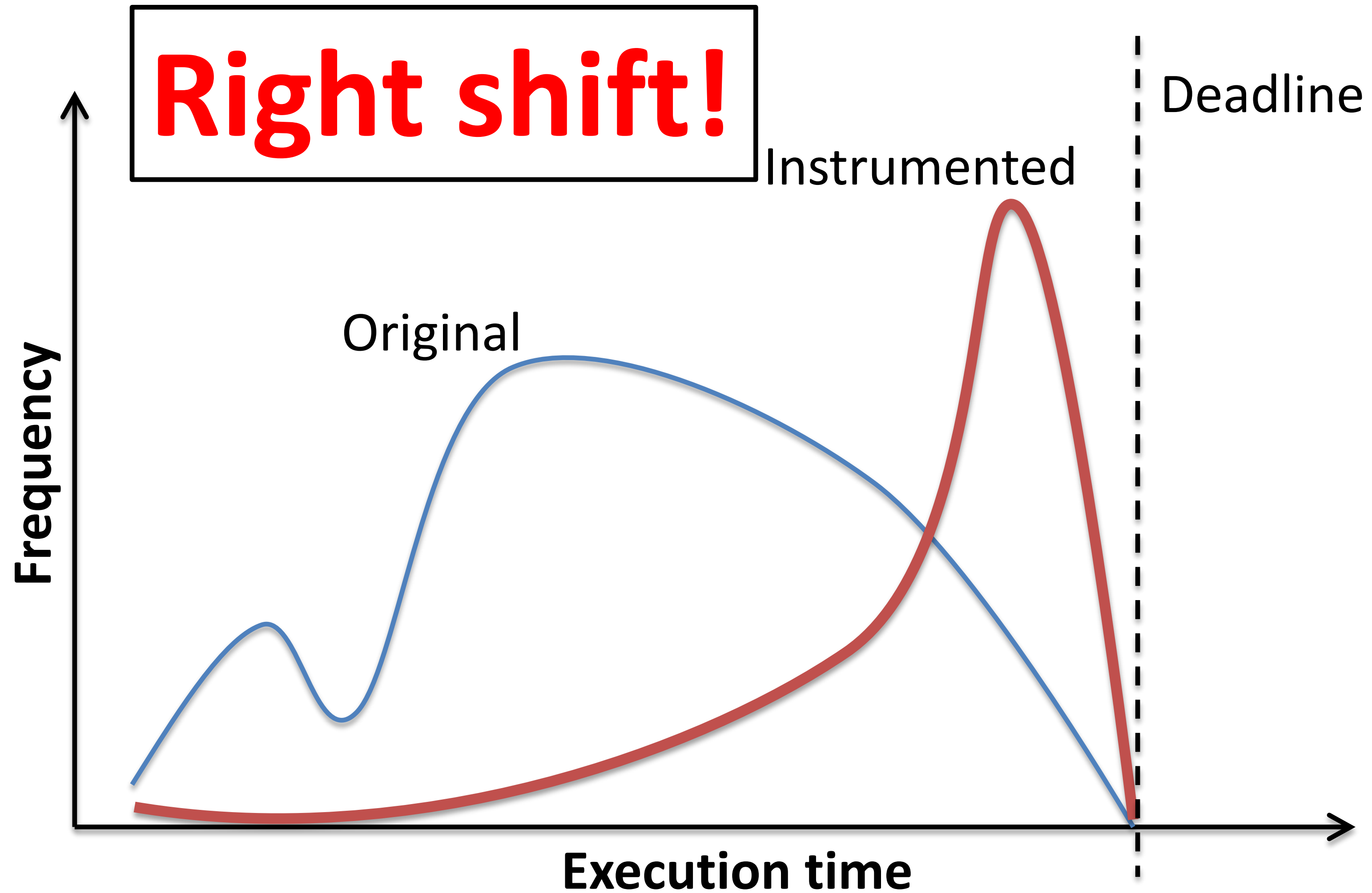
Current instrumentation methods preserve only logical correctness.

Can we capture runtime execution behavior (=variable assignments) with no or little timing interference?

Execution Time Profile



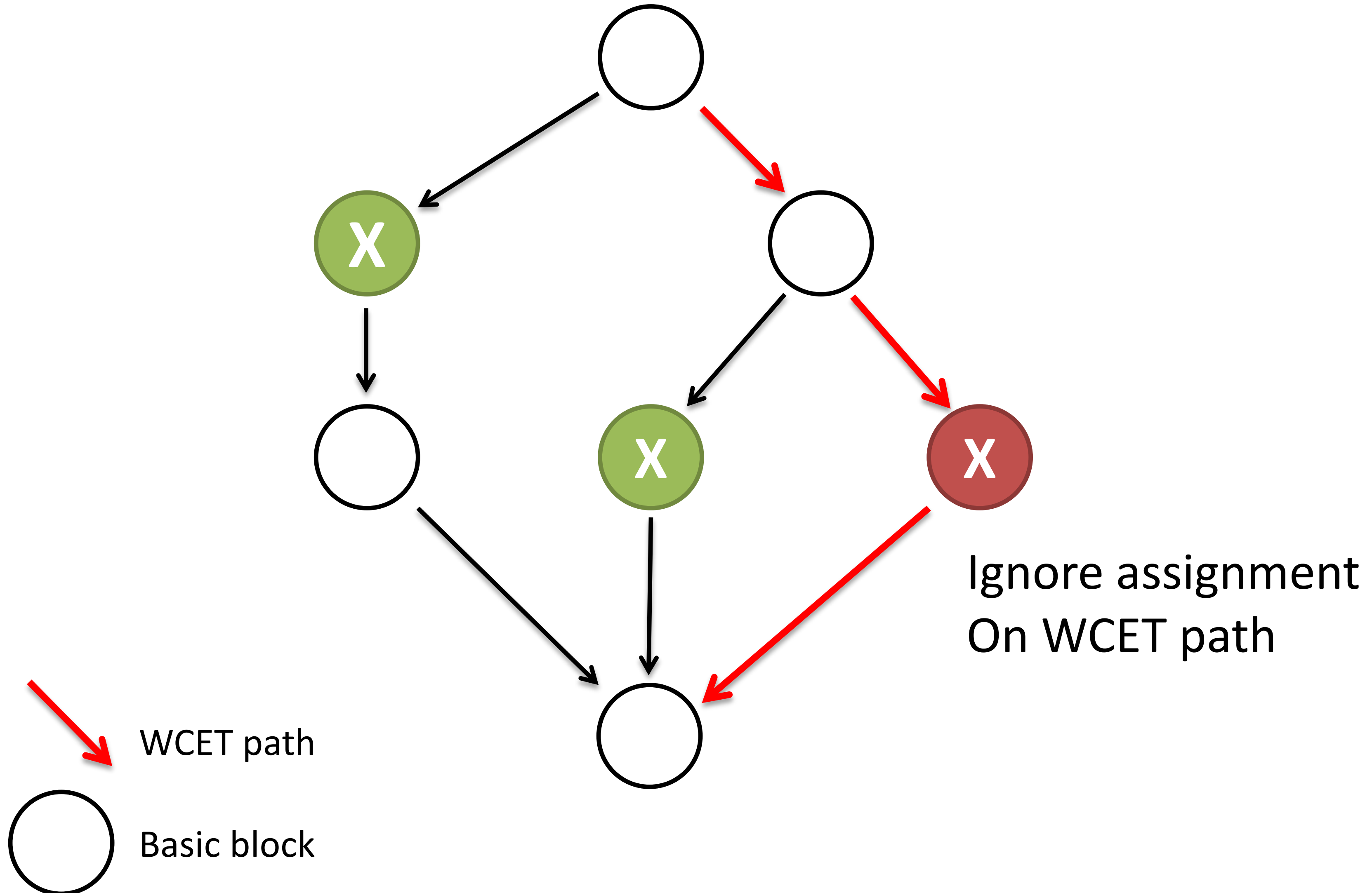
Idea in a Nutshell



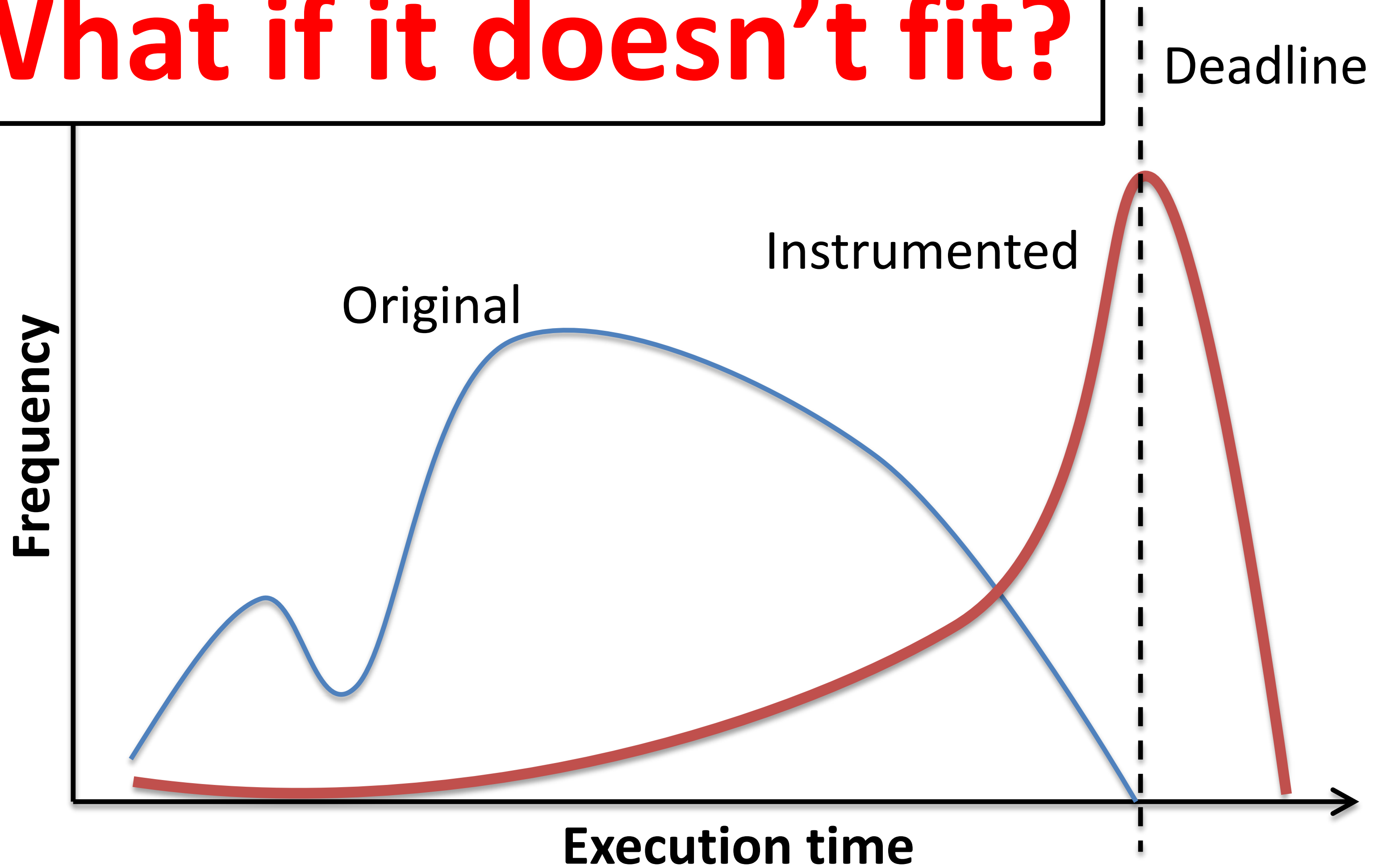
Challenges

- Can we actually create this right shift?
- What will we do, if there is insufficient slack?
- What does the optimal solution look like?

