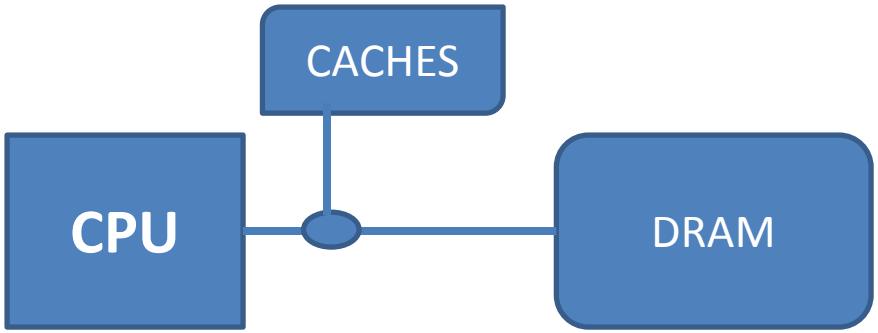
Static Analysis Driven Cache Performance Testing

Abhijeet Banerjee **Sudipta Chattopadhyay** **Abhik Roychoudhury**

Caches: Why are they needed ?

Caches are used to bridge the performance gap between CPU and DRAM



Caches have a significant impact on performance

Impact on performance due to Caches

Cache Hit occurs when a memory block accessed by the processor is in the cache …

Otherwise it is a **Cache Miss**

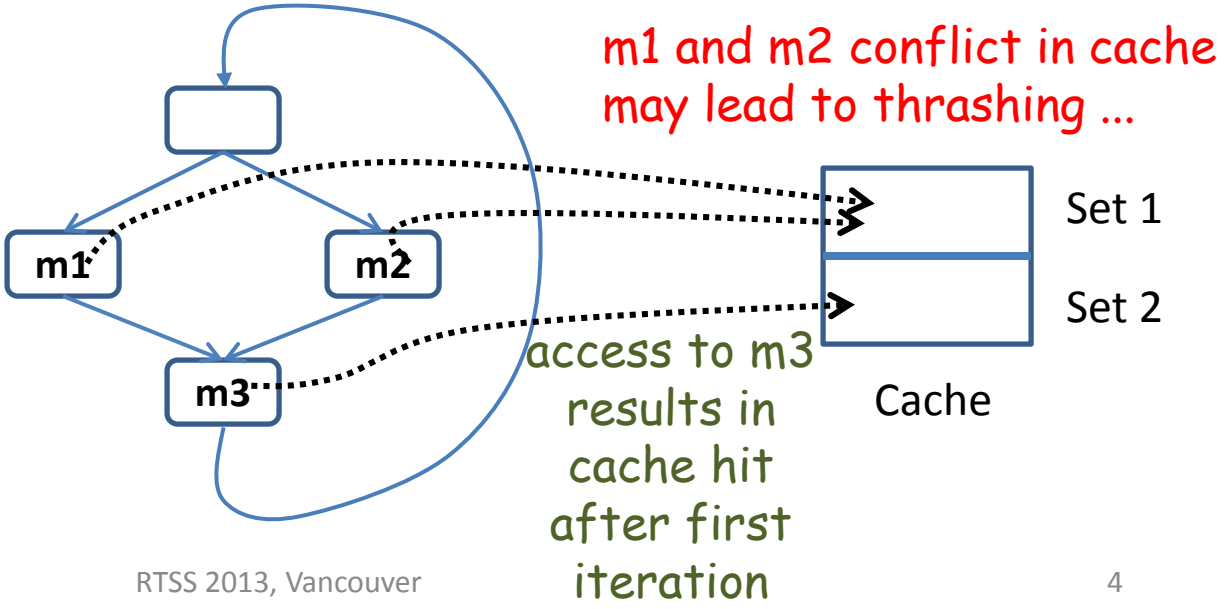
Cache Misses are bad because they negatively impact performance

What kind of memory access patterns leads to substantial cache misses ?

Cache Thrashing

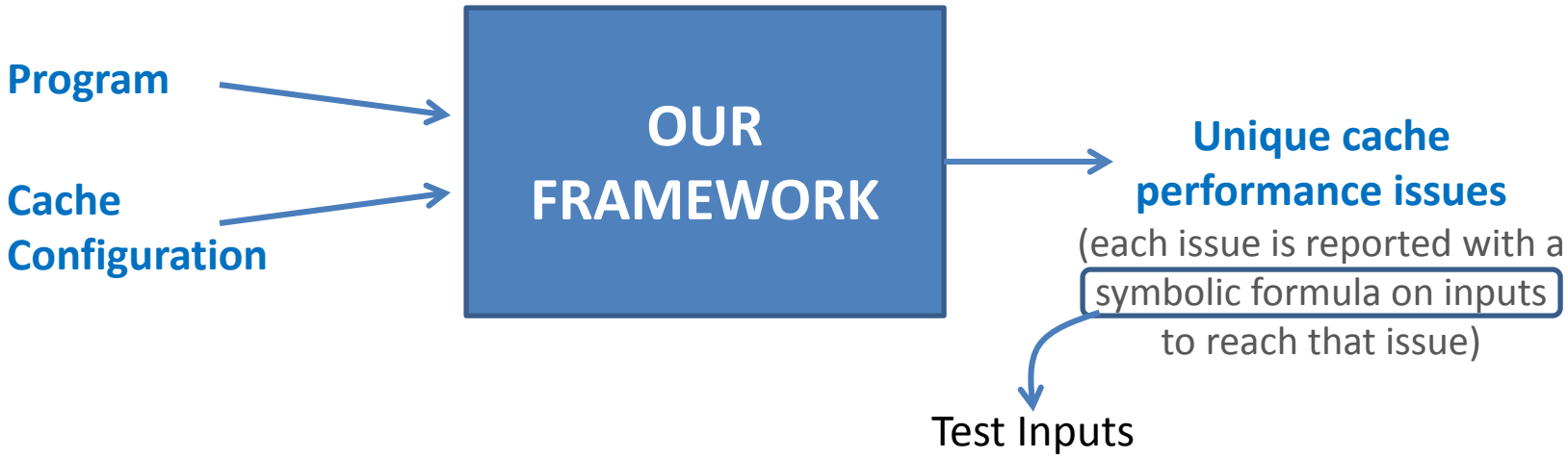
Cache Thrashing occurs when a frequently used cache block is replaced by another frequently used cache block … as a result lots of Cache Misses

```
While(true){
  if(x > 5){
    // m1 accessed
  }else{
    // m2 accessed
  }
 // m3 accessed
}
```

