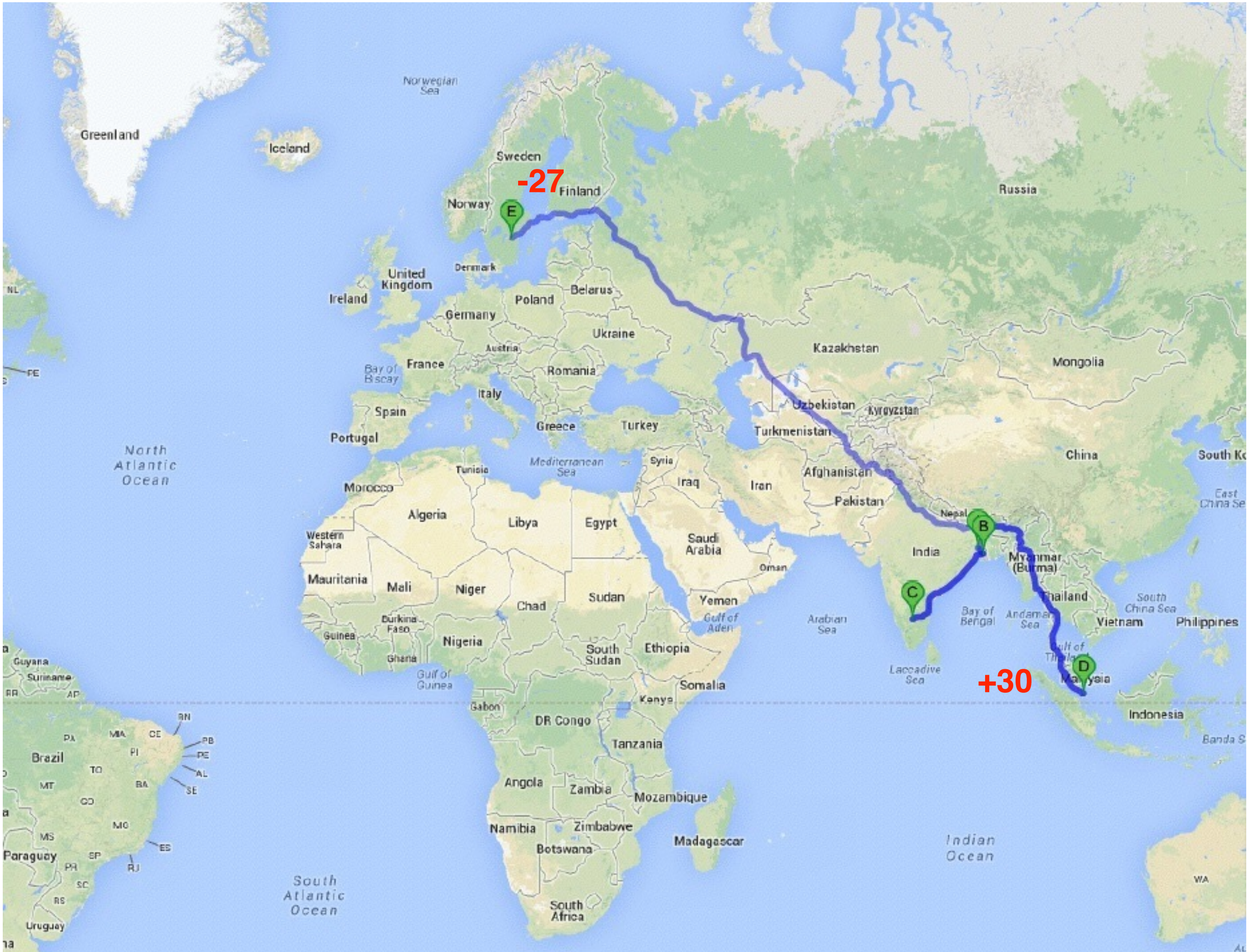
On Testing Non-functional Software Properties

Sudipta Chattopadhyay
Linköping University, Sweden

Joint work with Abhijeet Banerjee, Lee Kee Chong and Abhik Roychoudhury
National University of Singapore