Capturing on non-WCET Paths



Idea in a Nutshell



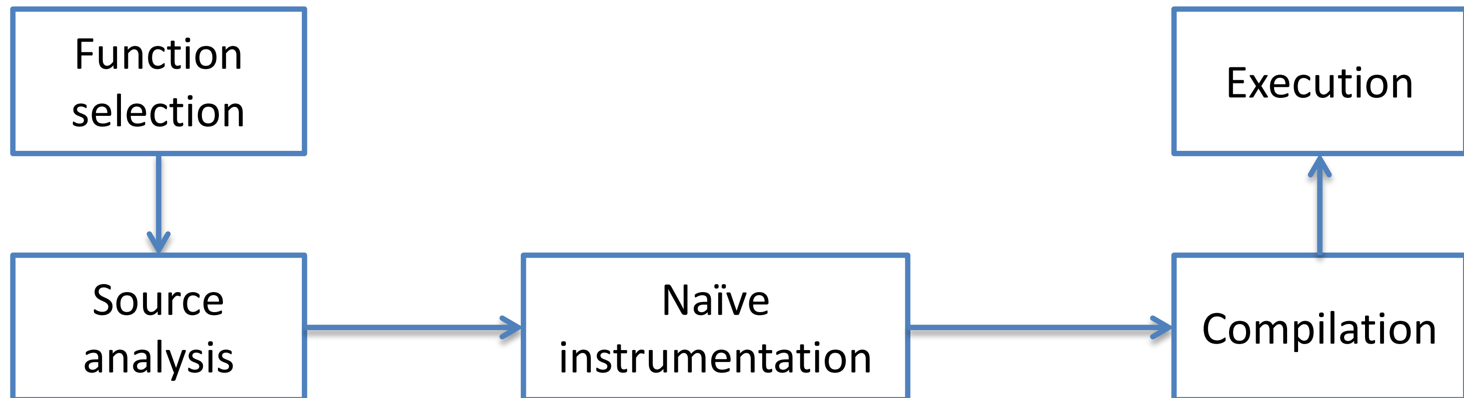
➔Concept of coverage

Coverage

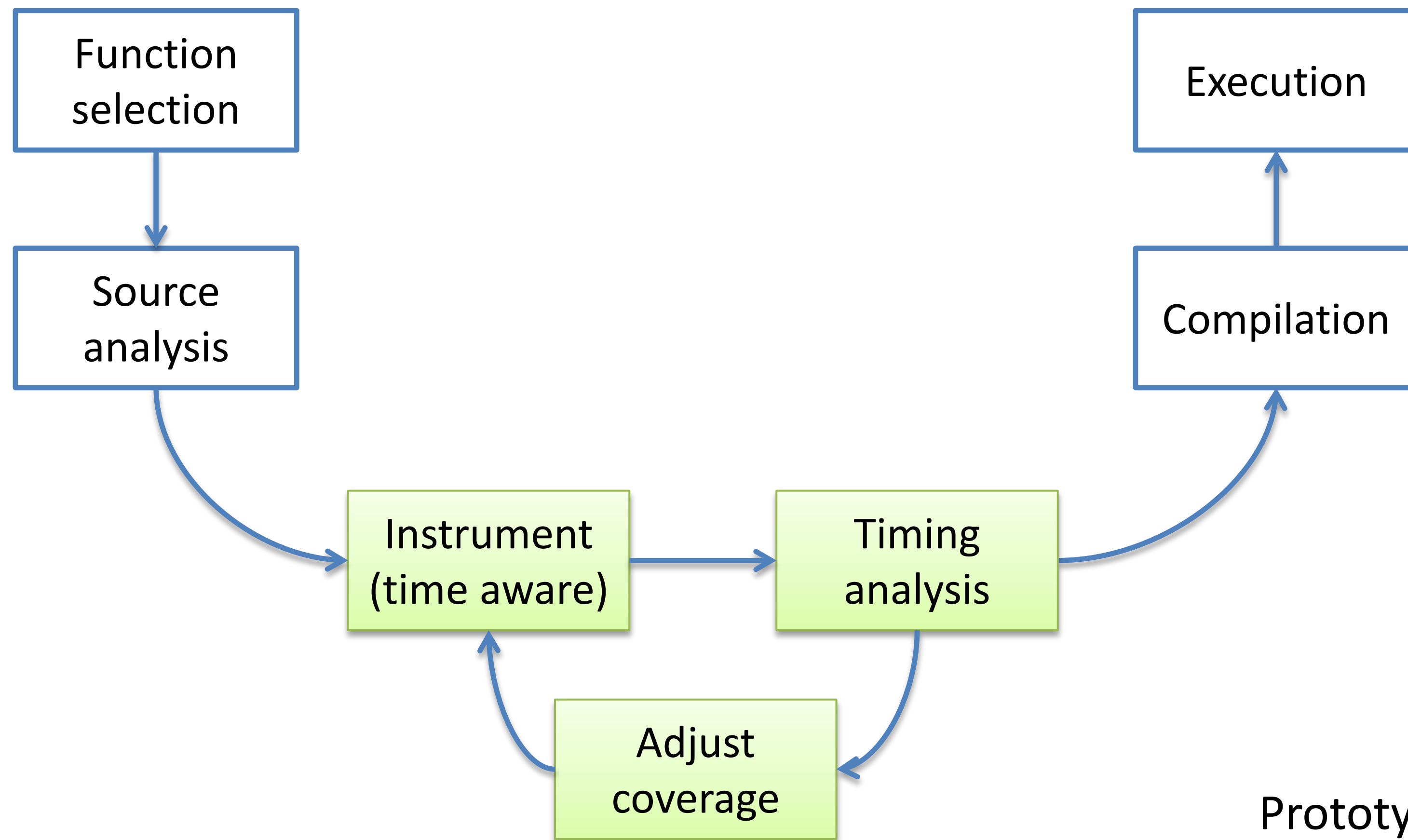
- **Coverage of an insertion point:** *p* of the last branching point
- **Coverage of a path:** miss ratio of assignments to logged assignments on the path
- **Coverage of an instrumentation:** miss ratio of on all paths

Optimality: For a given time budget, what placement of log statements yields the best coverage? [RTAS'09, TII]

Standard Toolchain



Time-aware Instrumenation Toolchain



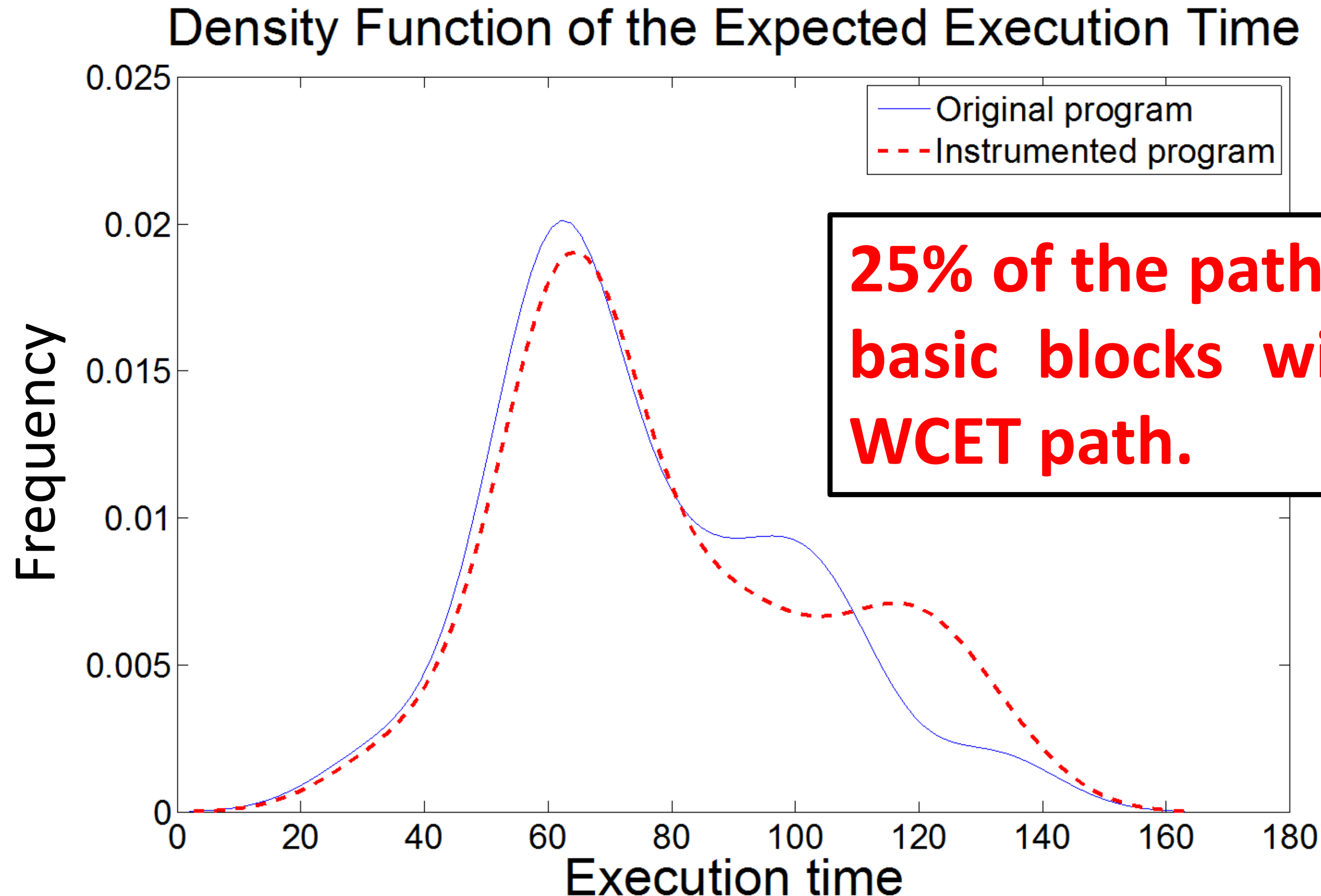
Prototypes for:

- ARM9
- ATMEL

Case Study: OLPC Keyboard Controller

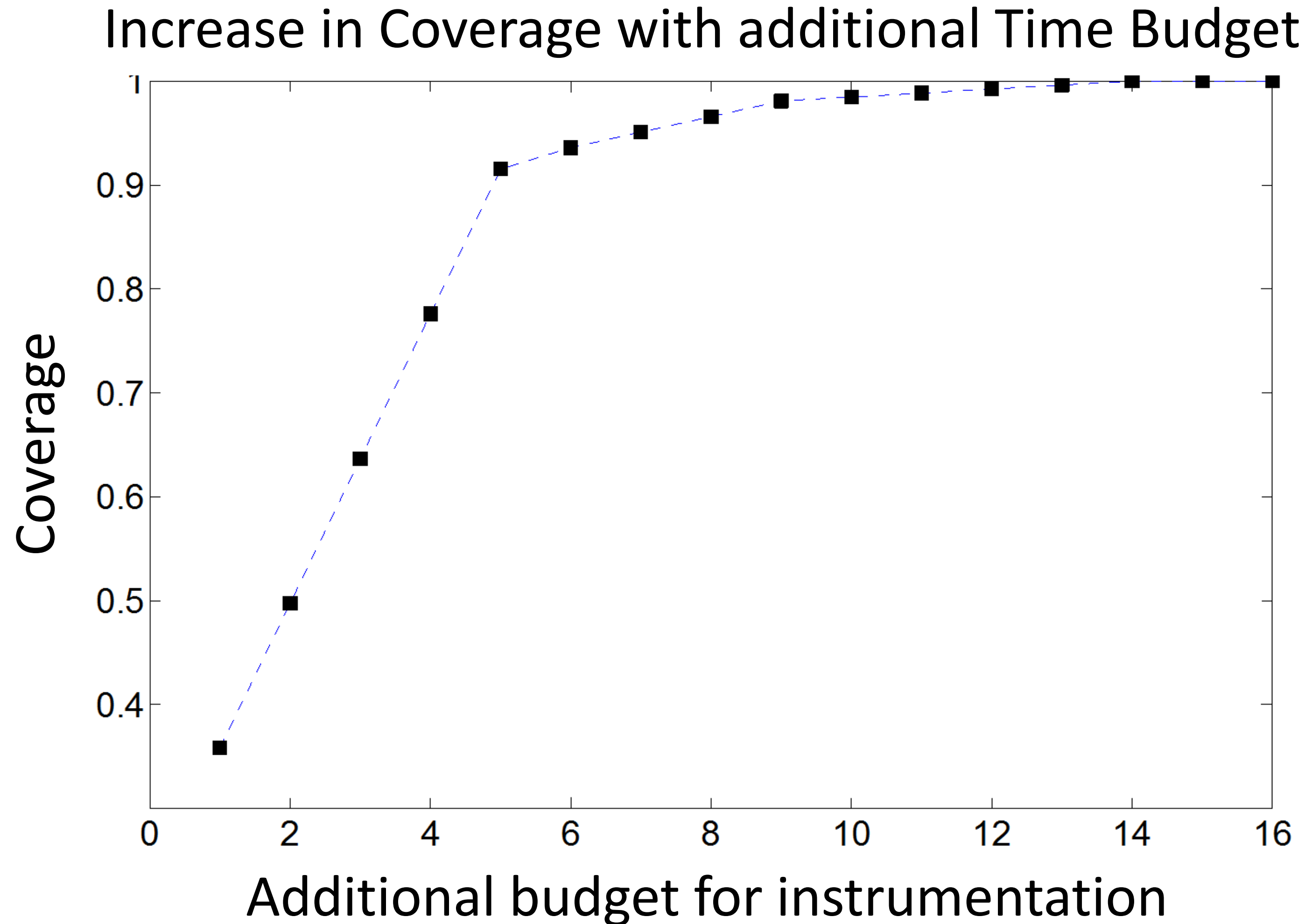
- Test feasibility
- Test hypothesis that shift in execution time occurs
- Experiment with time budgets

handle_power()