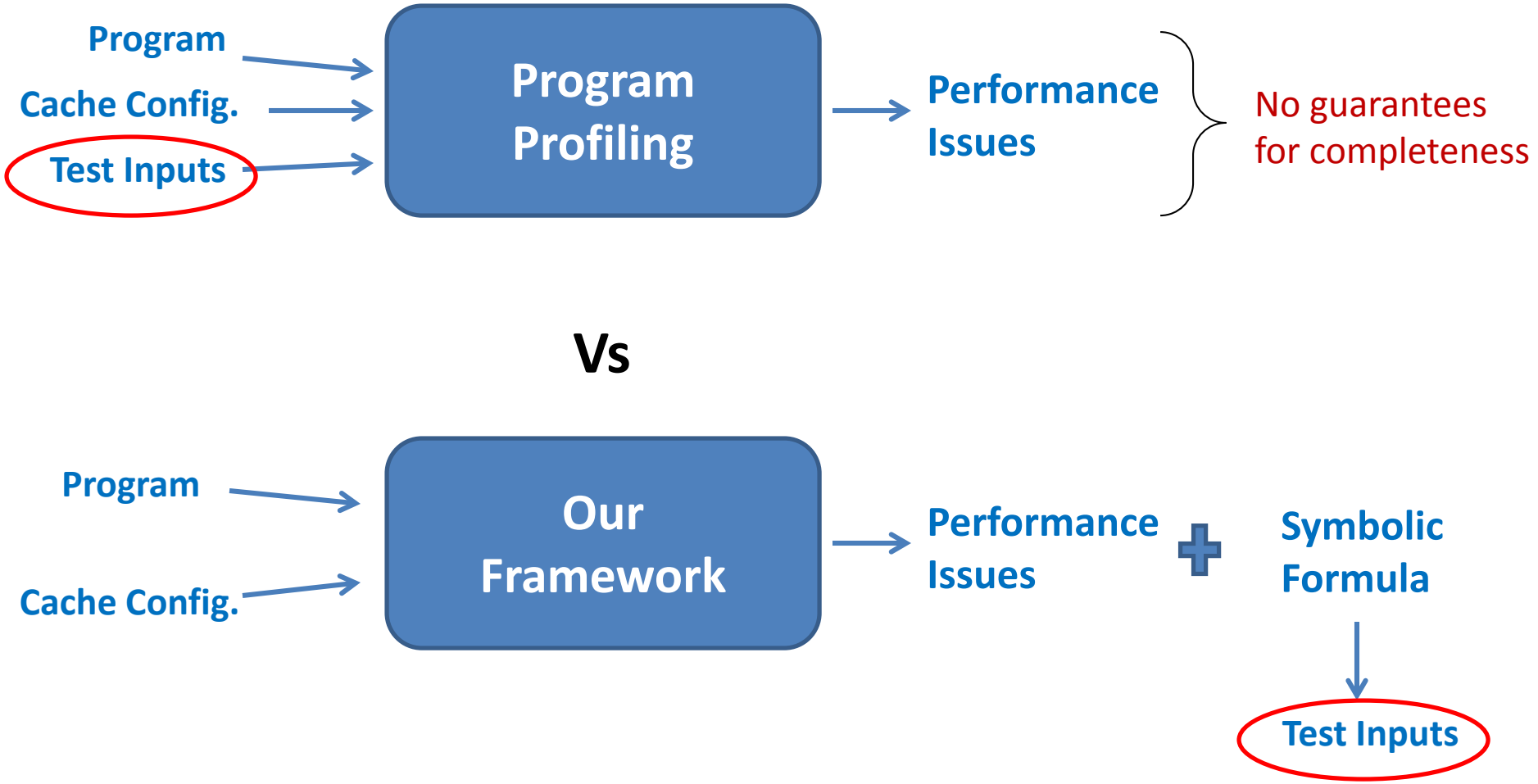


Objective of our work

To develop a test generation framework which aims to report all possible cache performance issues that may exist in some program execution.

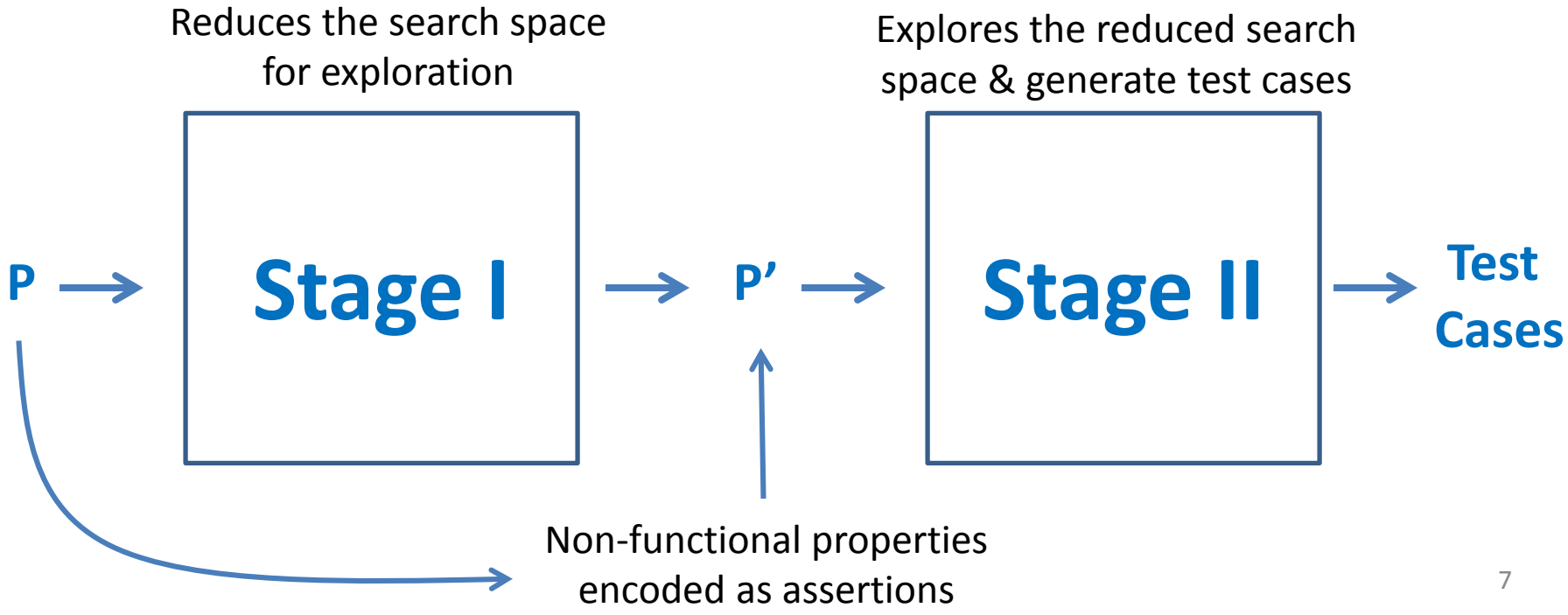


It is not a profiling technique !