Ahmed Rezine and Ke Jiang
Linköping University, Sweden


-27
+30
E
B
C
D
Greenland
Iceland
Norwegian Sea
Sweden
Finland
Norway
Russia
Denmark
United Kingdom
Ireland
Belarus
Poland
Germany
Ukraine
Austria
France
Bay of Biscay
Romania
Kazakhstan
Mongolia
Italy
Spain
Portugal
Greece
Turkey
Uzbekistan
Kyrgyzstan
Turkmenistan
China
North Atlantic Ocean
Mediterranean Sea
Syria
Iraq
Iran
Afghanistan
Pakistan
Nepal
India
Tunisia
Morocco
Algeria
Libya
Egypt
Western Sahara
Saudi Arabia
Oman
Yemen
Gulf of Aden
Arabian Sea
Mauritania
Mali
Niger
Chad
Sudan
Burkina Faso
Guinea
Nigeria
Ghana
Gulf of Guinea
South Sudan
Ethiopia
Somalia
Kenya
Gabon
DR Congo
Tanzania
Angola
Zambia
Mozambique
Namibia
Zimbabwe
Botswana
South Africa
Madagascar
Laccadive Sea
Bay of Bengal
Andaman Sea
Myanmar (Burma)
Thailand
Vietnam
South China Sea
Philippines
Indonesia
Malaysia
Brazil
Paraguay
Uruguay
Guyana
Suriname
South Atlantic Ocean
Indian Ocean

