A FRAMEWORK FOR REDUCING THE COST OF INSTRUMENTED CODE

Known from…

- Continuous Path and Edge Profiling
- Bug Isolation via Remote Program Sampling
- Low-overhead Memory Leak Detection using Adaptive Statistical Profiling (SWAT)
- Accurate, Efficient, and Adaptive Calling Context Profiling

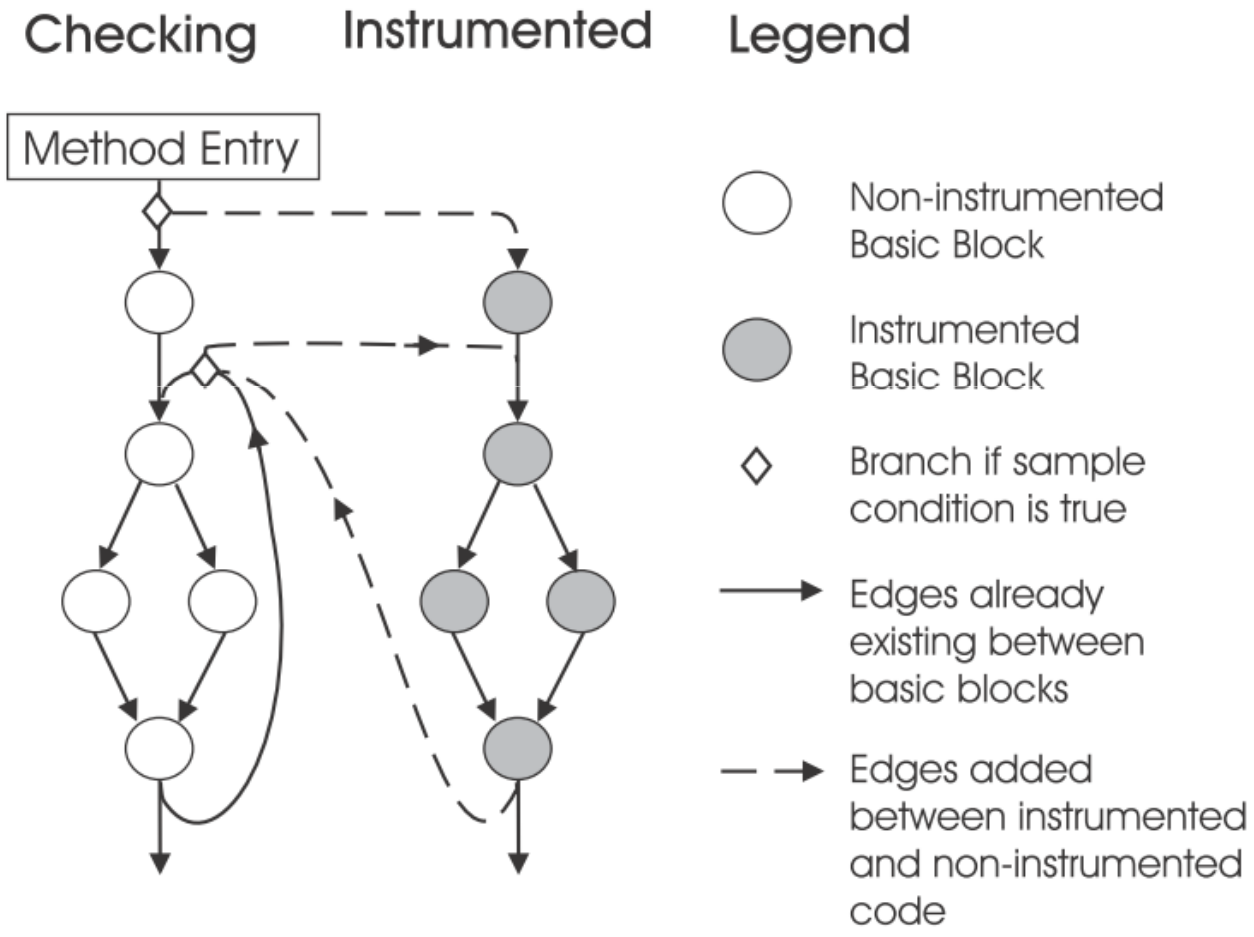
Problem

- JIT compilers need run-time sampling to make decisions
- Sampling code is expensive, sometimes reaching 30% to 10000%
- How to switch profiling on and off?