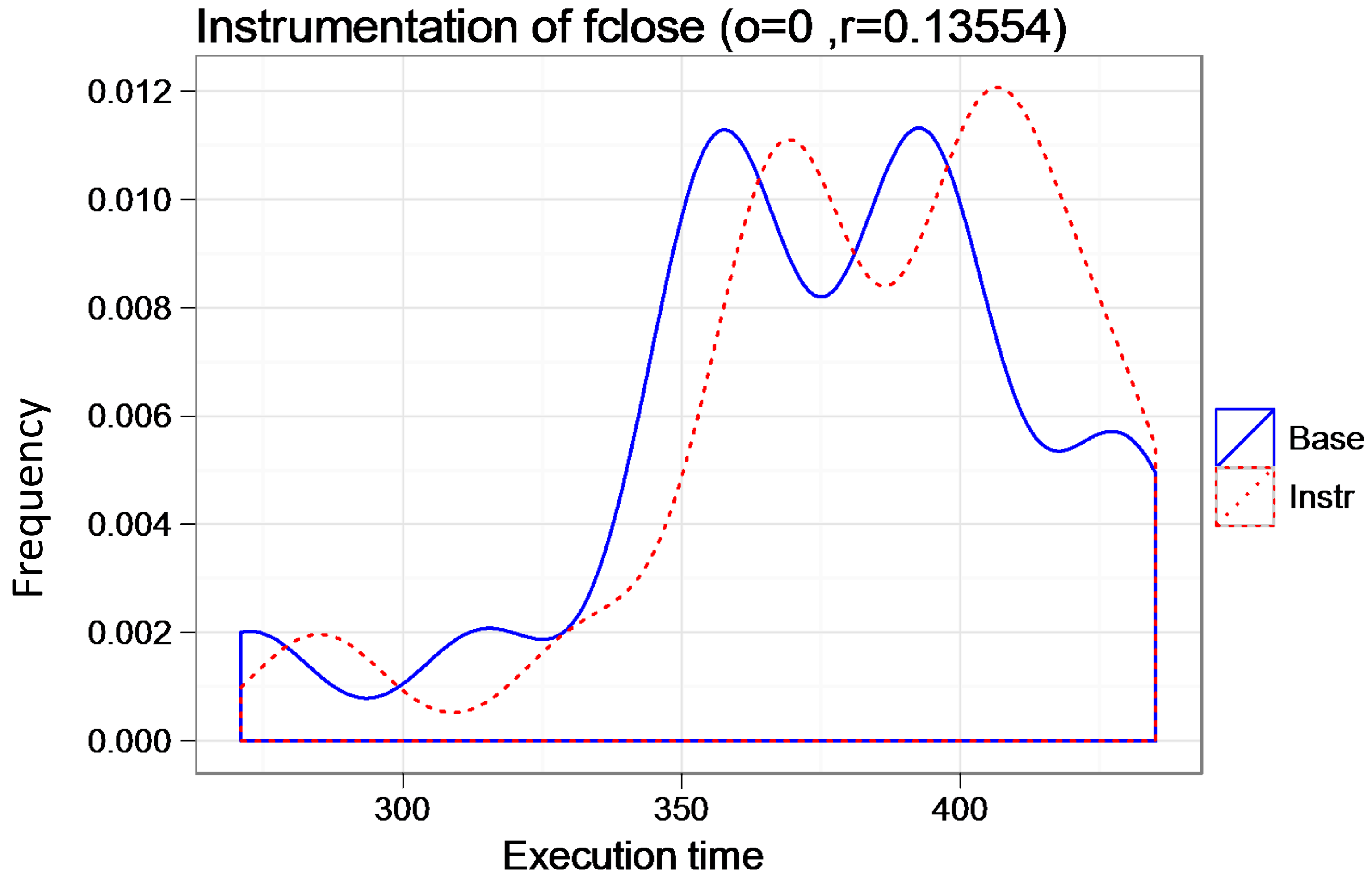
- **Approach works, but effects are limited without extra time budget.**

Increasing the Time Budget



- **Small increase in the time budget has huge effects.**

Case Study: Embedded FS



Case Study: Embedded FS

Instrumentation of fputs (o=0 ,r=0.21667)

Instrumentation of fputs (o=3 ,r=0.63258)

Instrumentation of fputs (o=10 ,r=0.9697)

Frequency

0.015
0.010
0.005
0.000

20 30 40 50 60 70

Execution time

Base
Instr

Ongoing Work on TAI

- What makes a program instrumentable?
- Can we transform a program to be more suitable for (time-aware) instrumentation?
- What other properties than time are of interest? *(arbitrary non-functional properties)*

Summary (TAI)

- Instrumentation can be time aware.
- The **"right shift" idea works** and is technically feasible.
- Long-term vision:
  - New methods with better coverage
  - **New methods for other properties**
  - Software & hardware hybrid solutions

TIME-TRIGGERED RUNTIME VERIFICATION

Bonakdarpour, B., S. Navabpour, and S. Fischmeister, "Sampling-based Runtime Verification", Proc. of the International Symposium on Formal Methods (FM), Limerick, Ireland, June, 2011.

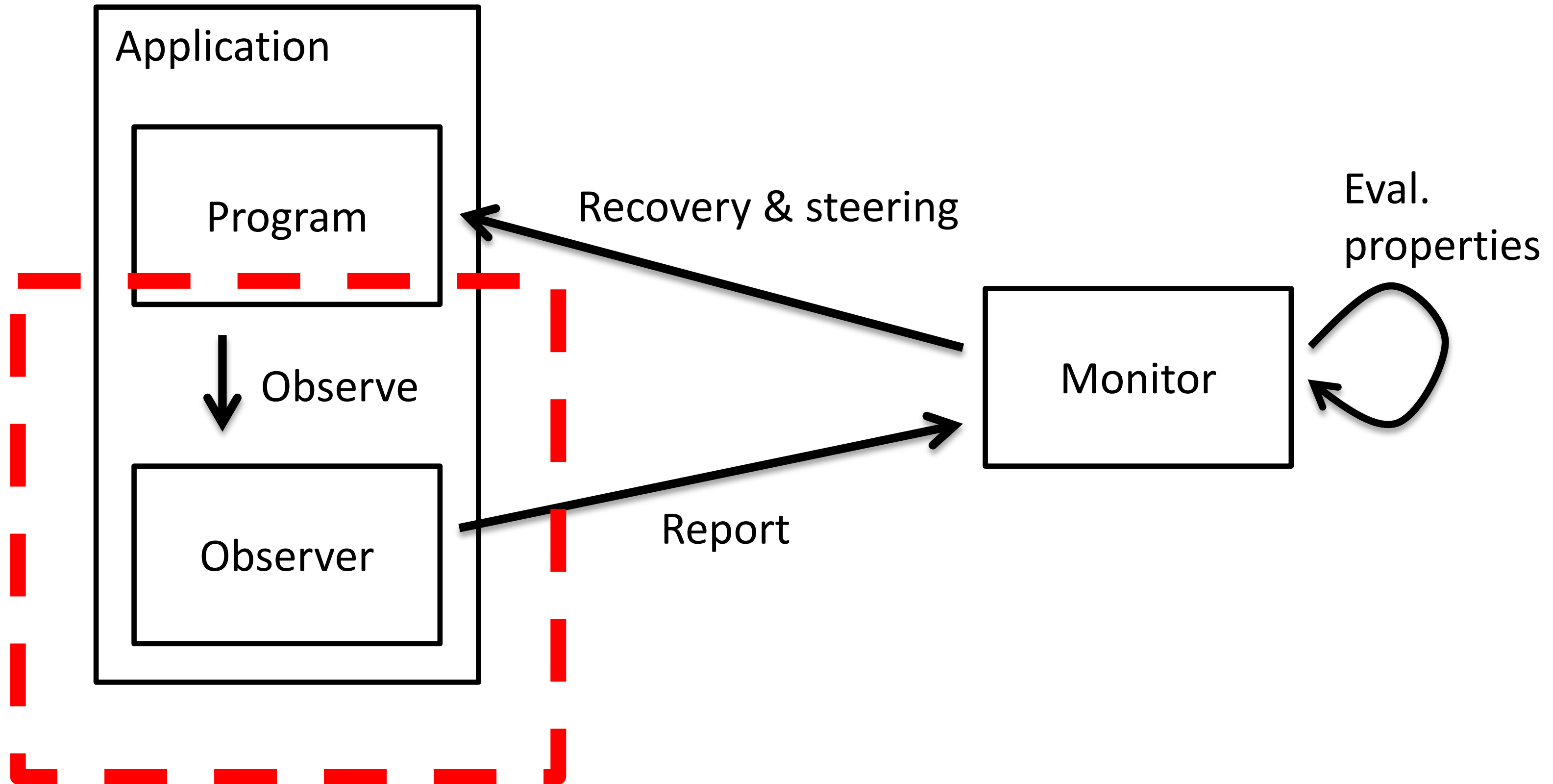
Navabpour, S., C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister, "Efficient Techniques for Near-optimal Instrumentation in Time-triggered Runtime Verification", Proc. of the 2nd International Conference on Runtime Verification (RV), San Francisco, USA, September, 2011.

Runtime Verification

- Observing program to check compliance with some specification.
  - Online, offline (traces)

- Example uses:
  - Runtime validation and safety
  - System steering
  - Performance monitoring and tuning
  - Debugging

An Online External RV System



Problem

Current approaches are event-triggered and can lead to transient overloads at run time.

Can we observe the program with predictable overhead?

Event-based Runtime Verification

We *instrument* lines 5 and 6 such that the monitor is invoked.

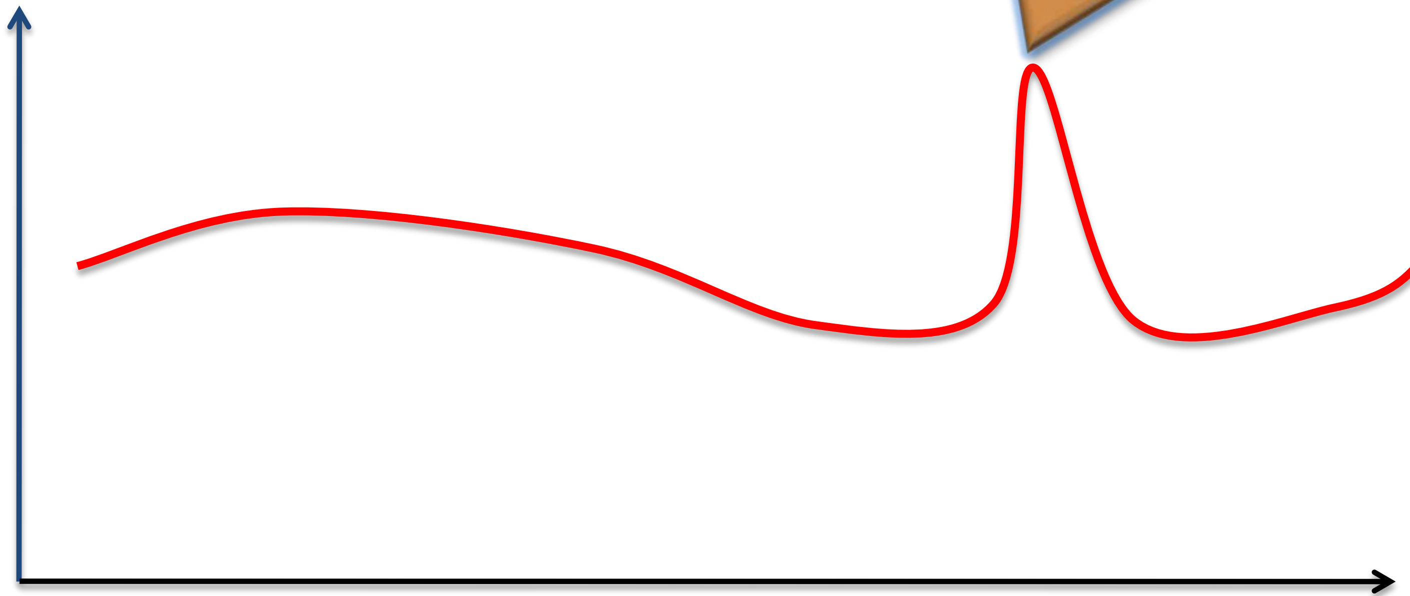
```
        a = scanf(...);
        if (a % 2 == 0) goto 9
        else {
4            printf(a + "is odd");
5:*          b = a/2;
6:*          c = a/2 + 1;
7:           goto 10;
8:      }
9:      printf(a + "is even");
10:     end program
```

'b' and *'c'* of interest

ET has Problems (Overhead)