Context

Developer

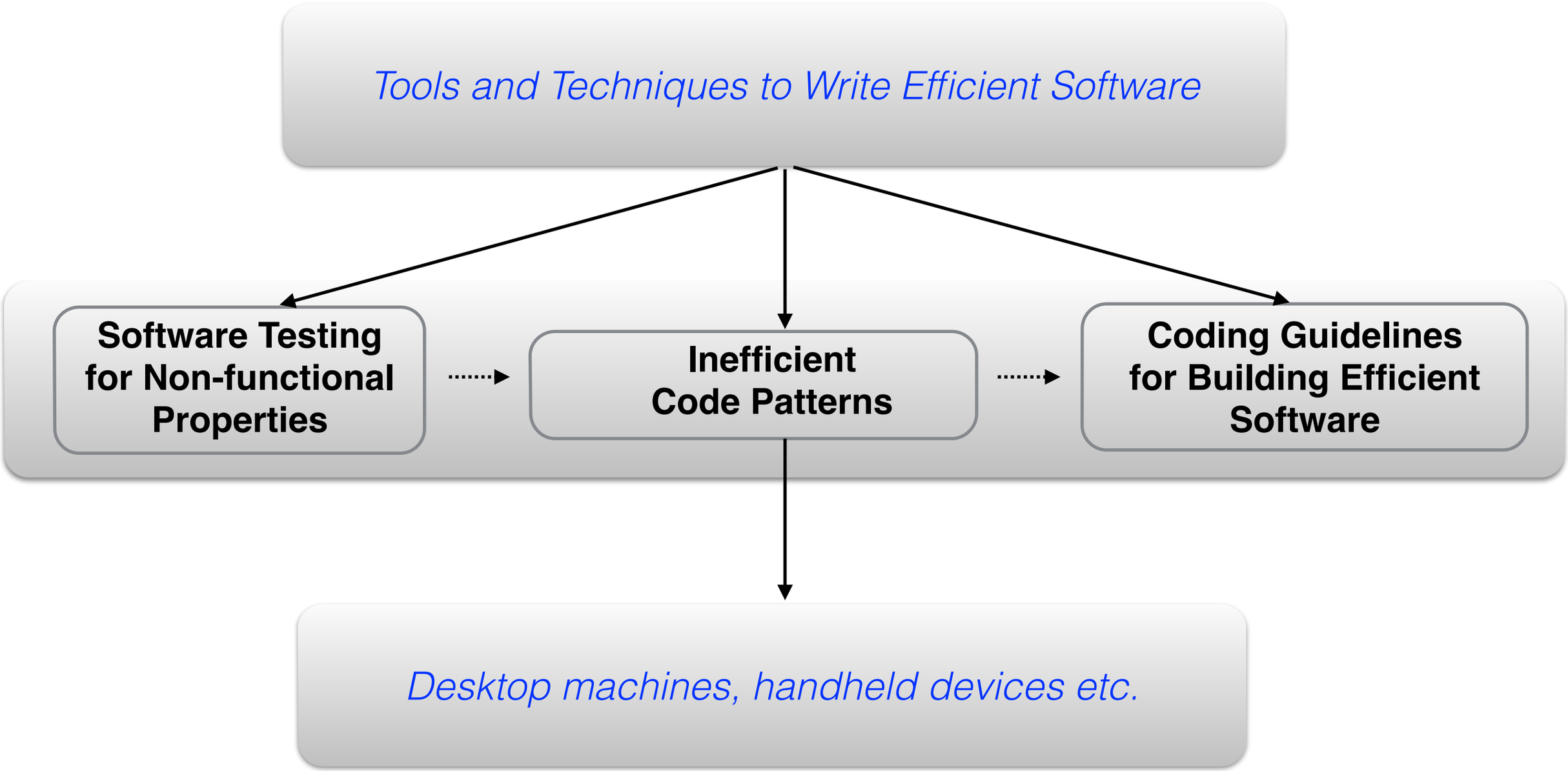
Write efficient software

Programming abstractions