Wishlist

- Toggle instrumentation at any point in the lifecycle of the program
- Dynamically adjust the trade-off between accuracy and performance
- Adapt to different instrumentations
- Portability
- Deterministic behavior

Sampling Framework



How to trigger instrumentation?

- Samples should be statistically accurate – reproducibility would be even better
- Hardware / OS interrupts are not fine grained enough
- Operations following expensive ops are more likely to be sampled

Compiler-inserted counters

- Each *n*th check leads to a sample
- The program maintains a global counter
- Maintaining the counter is reasonably cheap

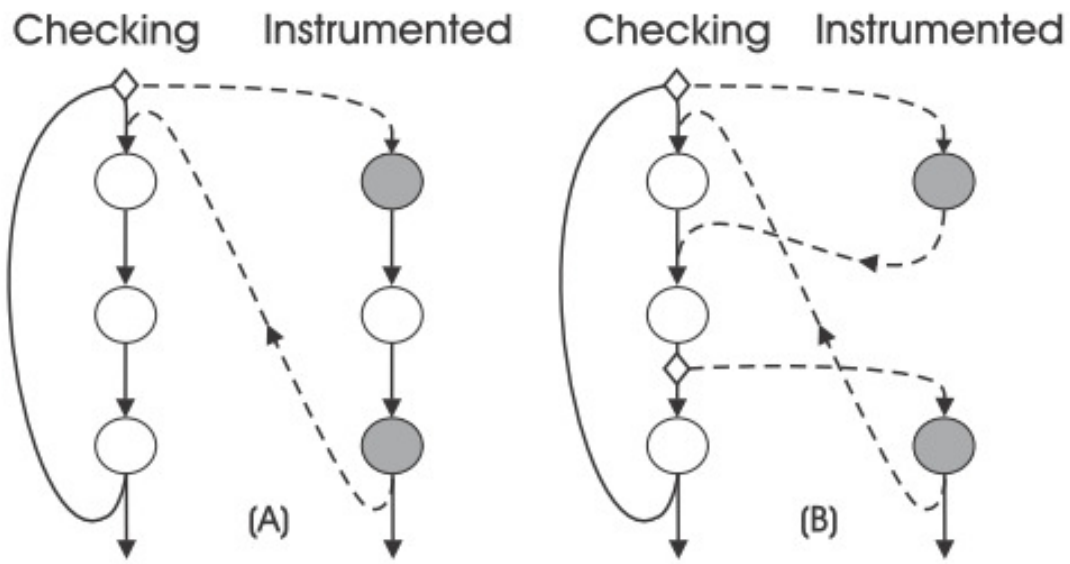
- What if the resetValue is equal to the number of loop iterations?

Compiler-inserted counters

Benchmark	Time-based (%)	Counter-based (%)
201_compress	88	98
202_jess	91	95
209_db	66	95
213_javac	59	73
222_mpegaudio	69	95
227_mtrt	51	67
228_jack	45	94
opt-compiler	58	65
pBOB	75	87
Volano	27	71
Average	63	84

Space Optimizations

- Keeping a second (instrumented) copy of the code can be expensive and is often unnecessary
- Non-instructed nodes do not have to be duplicated:

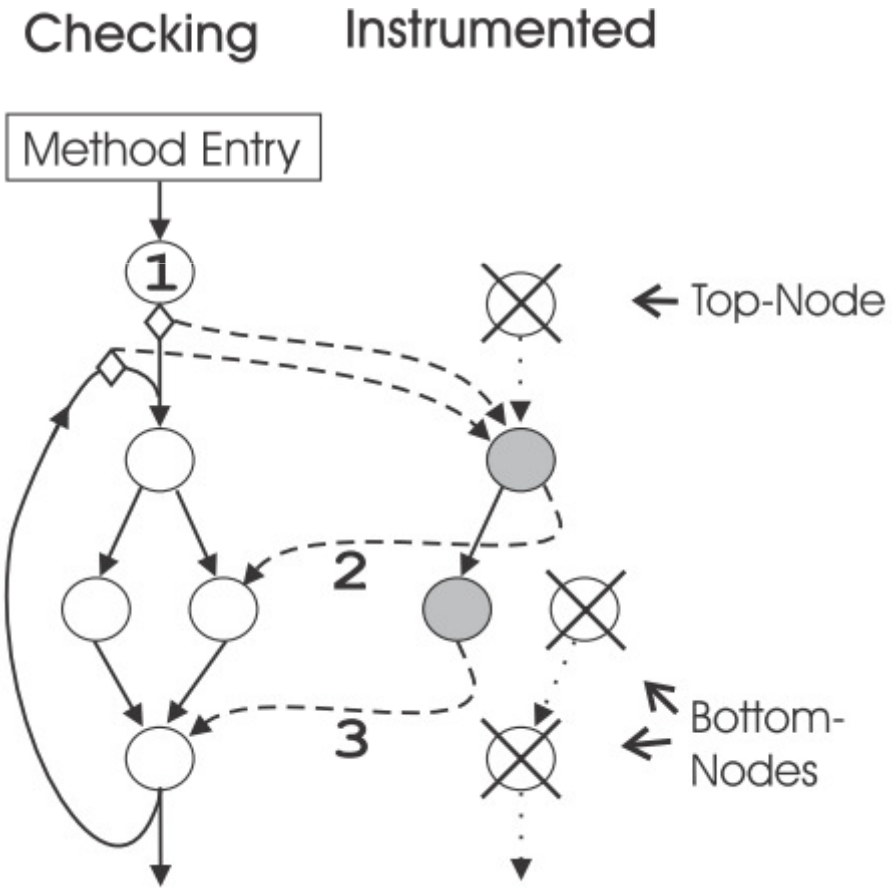


Space Optimizations

- Violates Invariant 1:
  - Number of checks in the code is not influenced by the instrumentation being executed

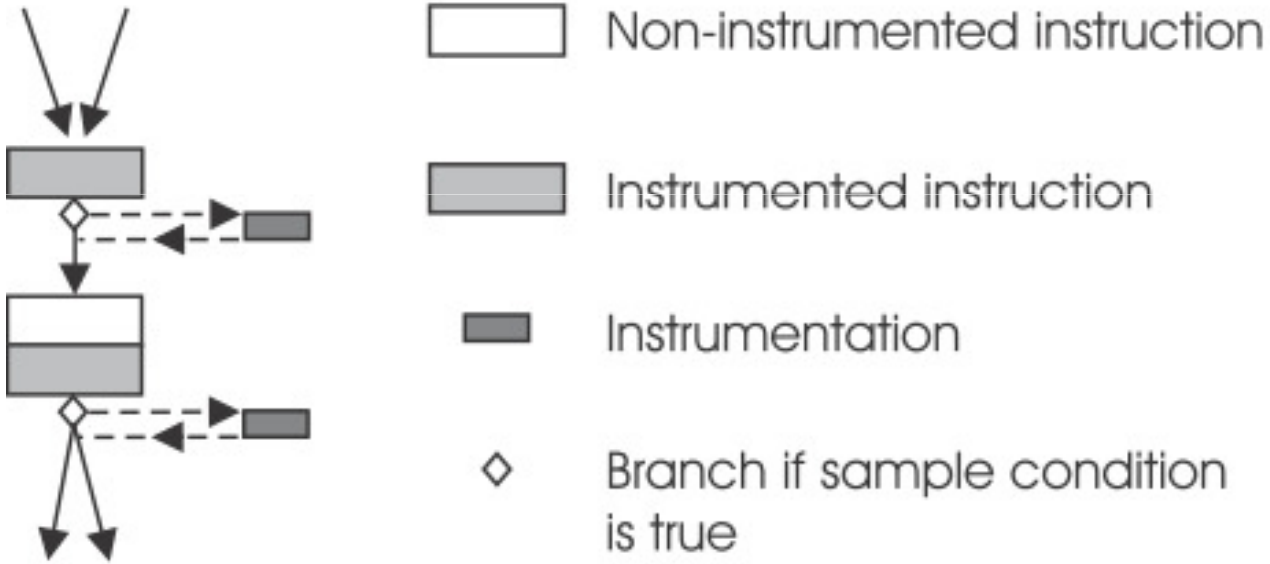
Space Optimization

- Variation-1 (maintains invariant)



Space Optimization

- Variation-2 (violates invariant)