Spikes in overhead are a problem in *real-time embedded* systems

Overhead



Time

Idea in a Nutshell

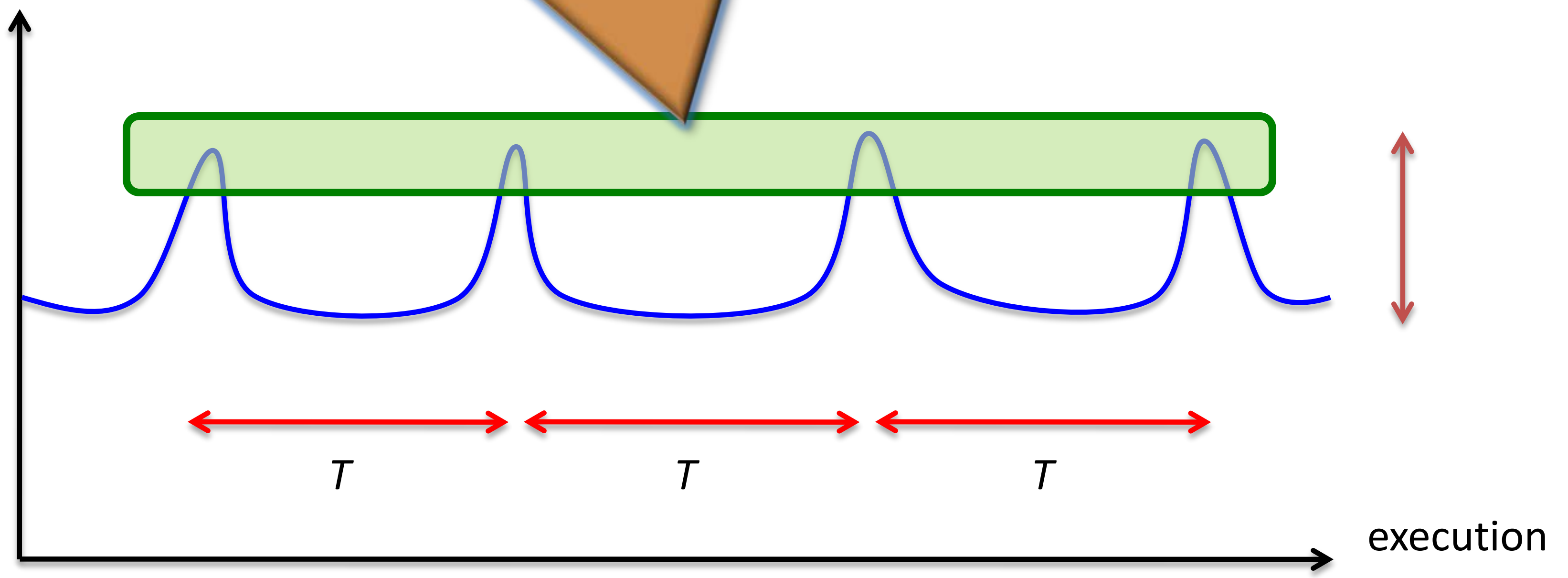
- Our idea is to ***bound*** the overhead of runtime verification and make it ***predictable*** ***(=> engineerable)***.

- We analyze the correctness of the system in a ***time-triggered*** fashion:
  - At the end of each period, the monitor is invoked to take a ***sample*** from the system to analyze its soundness.

Objective

Bounded and predictable overhead

overhead

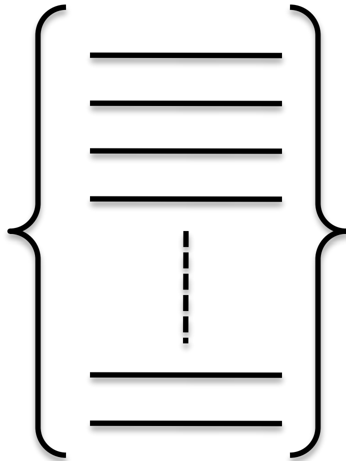


TTRV Problem 1

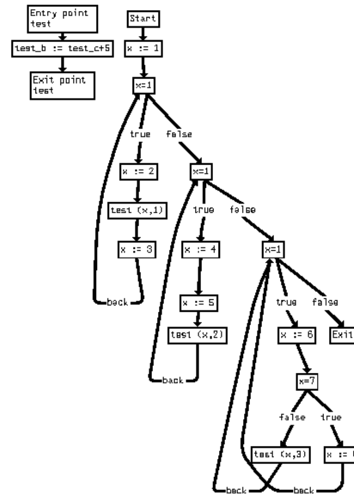
Identify the sampling period, such that the monitor observes all changes vital to evaluating the correctness of a given property.

Our Approach

Property



C
Program

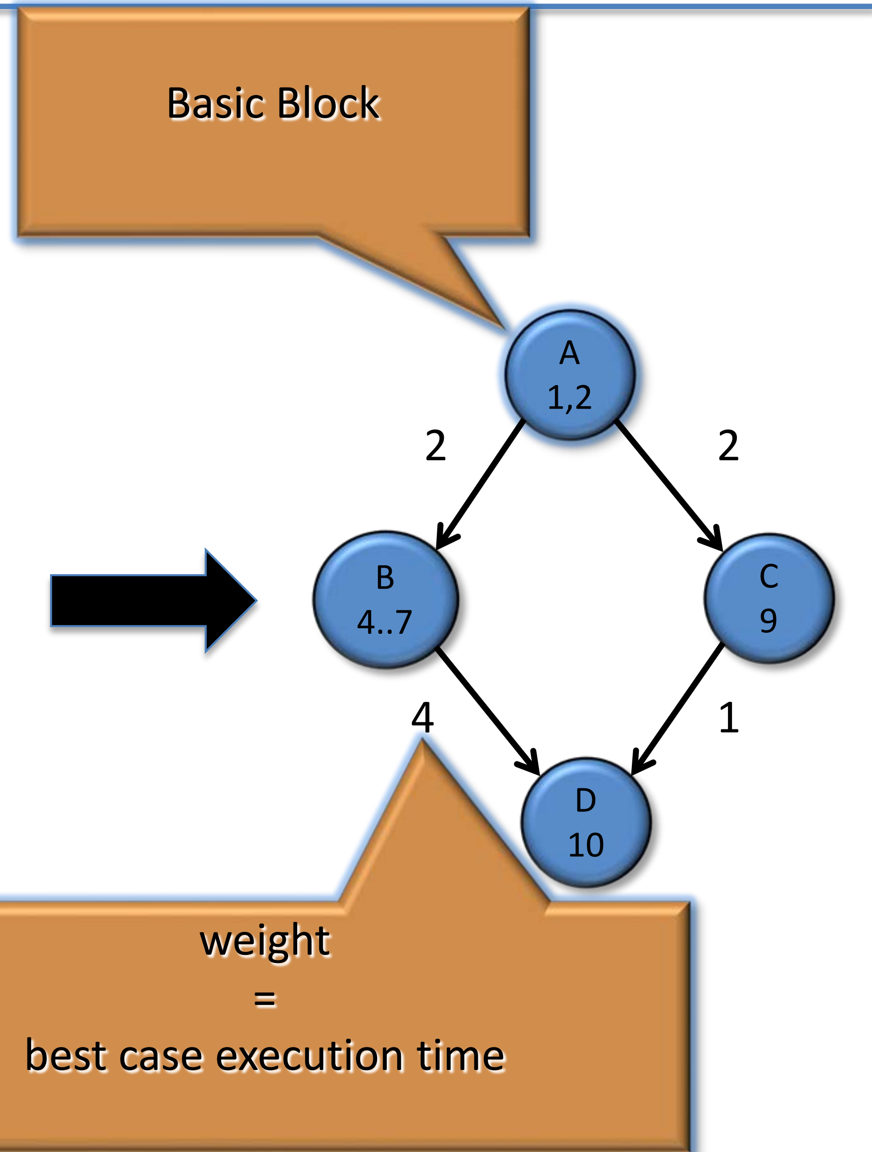


Control Flow Graph

Sampling
Frequency

Generating CFG

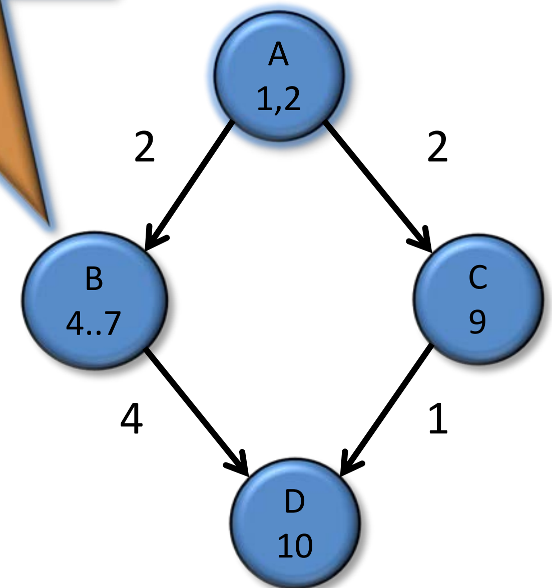
```
1:      a = scanf(...);
2:      if (a % 2 == 0) goto 9
3:      else {
4:            printf(a + "is odd");
5:*           b = a/2;
6:*           c = a/2 + 1;
7:            goto 10;
8:      }
9:      printf(a + "is even");
10:     end program
```



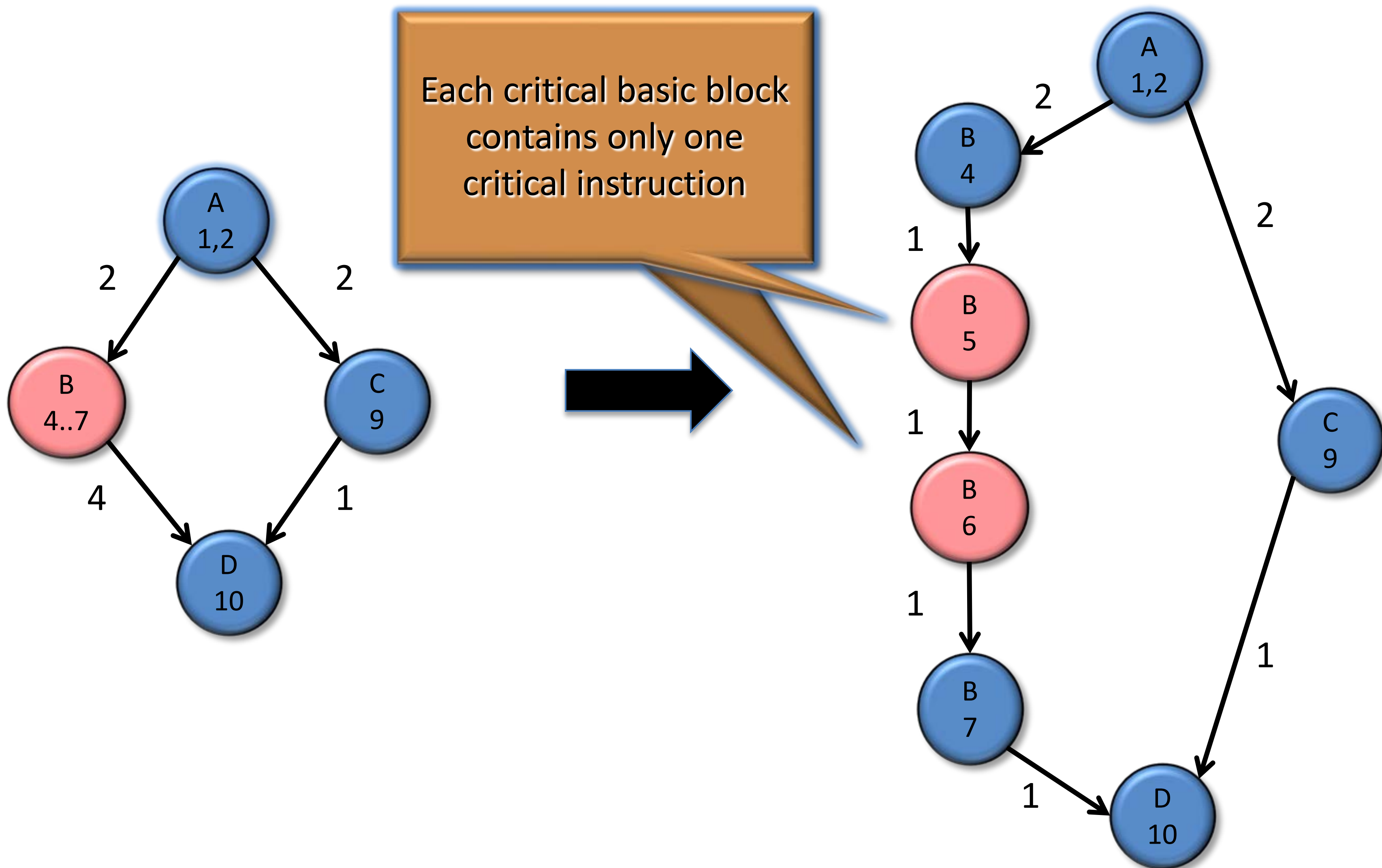
Calculating Sampling Period (1)