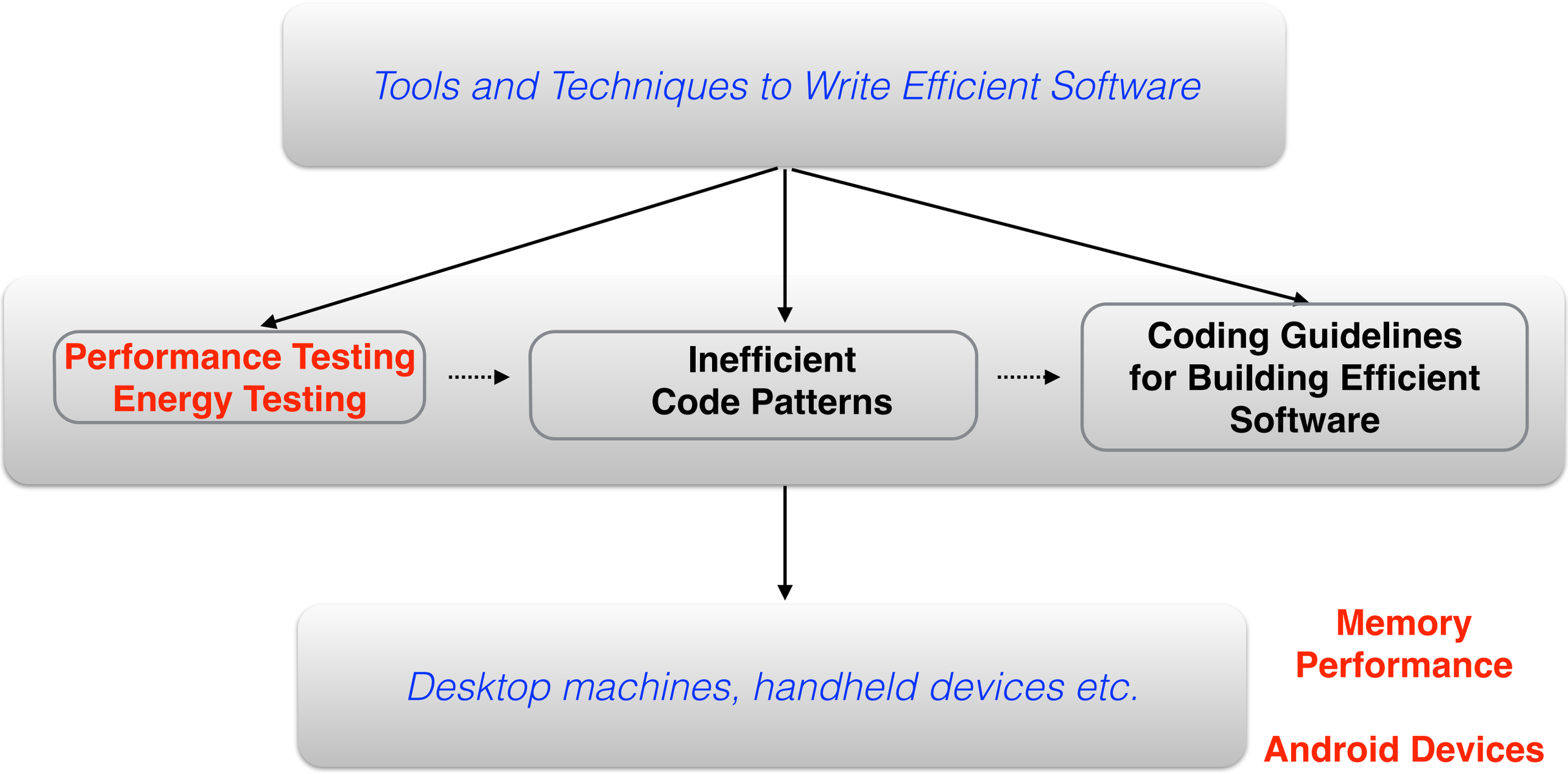
Tools and techniques

Desktops, handheld devices etc.