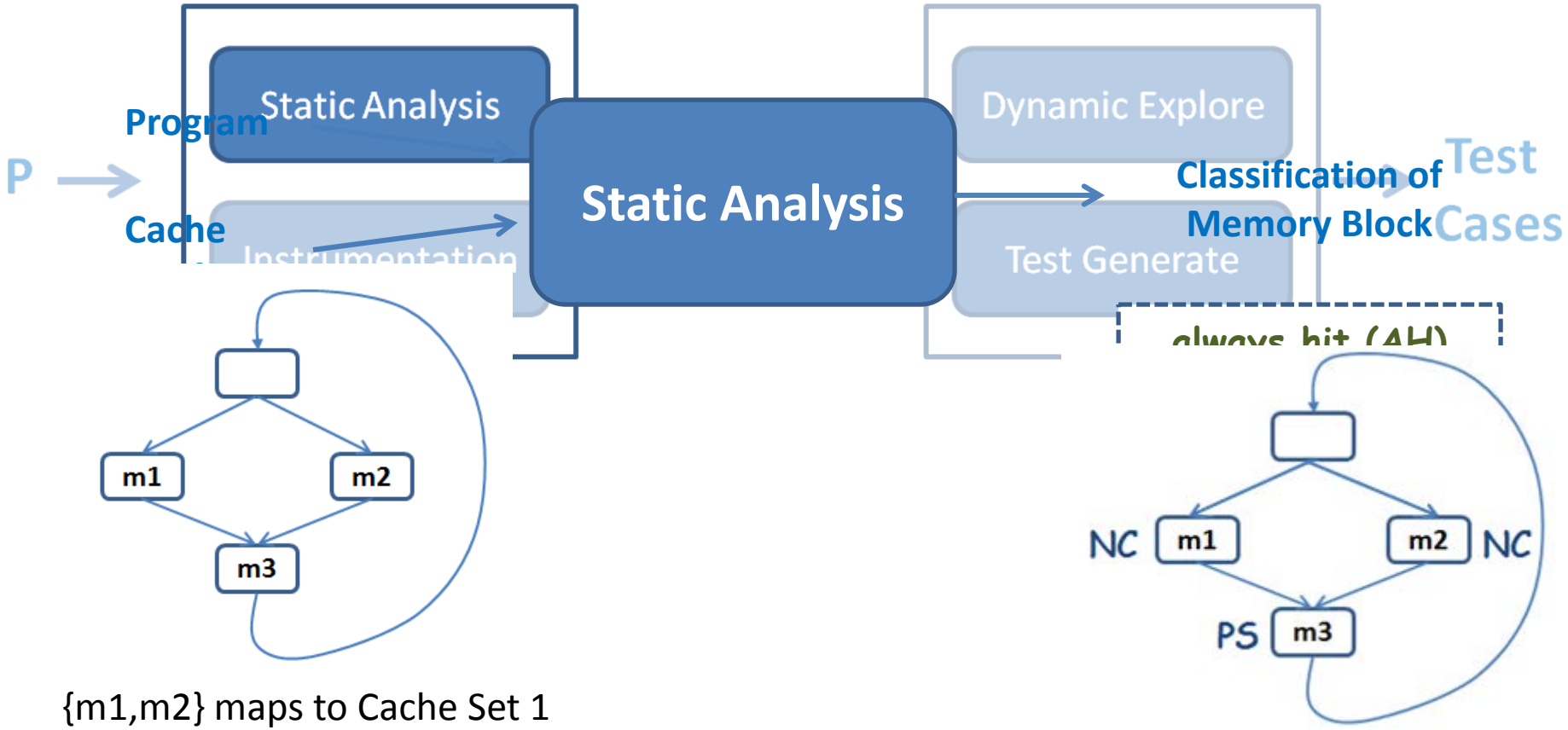


Key Idea

We reduce the problem of testing cache performance to an equivalent functionality testing problem



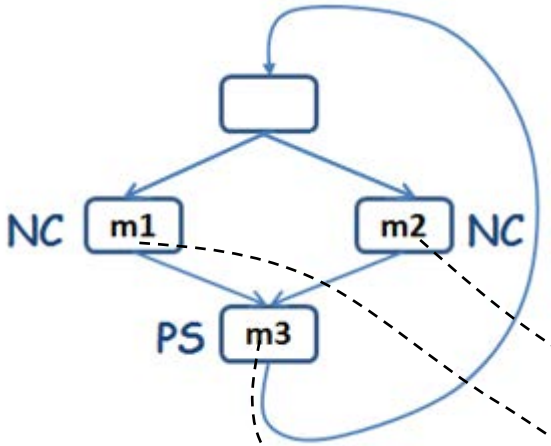
Static Analysis



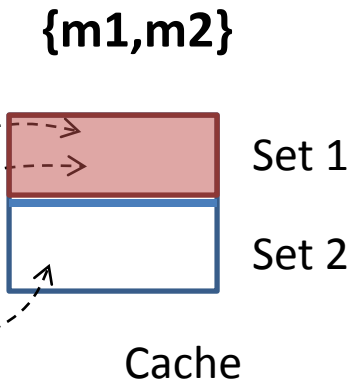
{m1,m2} maps to Cache Set 1
{m3} maps to Cache Set 2

Identifying Thrashing Scenarios

Classification of Memory Block → **Extract memory blocks potentially involved in Cache Thrashing** → Set of Cache Thrashing Scenarios