Critical Basic Block

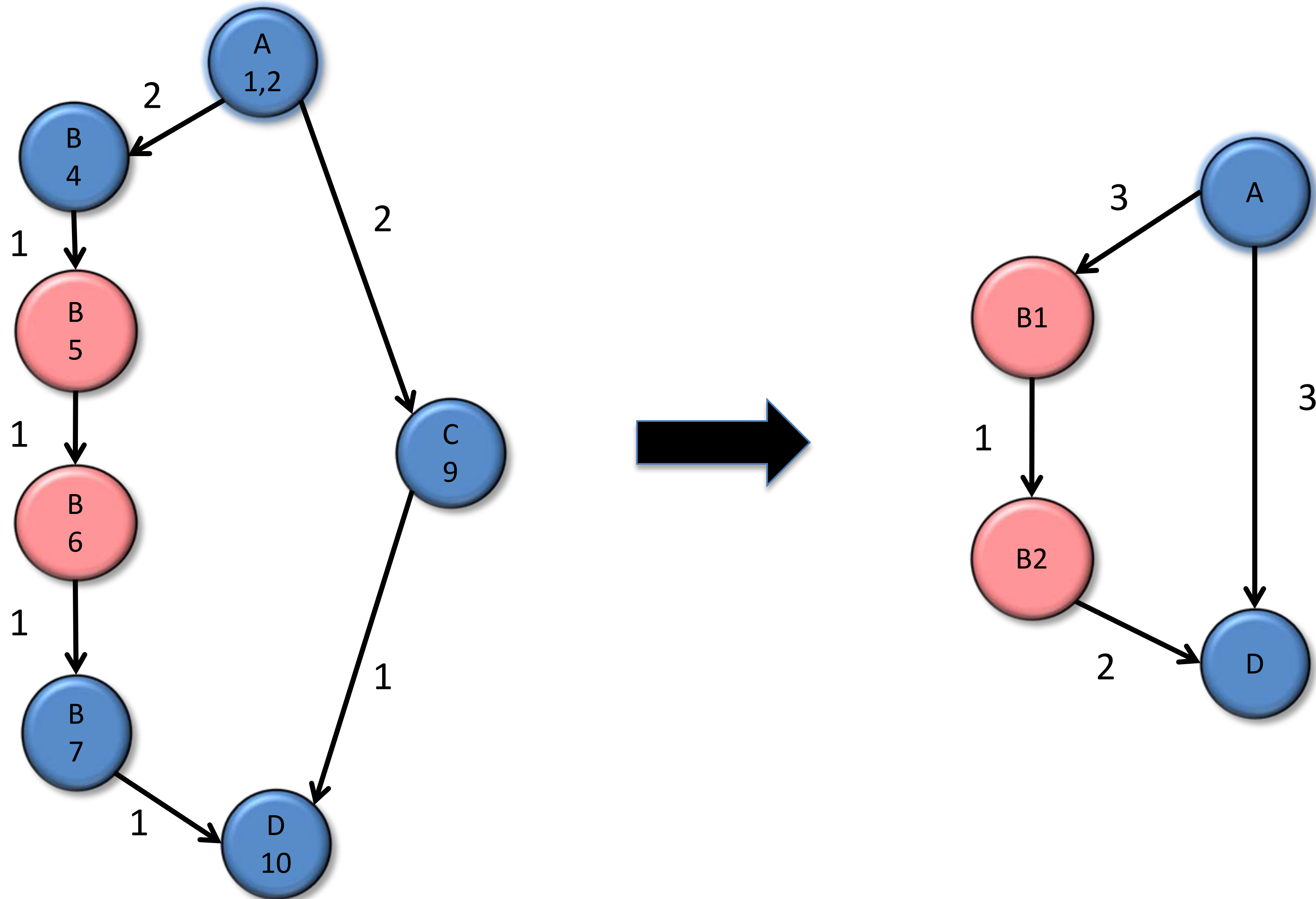
```
1:      a = scanf(...);
2:      if (a % 2 == 0) goto 9
3:      else {
4:              printf(a + "is odd");
5:*             b = a/2;
6:*             c = a/2 + 1;
7:              goto 10;
8:      }
9:      printf(a + "is even");
10:     end program
```



Calculating Sampling Period (2)



Calculating Sampling Period (3)

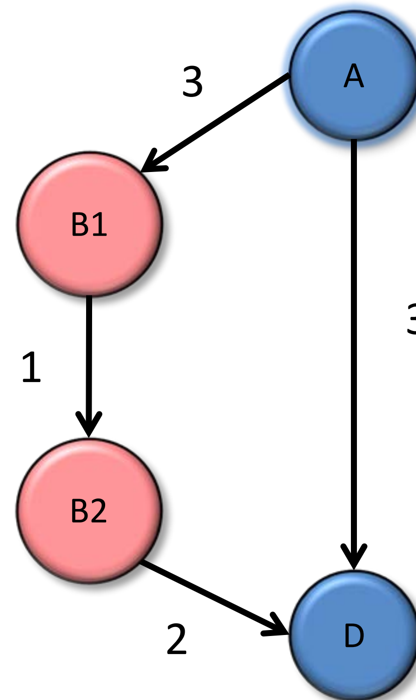


Minimum Sampling Period

The *minimum sampling period* for a property is the minimum arc weight that originates from a corresponding critical basic block.

Computing the Sampling Period

Sampling Period = 1



TTRV Problem 2

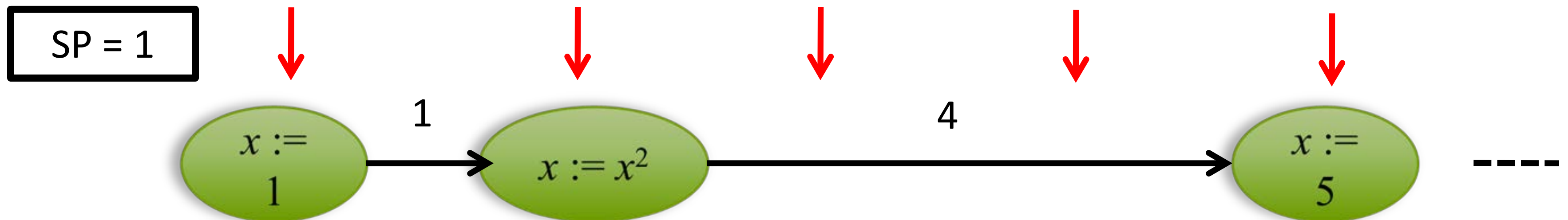
The basic sampling period can be very small. Can we increase the sampling period?

Use history information to increase the minimal sampling period.

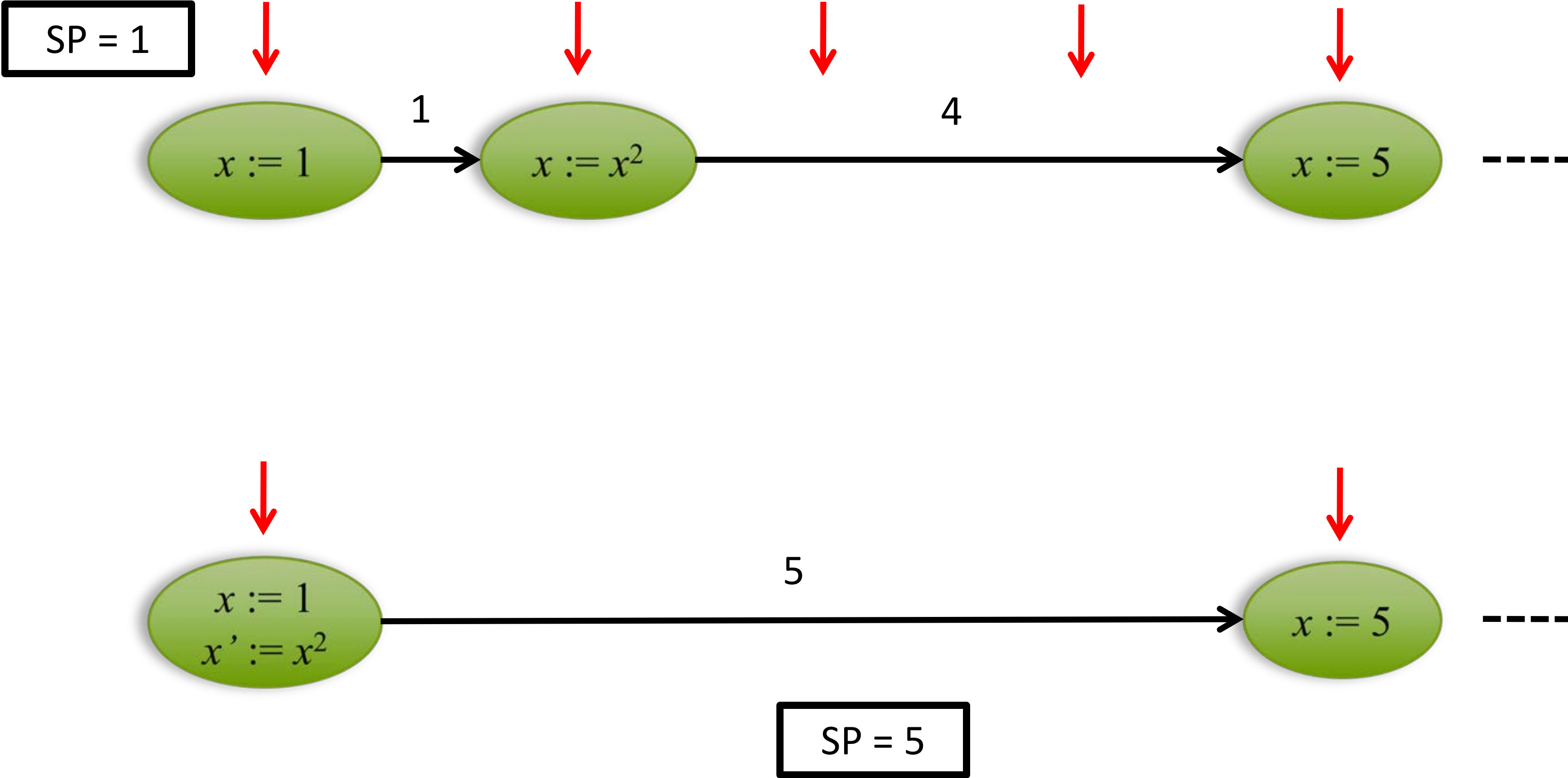
Calculating Sampling Period

The Problem in Detail

Minimum sampling period is a conservative estimate and often results in sampling with high frequency and *over-sampling* in some execution branches.



Solution: Leveraging Histories



Optimization Problem

Size of History

Sampling period

Instance. A weighted digraph $G = < V, A, w >$ and positive integers X and Y.

Decision problem. Does there exist a set U *of vertices*, such that by collapsing all vertices in U, we obtain a weighted digraph $G = < V', A', w' >$, where $|U| \leq Y$ and for all arcs (u, v) in A', $w'(u, v) \geq X$?