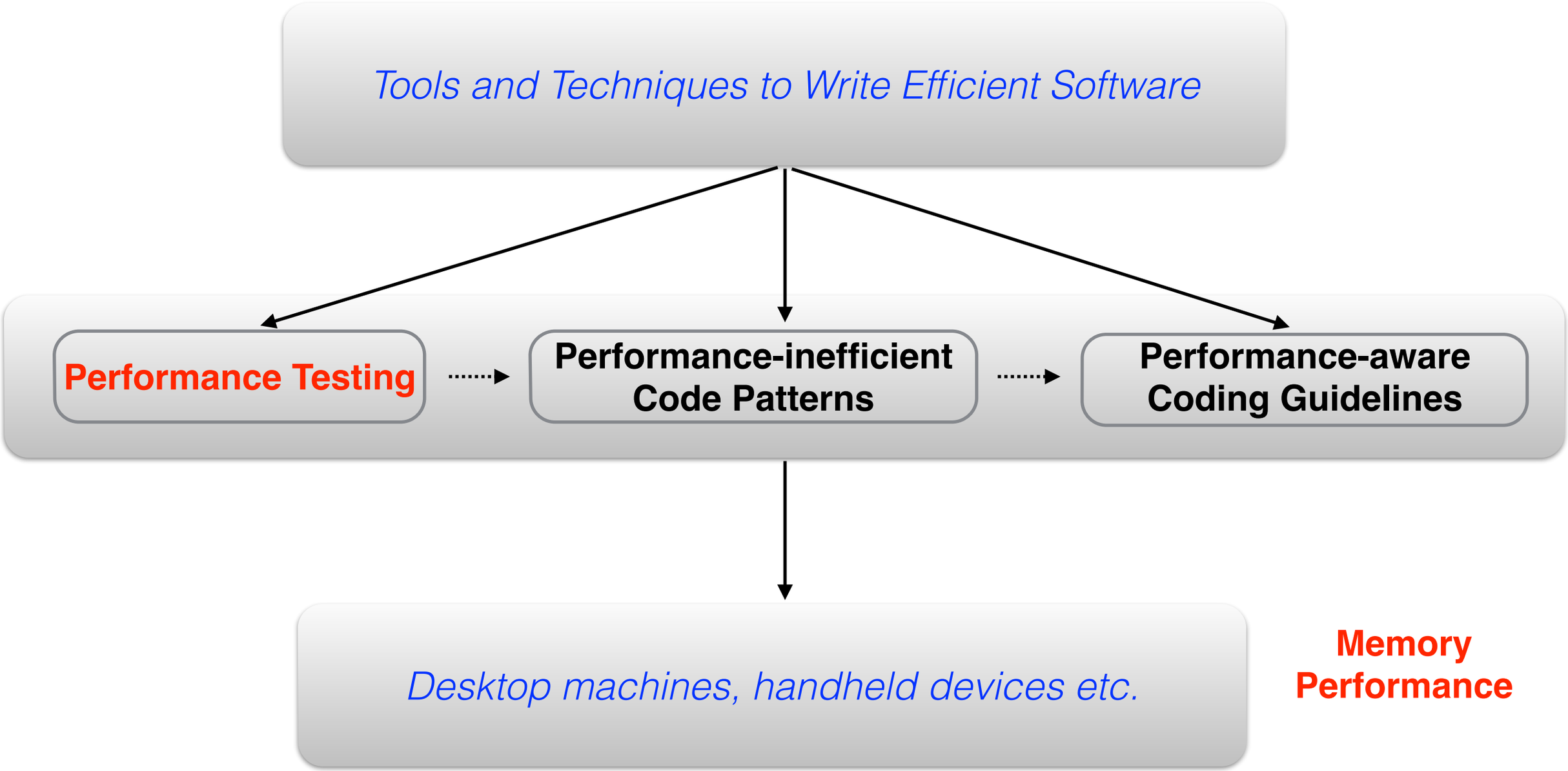
Overview


Tools and Techniques to Write Efficient Software
Software Testing for Non-functional Properties
Inefficient Code Patterns
Coding Guidelines for Building Efficient Software
Desktop machines, handheld devices etc.

Overview

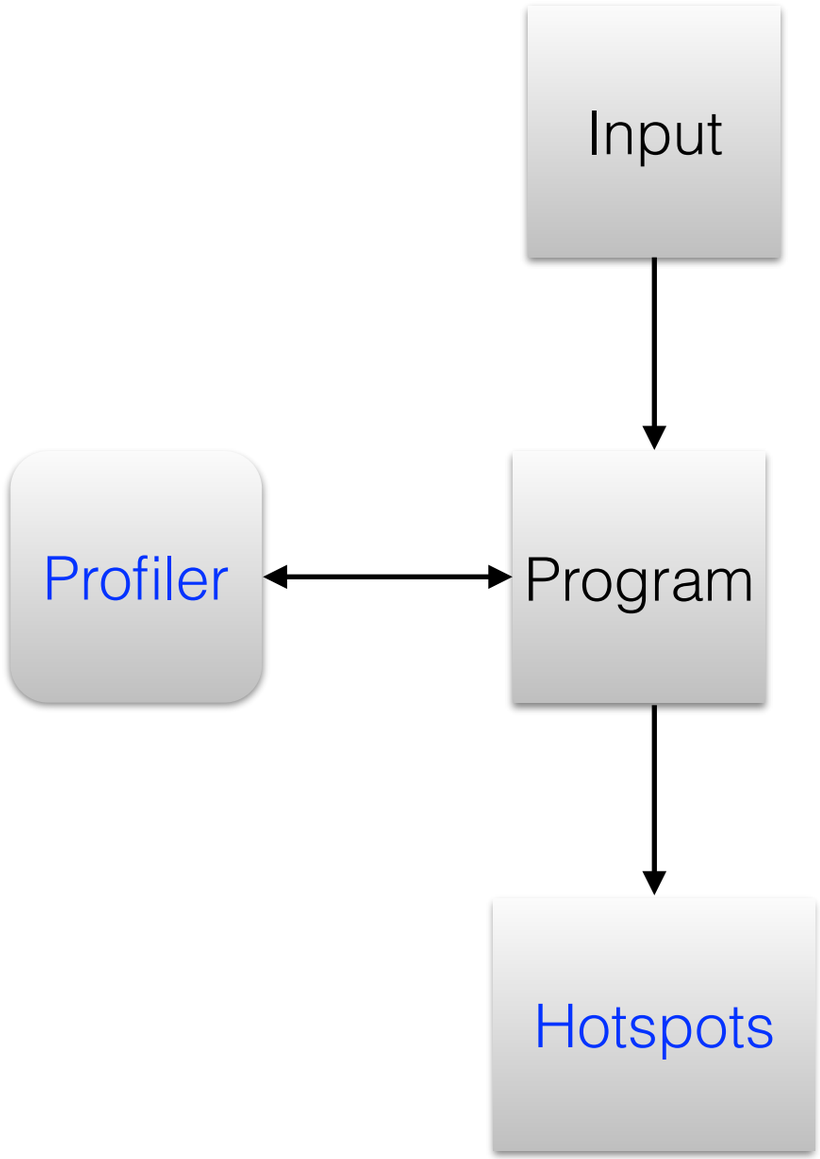

Tools and Techniques to Write Efficient Software
Performance Testing
Energy Testing
Inefficient
Code Patterns
Coding Guidelines
for Building Efficient
Software
Desktop machines, handheld devices etc.
Memory
Performance
Android Devices