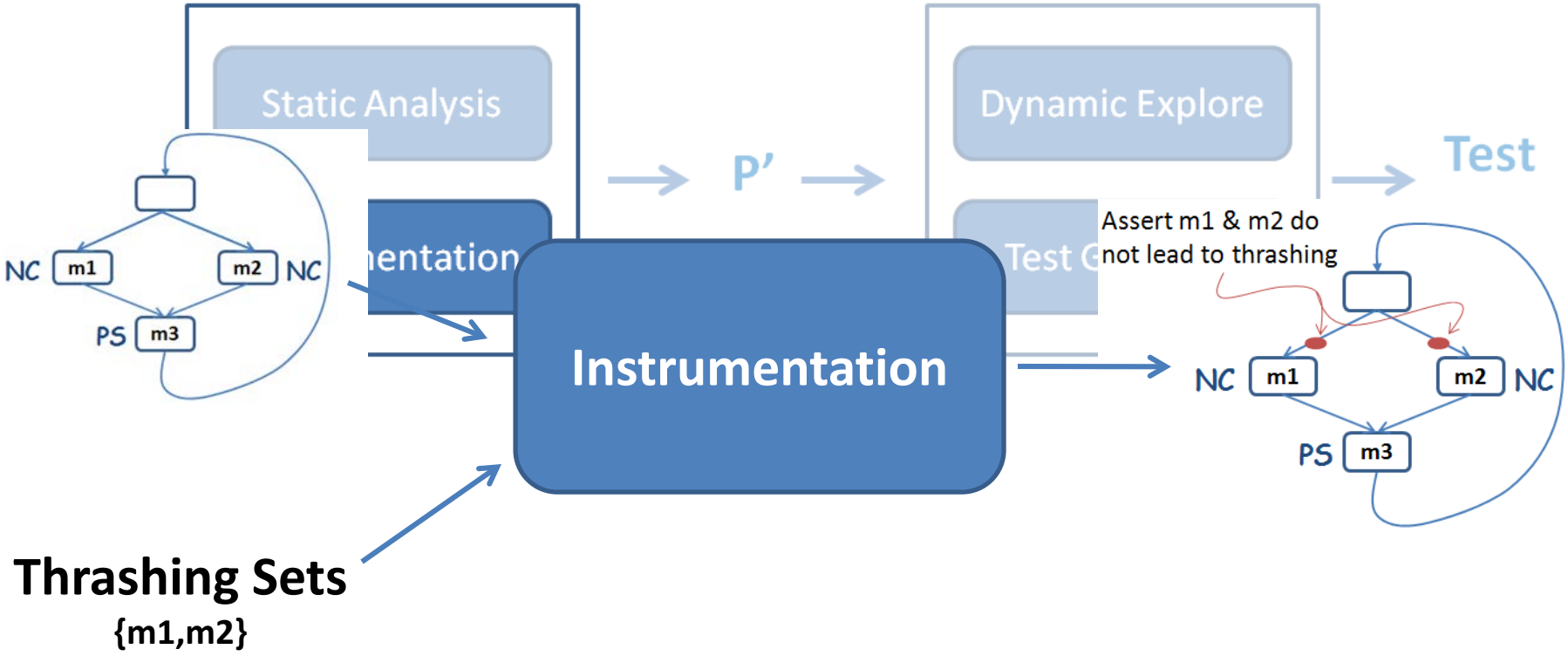


Extract
always miss (AM)
not classified (NC)
For each cache set

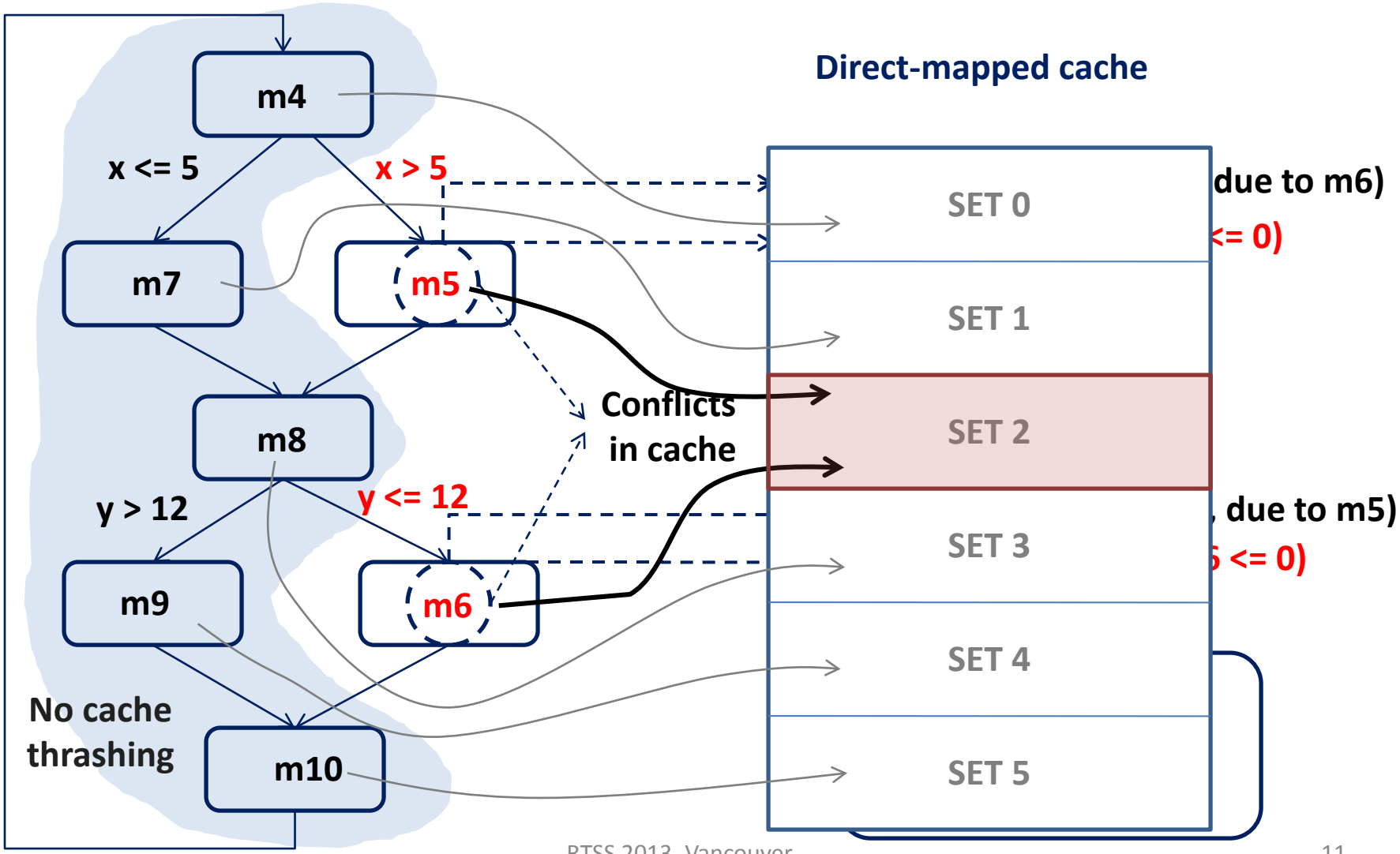


Instrumentation

Encode each thrashing scenario as an assertion at appropriate program location

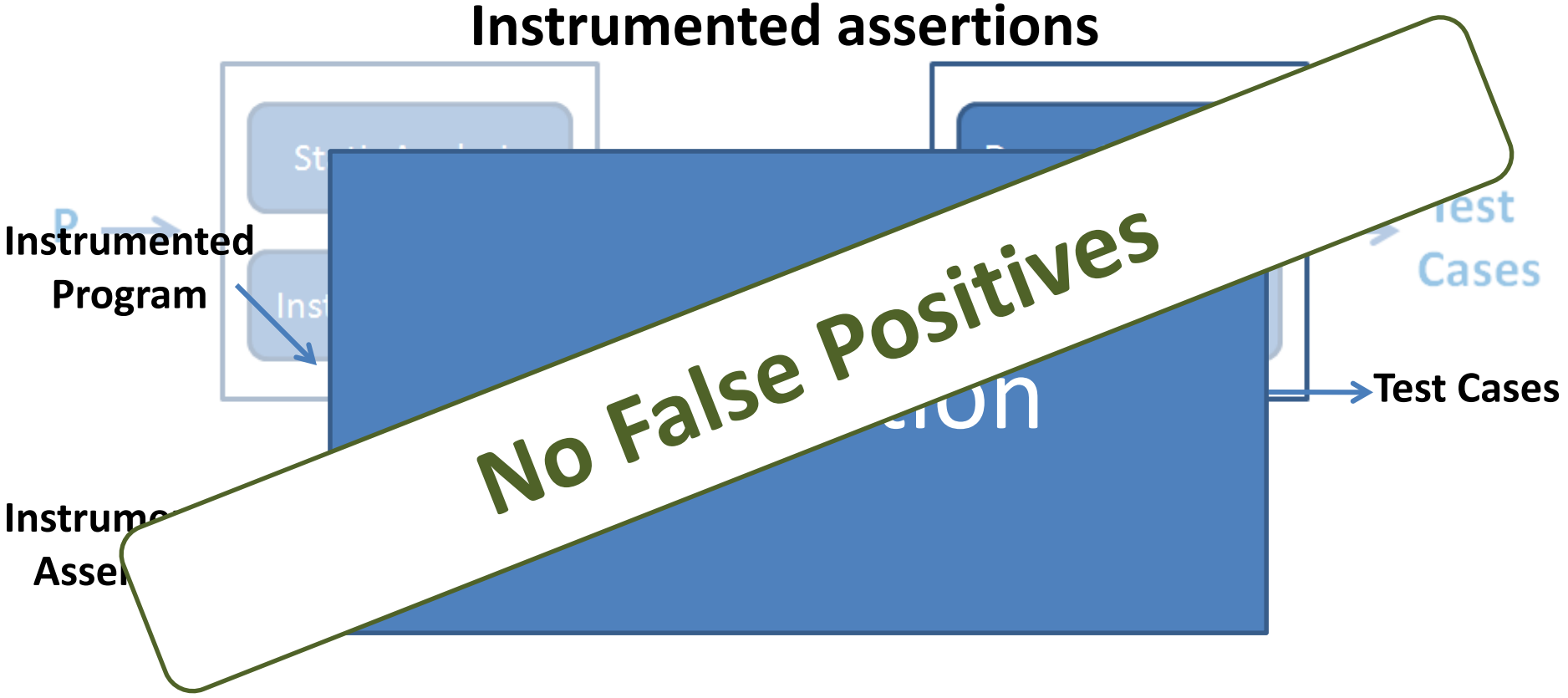


Generating Assertions

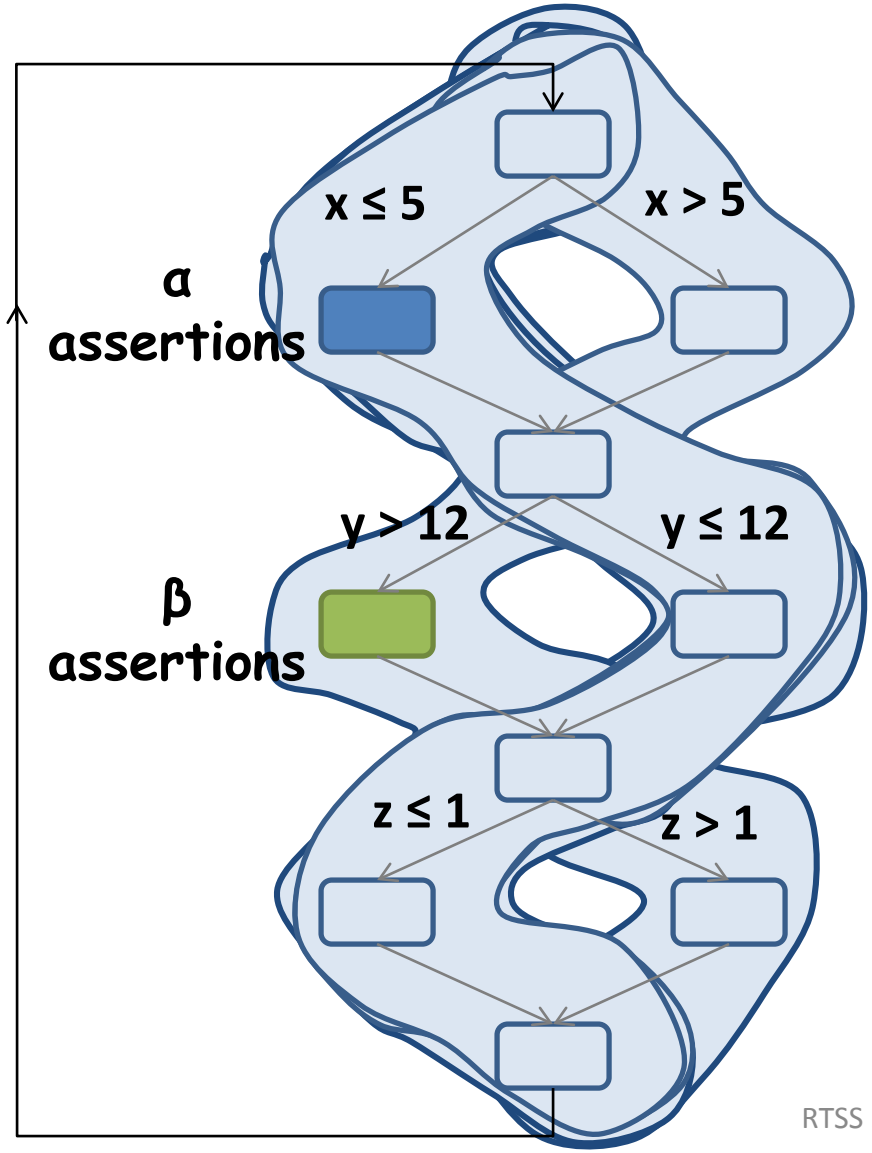


Dynamic Exploration

Exploration is performed to check the violation of Instrumented assertions