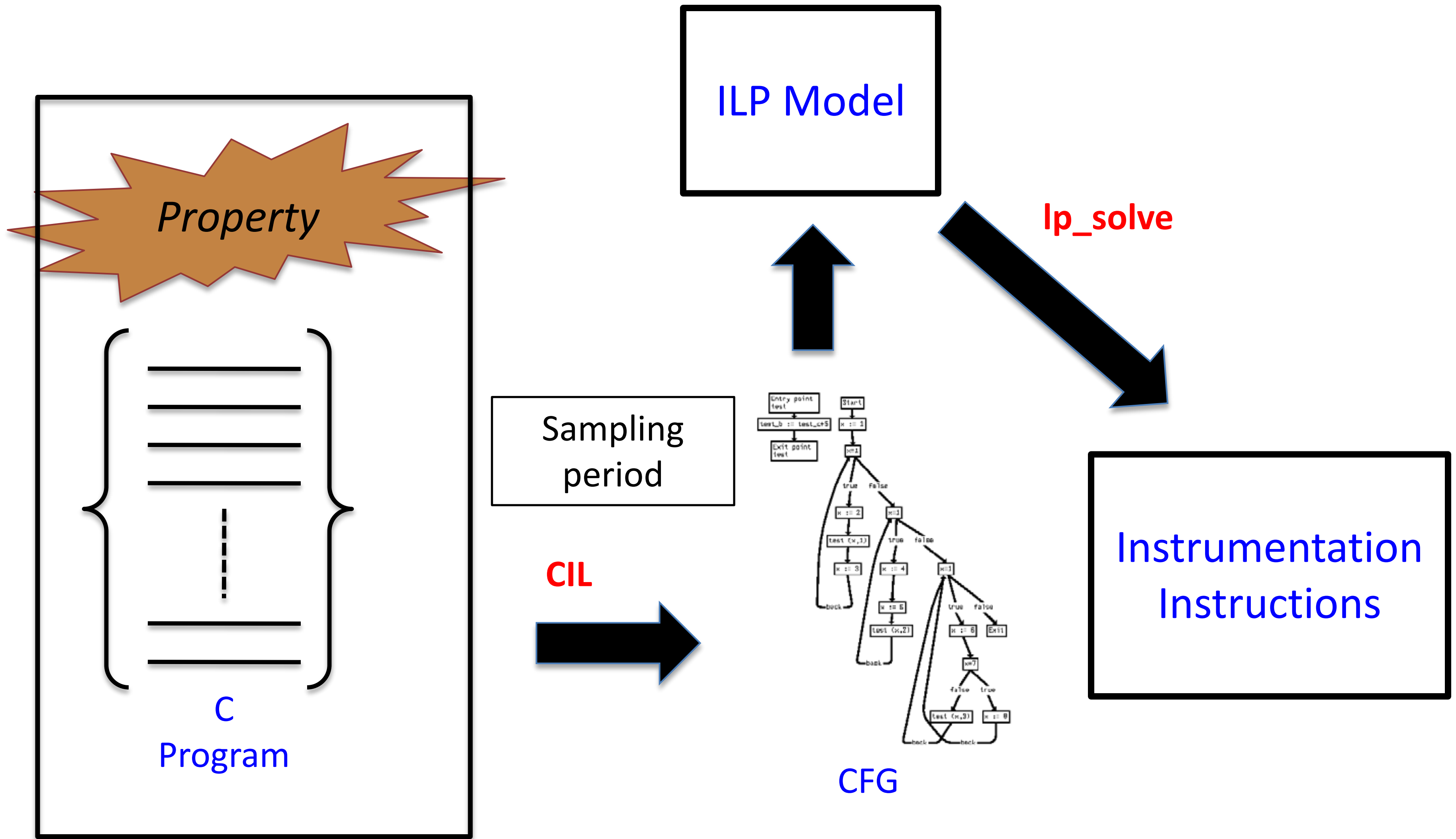
Mapping to ILP

$$\begin{cases} \text{Minimize} \quad c.\mathrm{z} \\ \\ \text{Subject to } A.\mathrm{z} \geq \mathrm{b} \end{cases}$$

- Variables
  - *x* for collapsing vertices
  - *a* for arc weights
  - *y* as choice variables to simplify the encoding

- Constraints
  - All arc weights must be greater than the sampling period
  - Updates on arc weights when collapsing vertices
  - Loops with critical vertices must not be collapsed

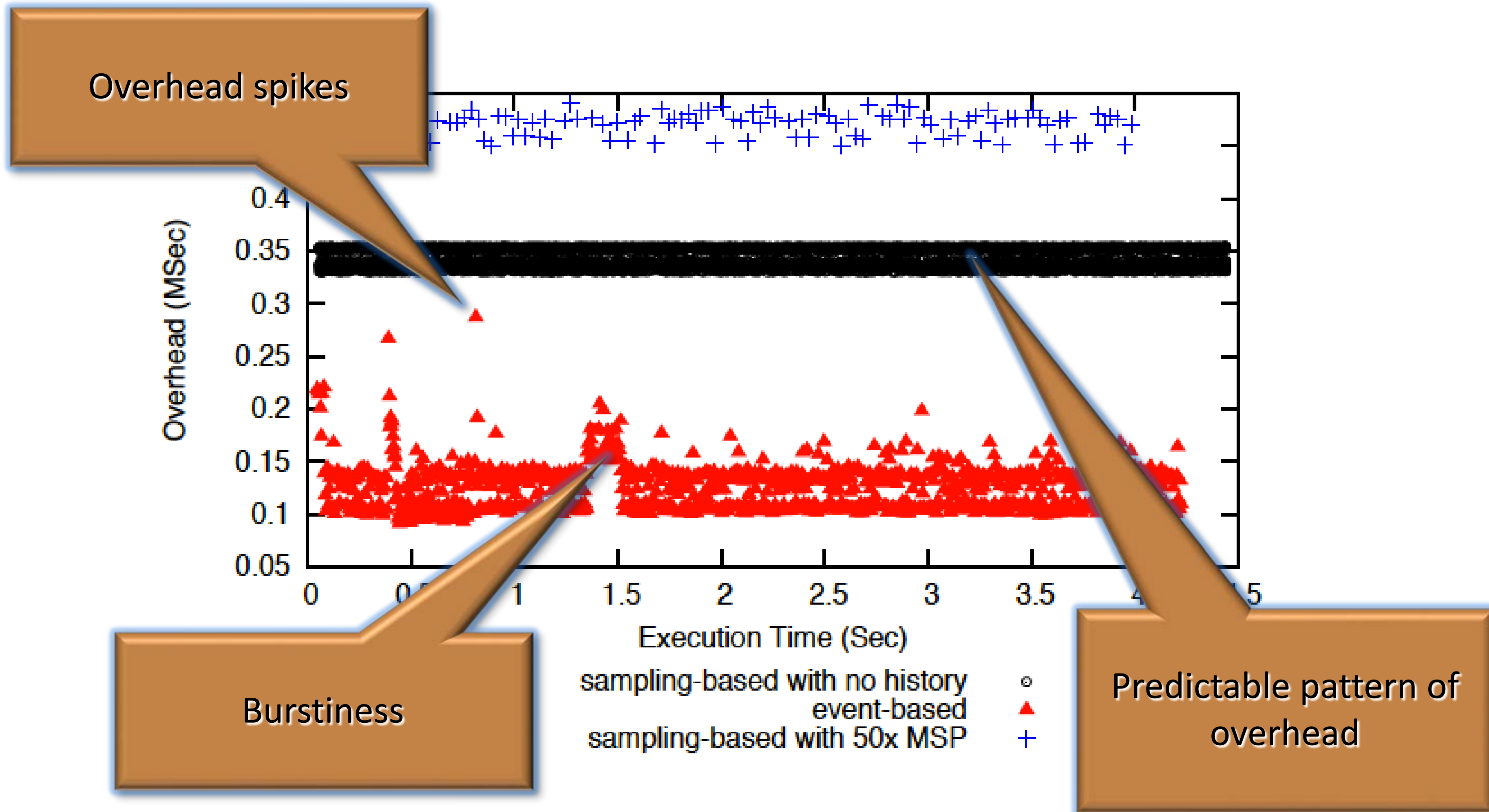
Prototype Tool Chain



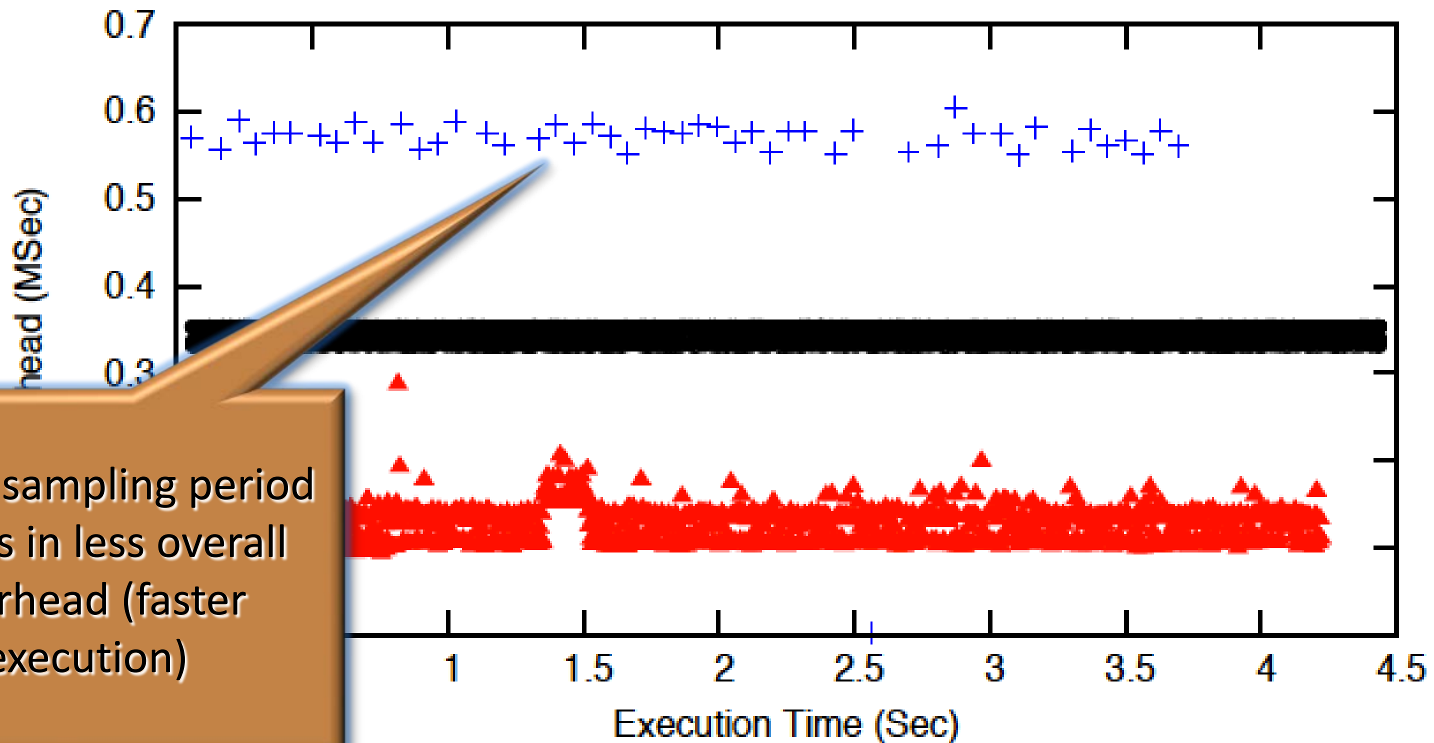
Experimental Setting (from MiBench)

- *Blowfish:* 745 lines of code. Results in a CFG of 169 vertices and 213 arcs. We take 20 variables for monitoring.
- *Dijkstra:* 171 lines of code. Results in a CFG of 65 vertices and 78 arcs. We take 8 variables for monitoring.
- 3 experiments
  - Event-triggered monitoring
  - Sampling-based monitoring without history
  - Sampling-based monitoring with history
- All experiments are conducted on a Mac Book Pro with 2.26GHz Intel Core 2 Duo and 2GB main memory.

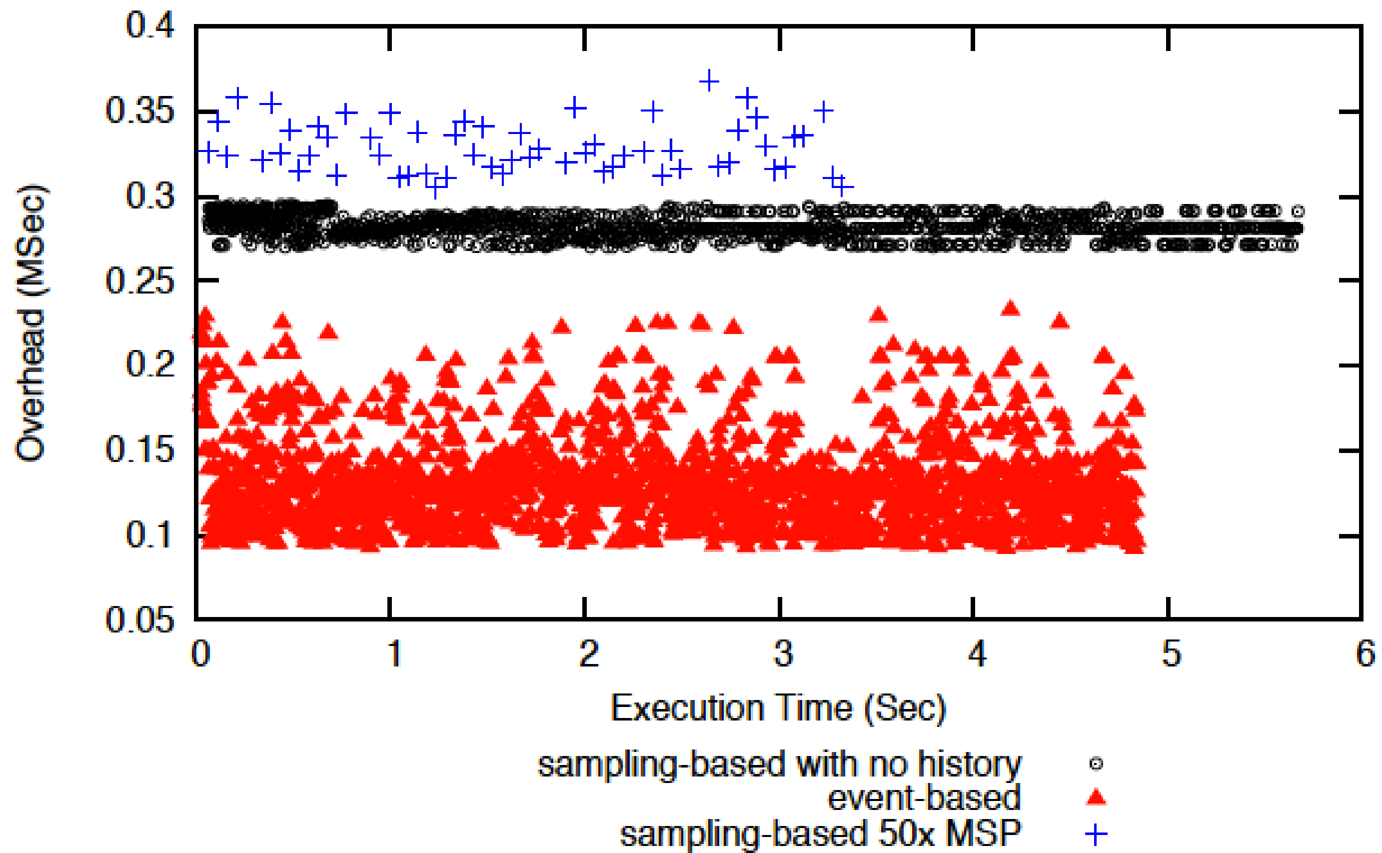
Experimental Results (Blowfish – 50x)