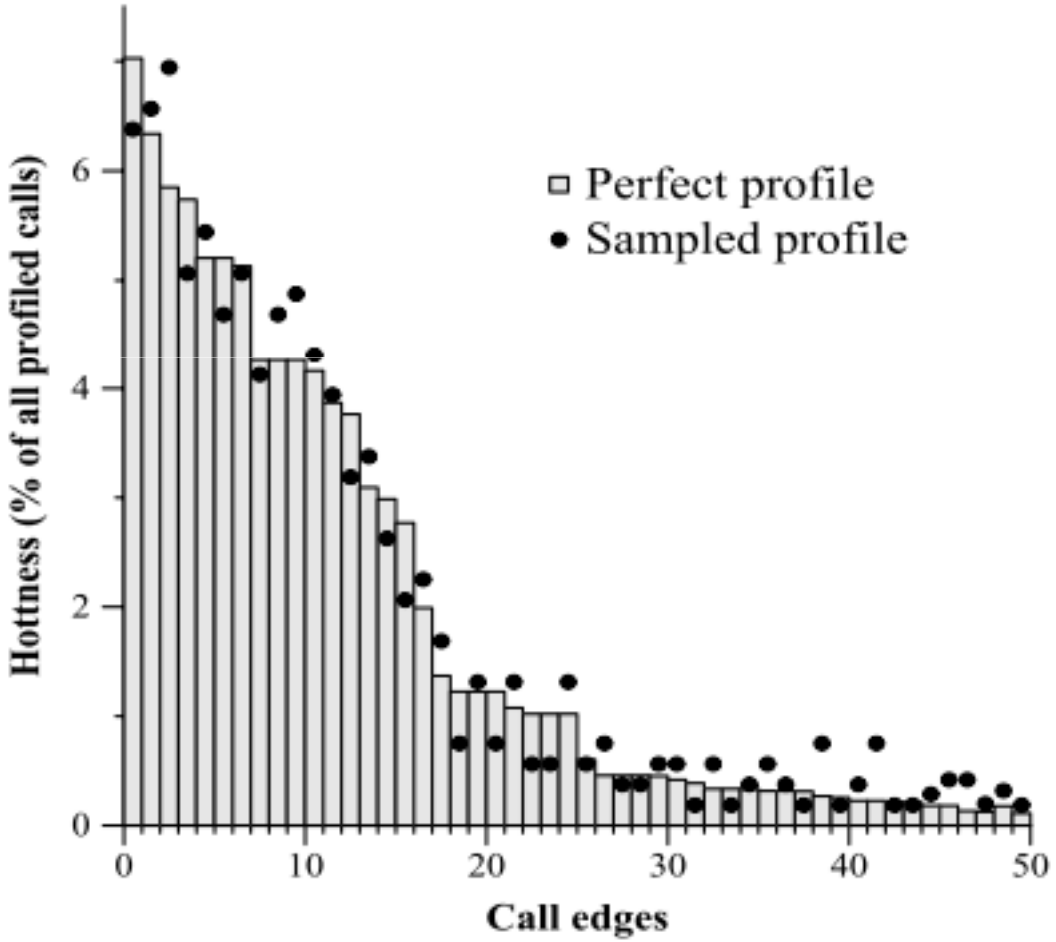
Evaluation

- Using Jalapeno VM
- Call-Edge instrumentation
- Field-Access instrumentation

Evaluation

Benchmarks	Variation-0				Variation-2	
	All Checks (%)	Backedges Only (%)	Method entry Only (%)	Maximum space Overhead (KB)	Call-edge (%)	Field-access (%)
201_compress	5.9	3.1	-2.5	40	-2.5	102.1
202_jess	6.3	4.2	2.3	75	2.3	55.7
209_db	*	*	*	45	*	3.5
213_javac	1.3	0.6	2.1	128	2.1	14.2
222_mpegaudio	8.4	7.9	0.9	157	0.9	52.7
227_mtrt	0.9	0.6	*	57	*	60.1
228_jack	6.1	4.3	*	87	*	43.2
opt-compiler	2.6	1.6	1.5	103	1.5	48.3
pBOB	2.4	*	2.7	300	2.7	39.1
Volano	2.7	0.6	1.4	36	1.4	1.4
Average	3.6	2.3	0.8	84	0.8	41.2

Evaluation


Perfect profile
Sampled profile
Hottness (% of all profiled calls)
0
2
4
6
0
10
20
30
40
50
Call edges

Summary

- Arnold-Ryder Framework gives good results while drastically reducing the performance overhead
- As seen in other papers, there are some drawbacks which can be addressed by modifying the framework