Exploration by Greedy Strategy

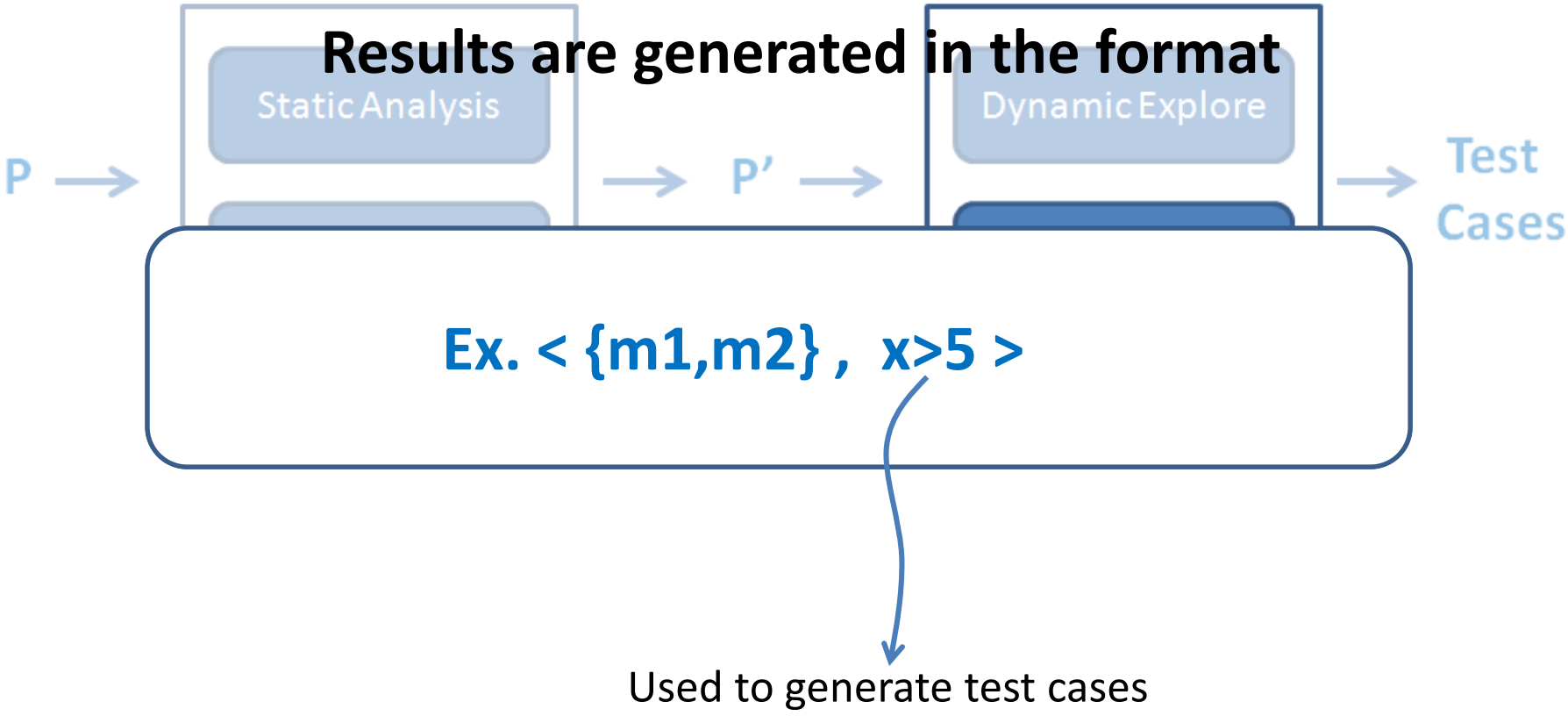


$(x \le 5) \wedge (y \le 12) \wedge (z \le 1)$

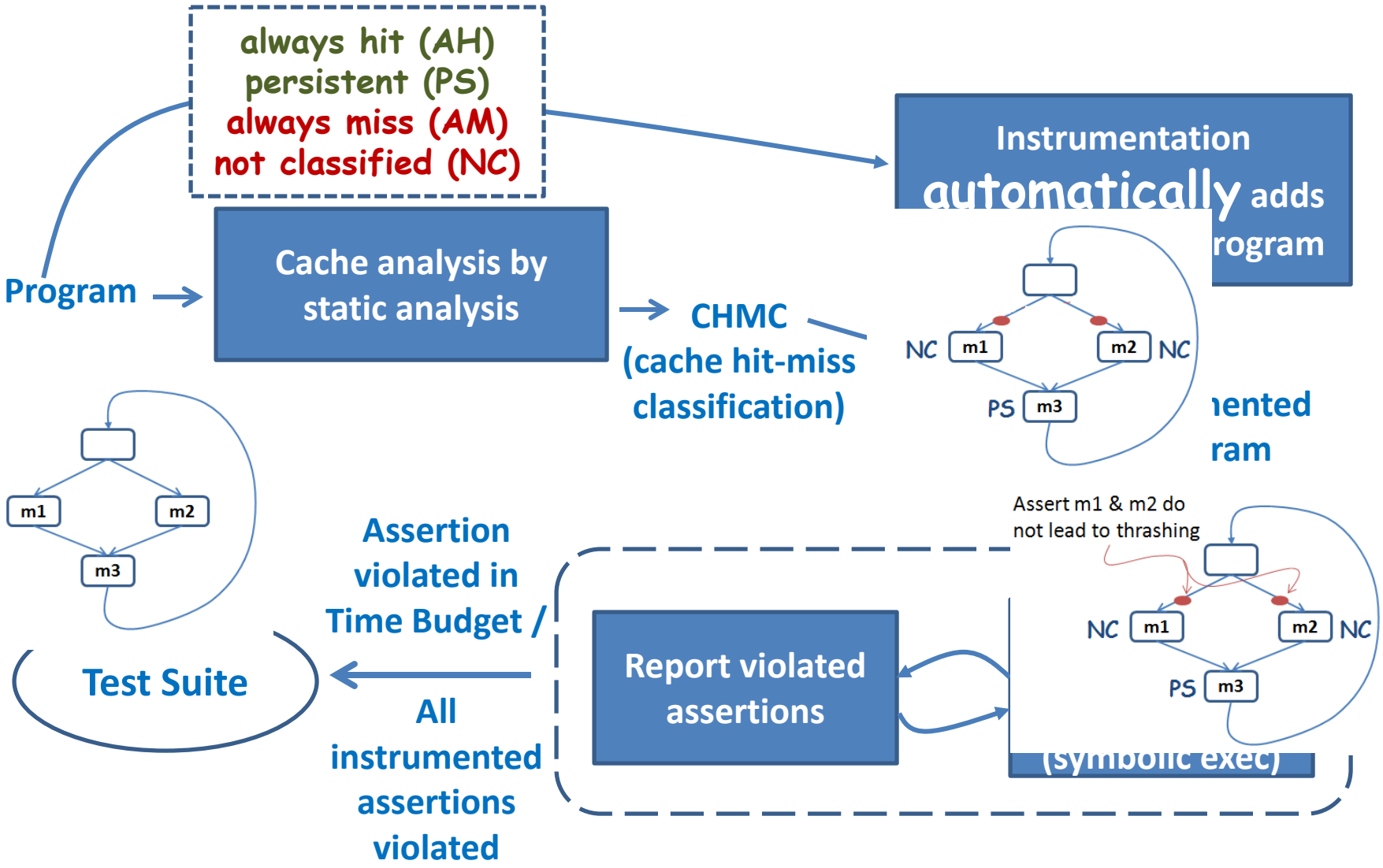
$\Rightarrow \alpha$ assertions checked

Exploration performed using the Control Dependency Graph CDG

Test Generation



Recap

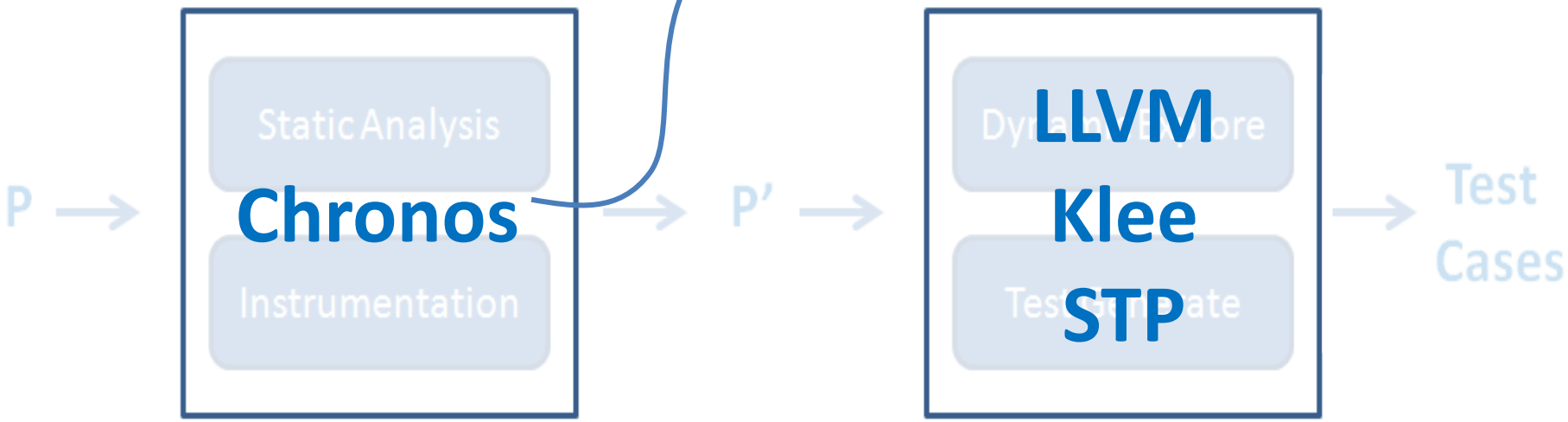

always hit (AH)
persistent (PS)
always miss (AM)
not classified (NC)
Instrumentation
automatically adds
rogram
Program
Cache analysis by
static analysis
CHMC
(cache hit-miss
classification)
NC
m1
m2
NC
PS
m3
ıented
ram
m1
m2
m3
Assertion
violated in
Time Budget /
All
instrumented
assertions
violated
Test Suite
Report violated
assertions
Assert m1 & m2 do
not lead to thrashing
NC
m1
m2
NC
PS
m3
(symbolic exec)

Experiments

- Instruction Caches
- Data Caches

RTS' 00 (Theiling et al)

RTAS' 11(Our work)