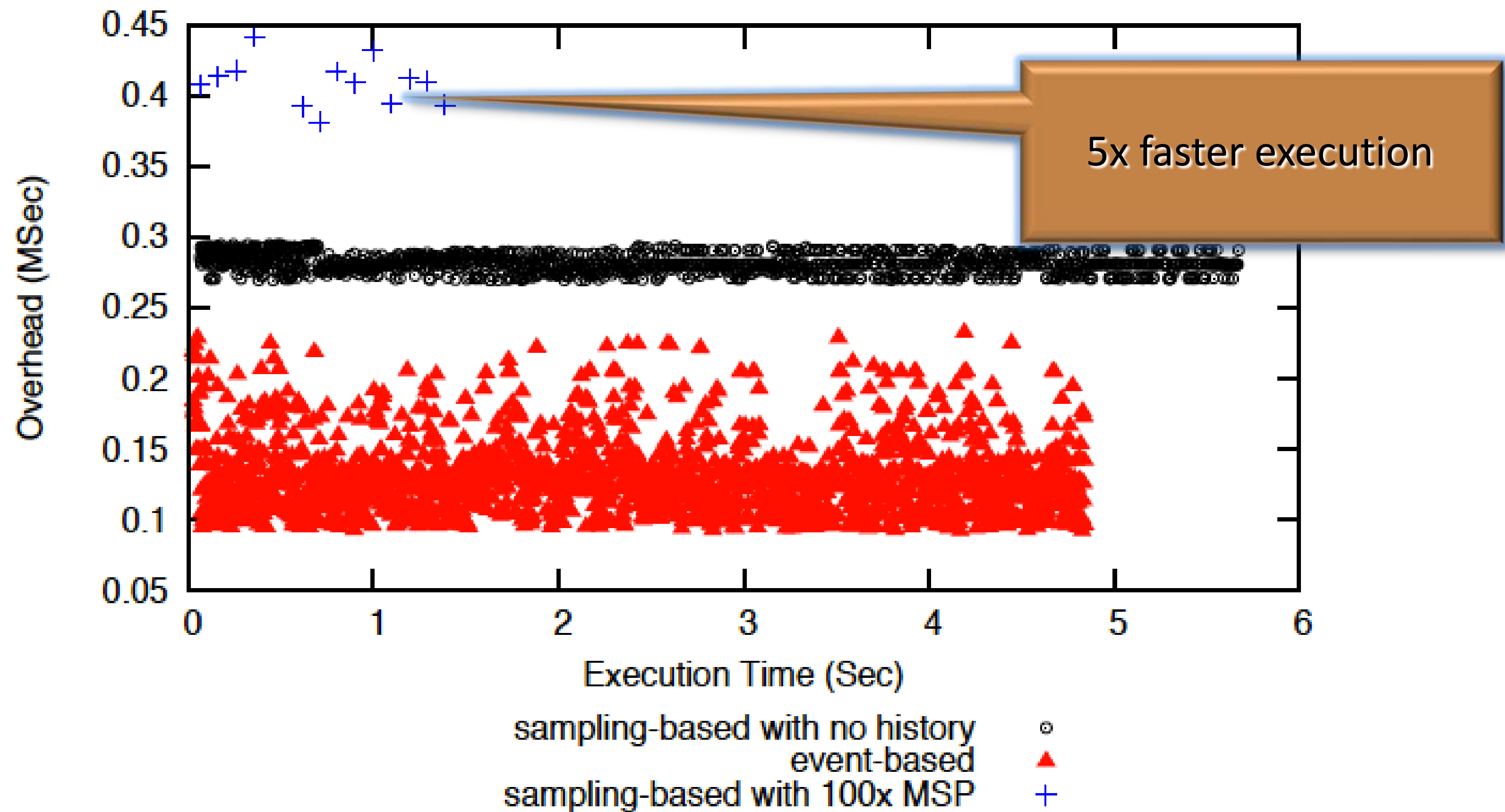
Experimental Results (Blowfish – 100x)



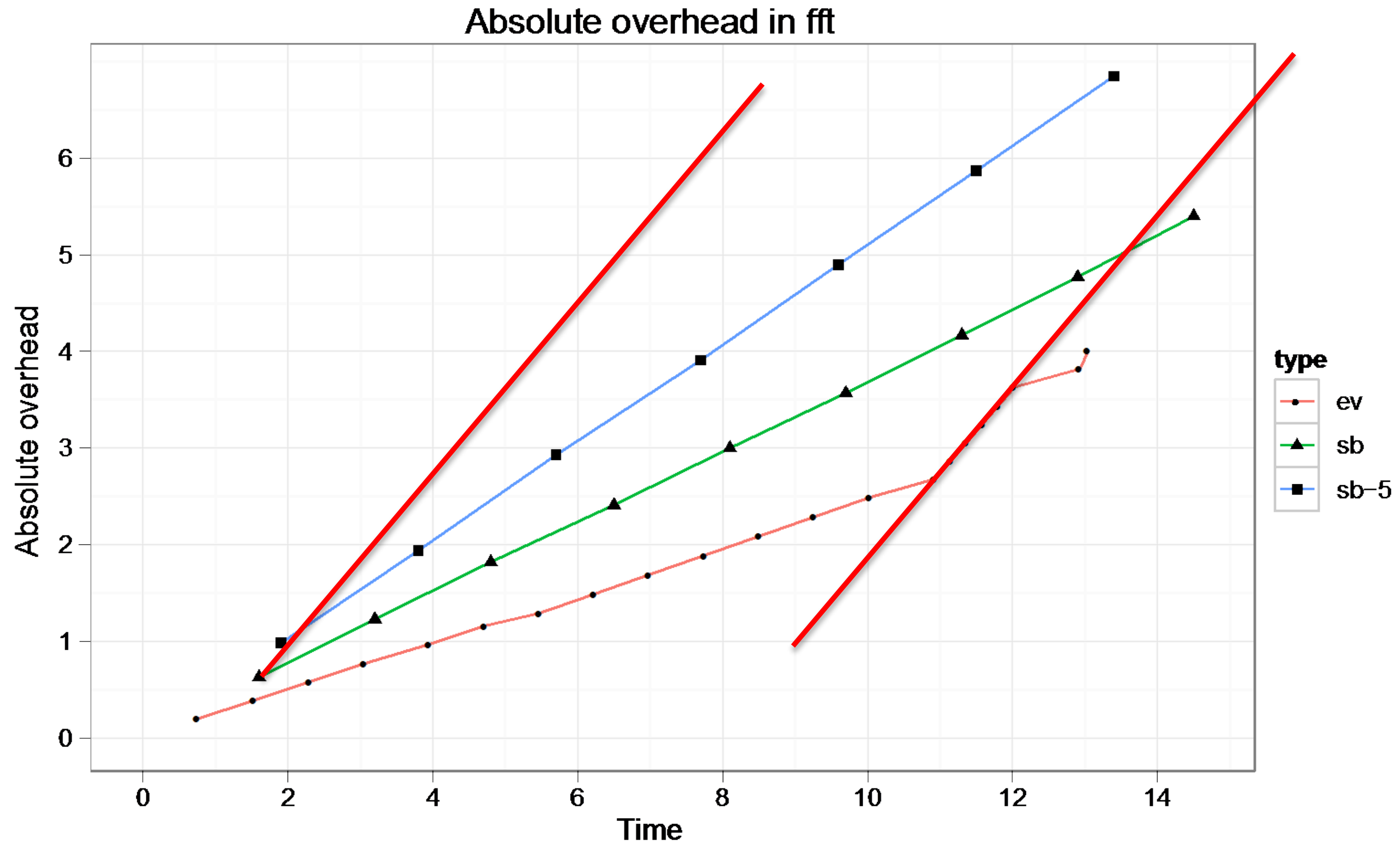
Experimental Results (Dijkstra – 50x)



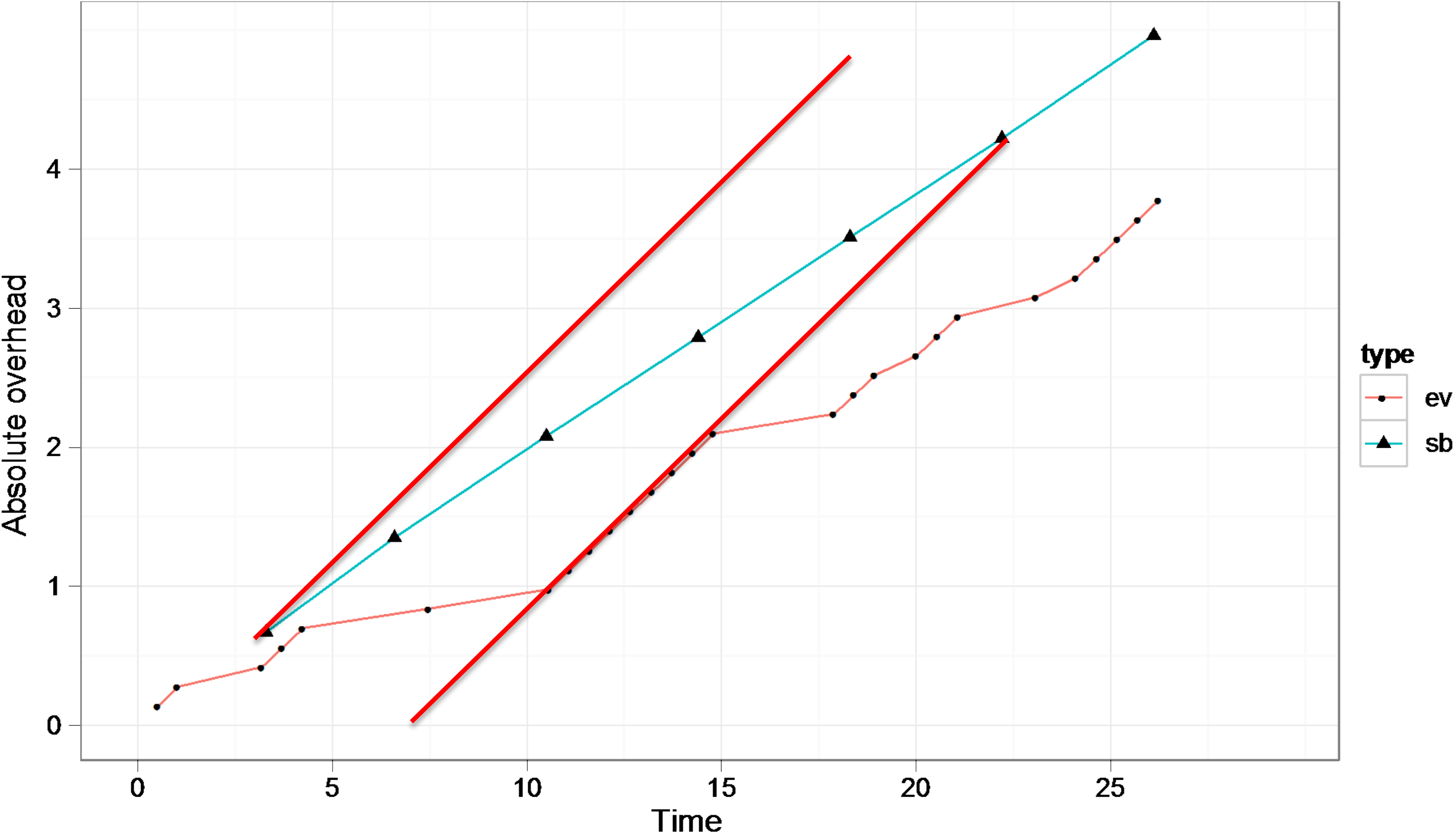
Experimental Results (Dijkstra – 100x)