Evaluation

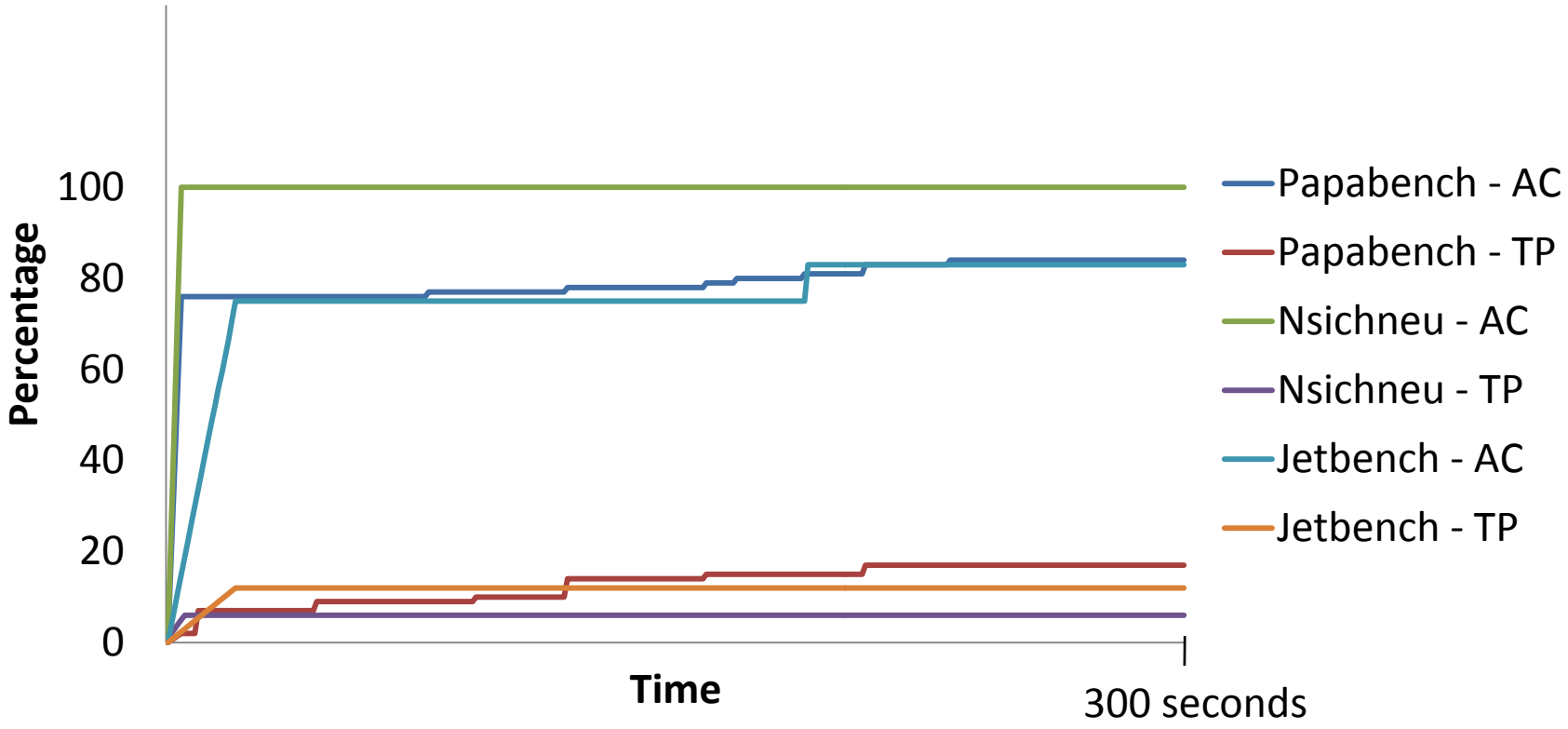
$$\textbf{Assertion Coverage (AC)} = \frac{\text{Unique assertions checked} * 100}{\text{Unique assertions instrumented}}$$

100 % coverage implies all unique assertions have been checked at least once

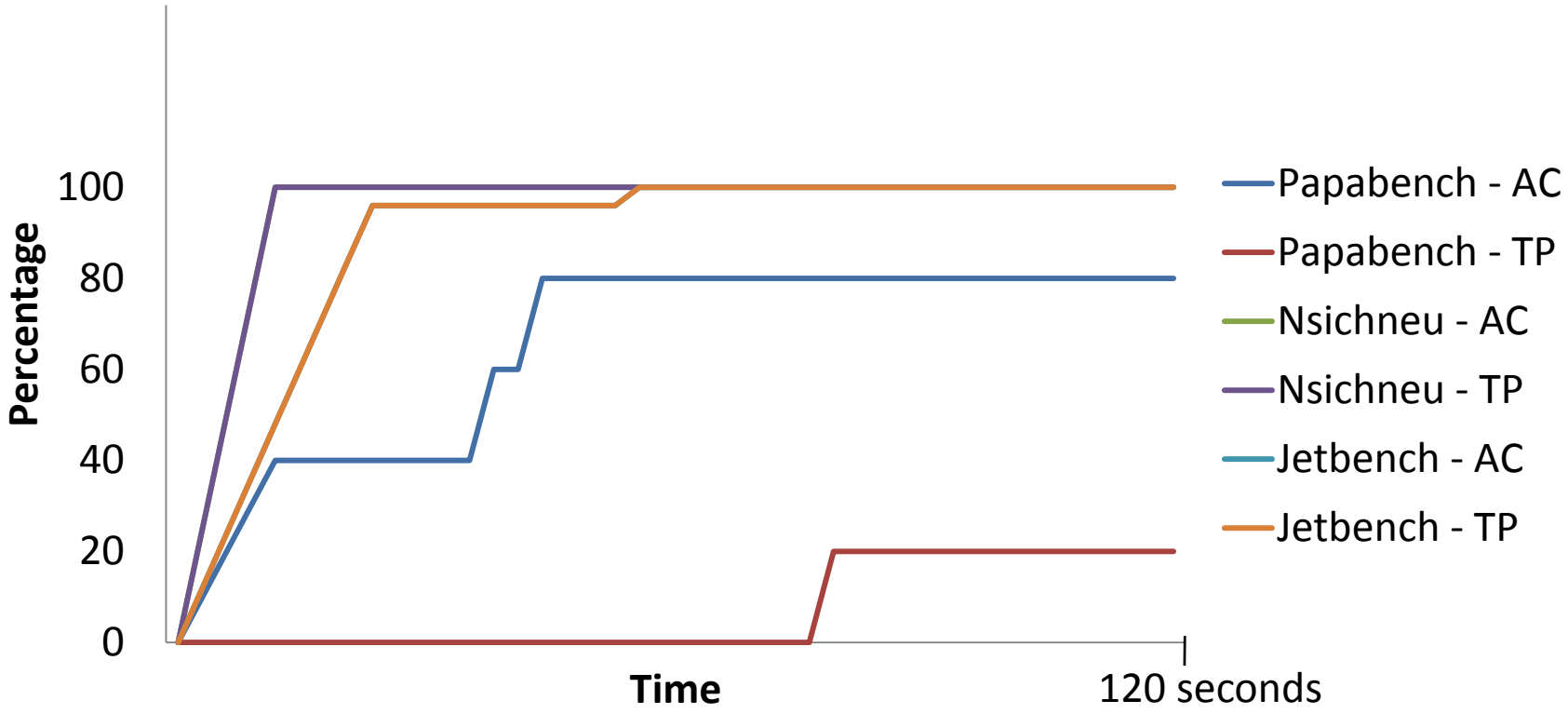
$$\textbf{Thrashing Potential (TP)} = \frac{\text{Unique assertions violated} * 100}{\text{Unique assertions instrumented}}$$

Gives an idea about the thrashing potential for a program, for a given cache configuration

Results – Instruction Caches



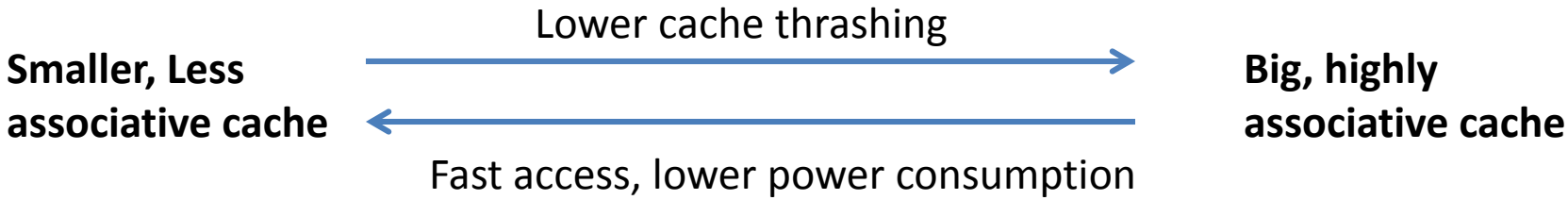
Results – Data Caches

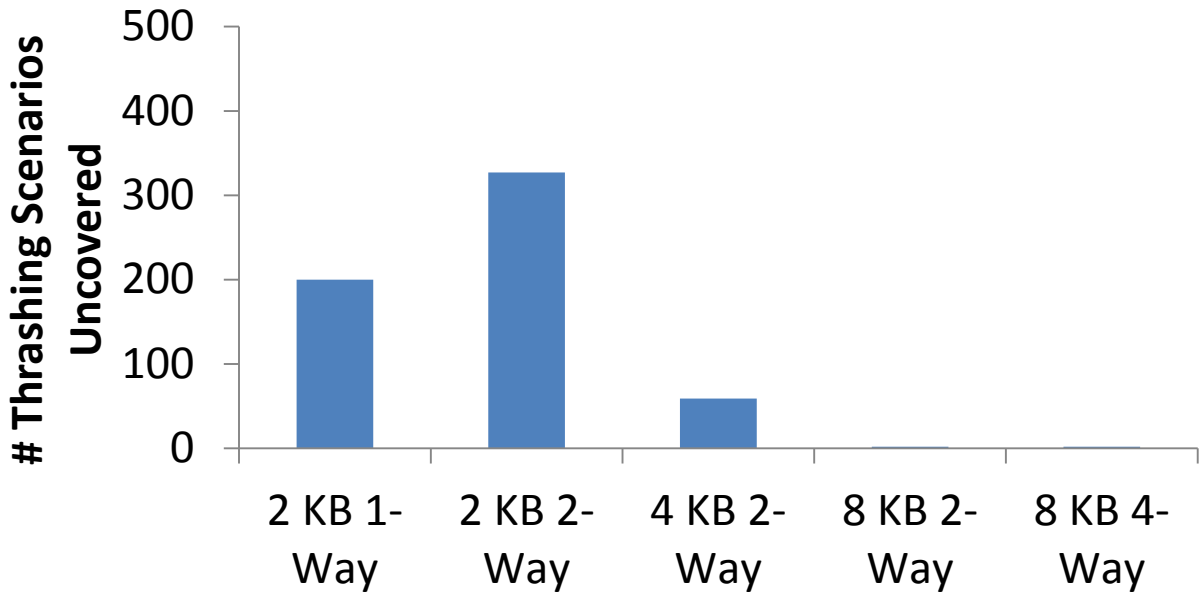


Observations

- Programs with lesser number of input dependent paths were explored faster
- For most experiments, only a small fraction of instrumented assertions were violated
- Most assertions were explored early. Shows the goodness of directed search

Application: Design space exploration

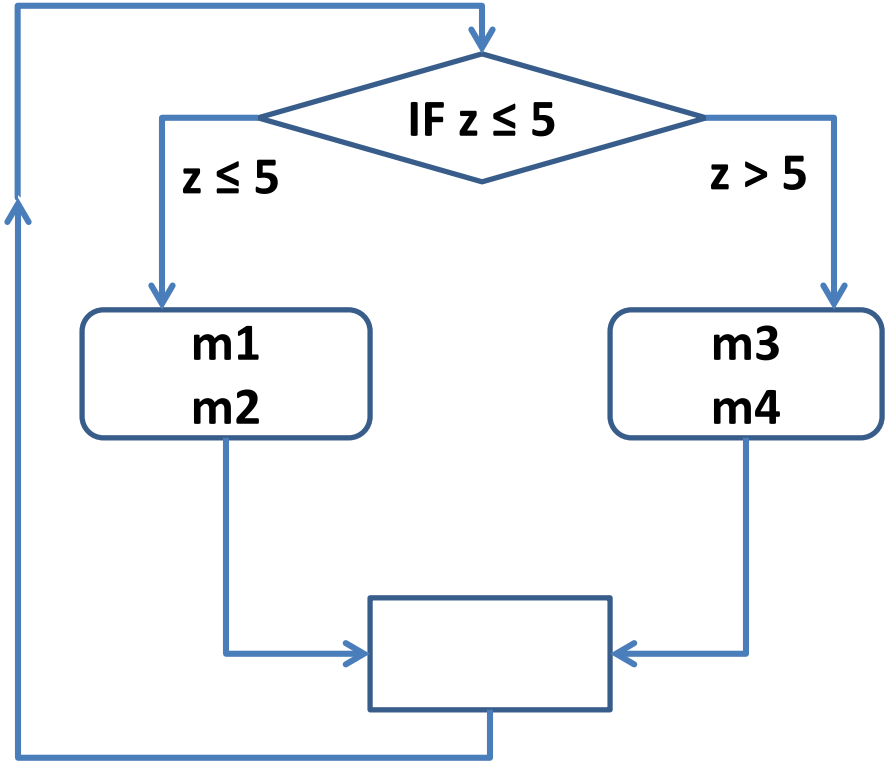




Number of cache thrashing scenarios discovered for papabench, for various cache configurations

Application: Performance Optimization

Can be used to devise improved Cache Locking Techniques



Direct mapped cache

m1, m2, m3, m4 conflict in cache

Traditional Cache Locking

Either lock m1 OR m2 OR m3 OR m4

Conditional Cache Locking

Lock m1 OR m2 IF z ≤ 5

Lock m3 OR m4 IF z > 5

Related Work

Existing Work	Our Work
Testing Functionality PLDI 2005, OSDI 2008	**Testing Performance**
Profiling Not Complete	**Complete**
Partitioning I/P Space LCTES 2013 Requires manual effort May have false positives	**Automated** **No False Positives**

Conclusion

- A **test generation framework** that stresses the cache performance of a program

- Key novelty is in the **systematic combination of static analysis and dynamic test generation** via a set of instrumented assertions

- Applications in **Design Space Exploration** and **Performance Optimization**