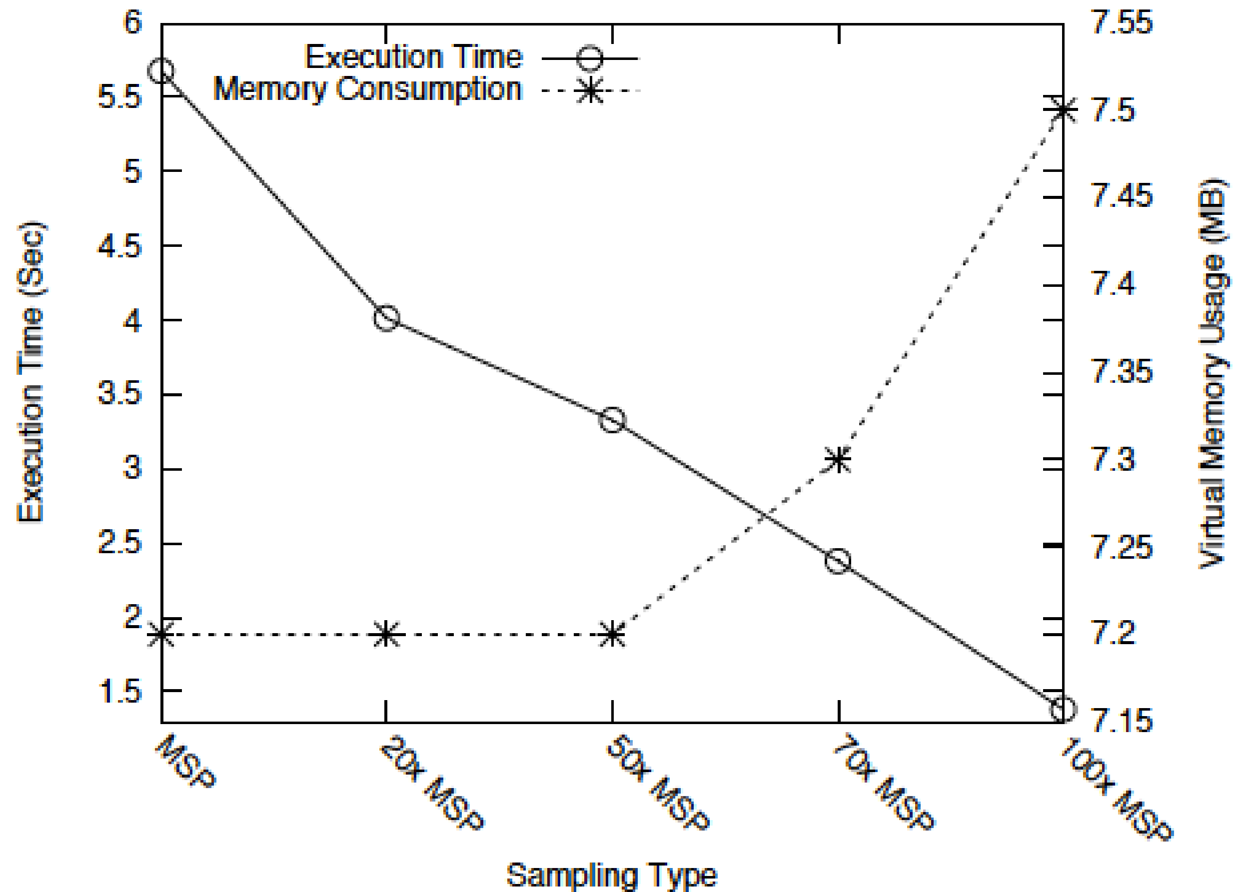
Cumulative Overhead



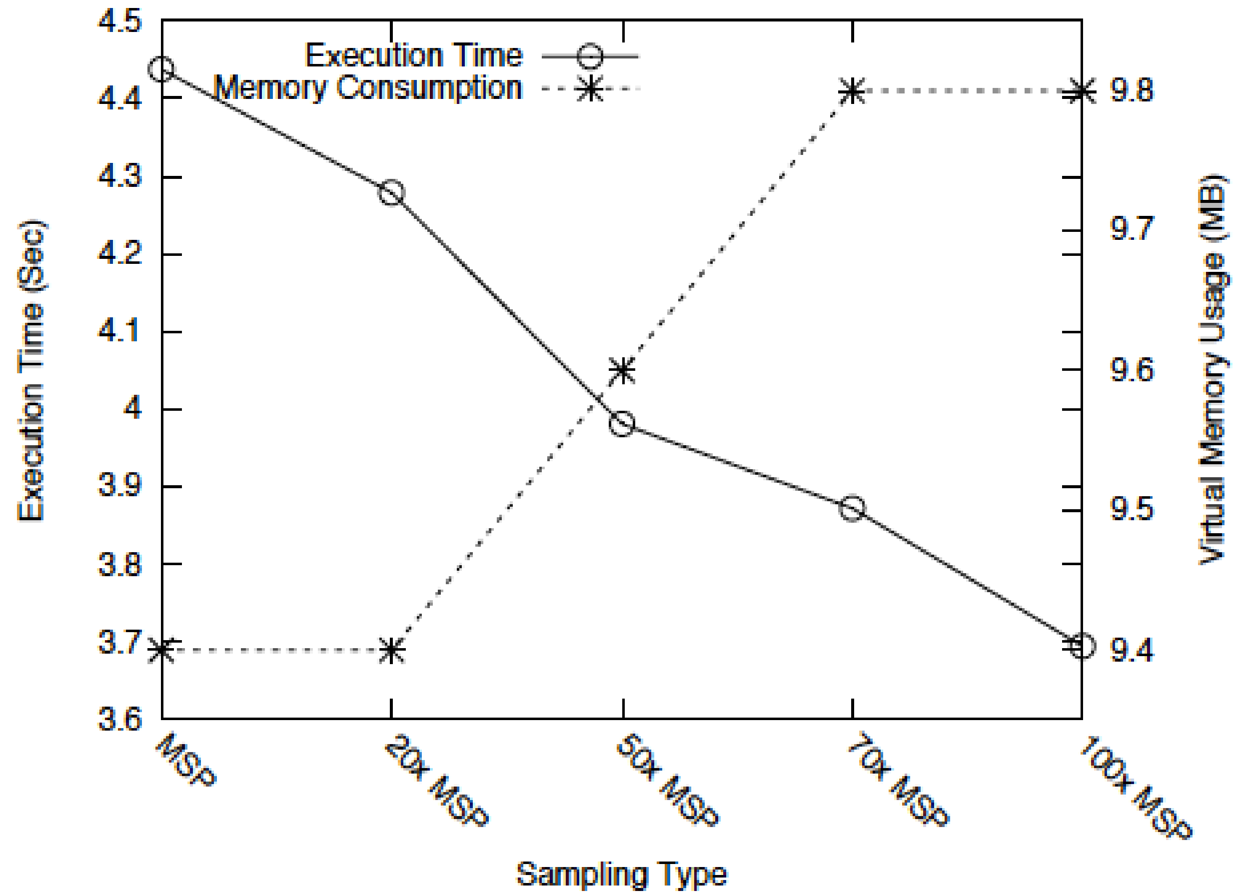
Cumulative Overhead



Experimental Results (Dijkstra – Memory)



Experimental Results (Blowfish – Memory)



TTRV Problem 3

Optimal use of a history buffer is NP-complete with the size of the CFG.

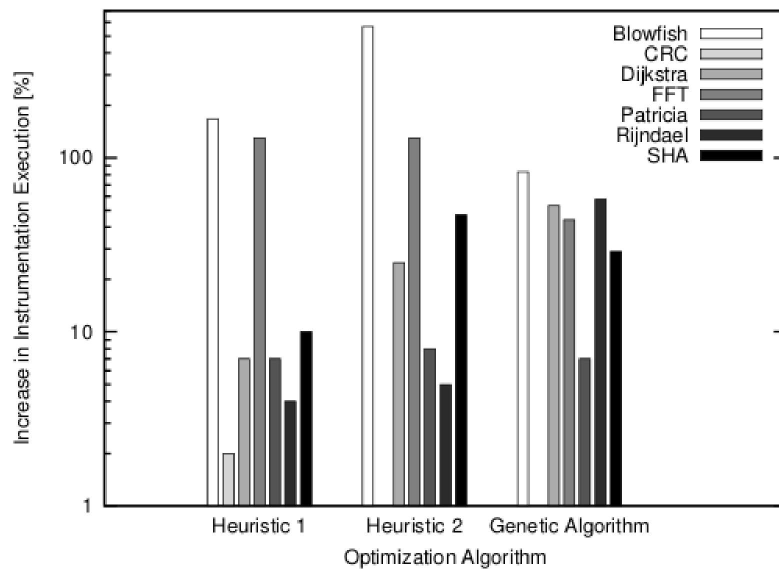
Can we find a near-optimal solution in reasonable time?

Short Answer: Yes

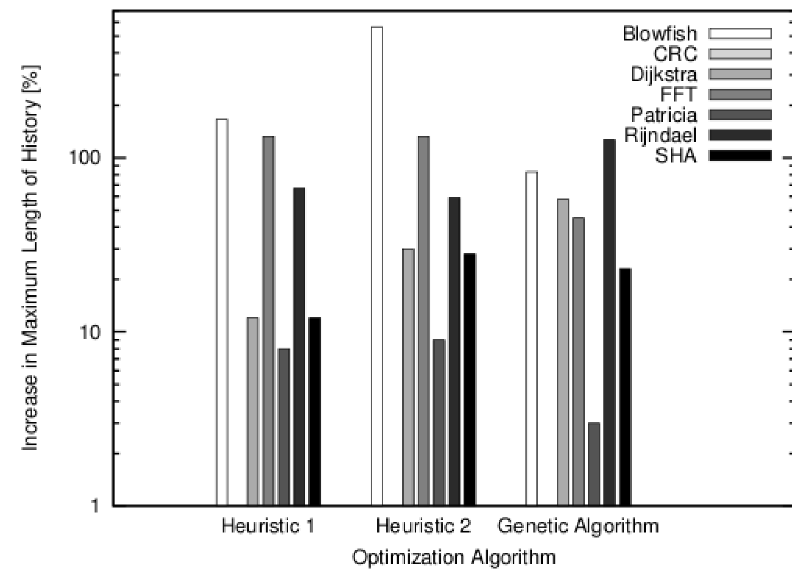
	CFG Size(|V|)	*ILP*		*Heuristic 1 (Greedy)*		*Heuristic 2 (VC)*		*Genetic Algorithm*	
		time (s)	**SOF**	**time (s)**	**SOF**	**time (s)**	**SOF**	**time (s)**	**SOF**
Blowfish	177	5316	–	0.0363	7.8	0.8875	8	383	2.5
CRC	13	0.35	–	0.0002	3.5	0.0852	3	0.254	1.5
Dijkstra	48	1808	–	0.0064	1.2	0.1400	1.2	116	1.7
FFT	47	269	–	0.0042	1.7	0.1737	1.8	74	1.1
Patricia	49	2084	–	0.0054	1.4	0.1369	1.6	140	1.5
Rijndael	70	3096	–	0.0060	1.6	0.2557	2.1	370	1.9
SHA	40	124	–	0.0039	2.2	0.1545	2.2	46	1.3
Susan	20 259	∞	–	3 181	N/A	26 211	N/A	923	N/A

Table 1. Performance of different optimization techniques.

Short Answer: Yes



(a) Increase in the number of execution of instrumentation instructions.



(b) Increase in the maximum size of history between two samples.

Summary (TTRV)

- A time-triggered approach is a feasible approach for runtime verification
- Surprising result that TTRV can even lead to better overall performance
- Lot of open problems:
  - Multicore? Integration of the monitor? Fair distribution of instrumentation? Adaptive and dynamic TTRV? Hybrid ET&TTRV? Concurrent applications?

Conclusions

- Embedded software is everywhere and increasing in complexity and size.
- Many development activities require comprehending the system, and we thus need information extraction.
- Current tool only support preserving logical correctness. The presented work provides a glimpse of what can be possible.

- While other disciplines have a thorough understanding of the probe effect, software engineering considers only logical correctness.

- **Understanding how to extract information from programs at run time is a widely unexplored area.**
  (=> software probe effect beyond logical correctness)

Acknowledgements

This research was mostly supported by the Canadian tax payer.

Thank you!

Questions?

(PS: Postdoc and grad student positions available, just talk to me afterwards or email me sfischme@uwaterloo.ca)