Overview


Tools and Techniques to Write Efficient Software
Performance Testing
Performance-inefficient Code Patterns
Performance-aware Coding Guidelines
Desktop machines, handheld devices etc.
Memory
Performance

State-of-the-art in Detecting Performance Loss

Program profiling

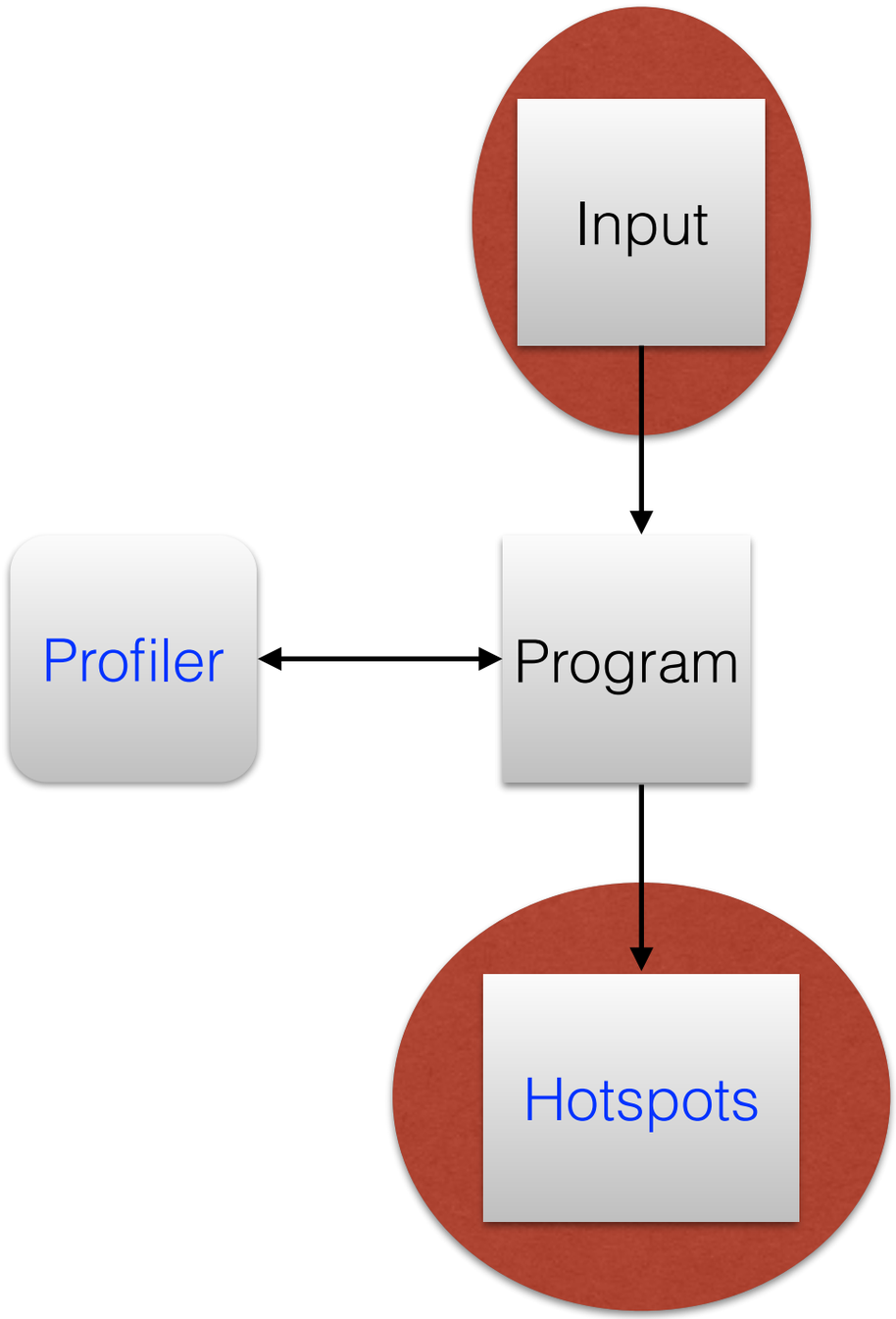


State-of-the-art in Detecting Performance Loss

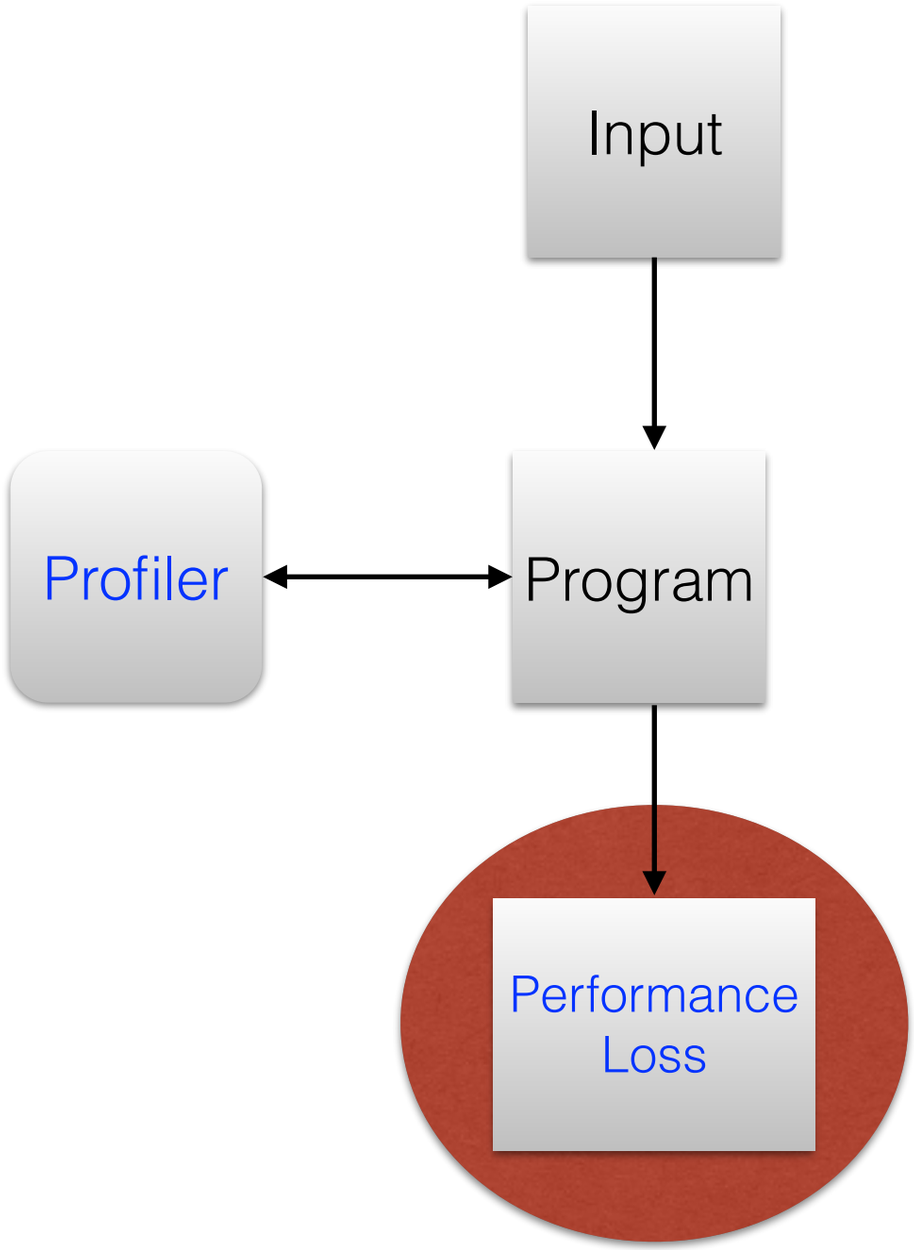
Program profiling

Program inputs that expose performance loss

Detecting performance loss

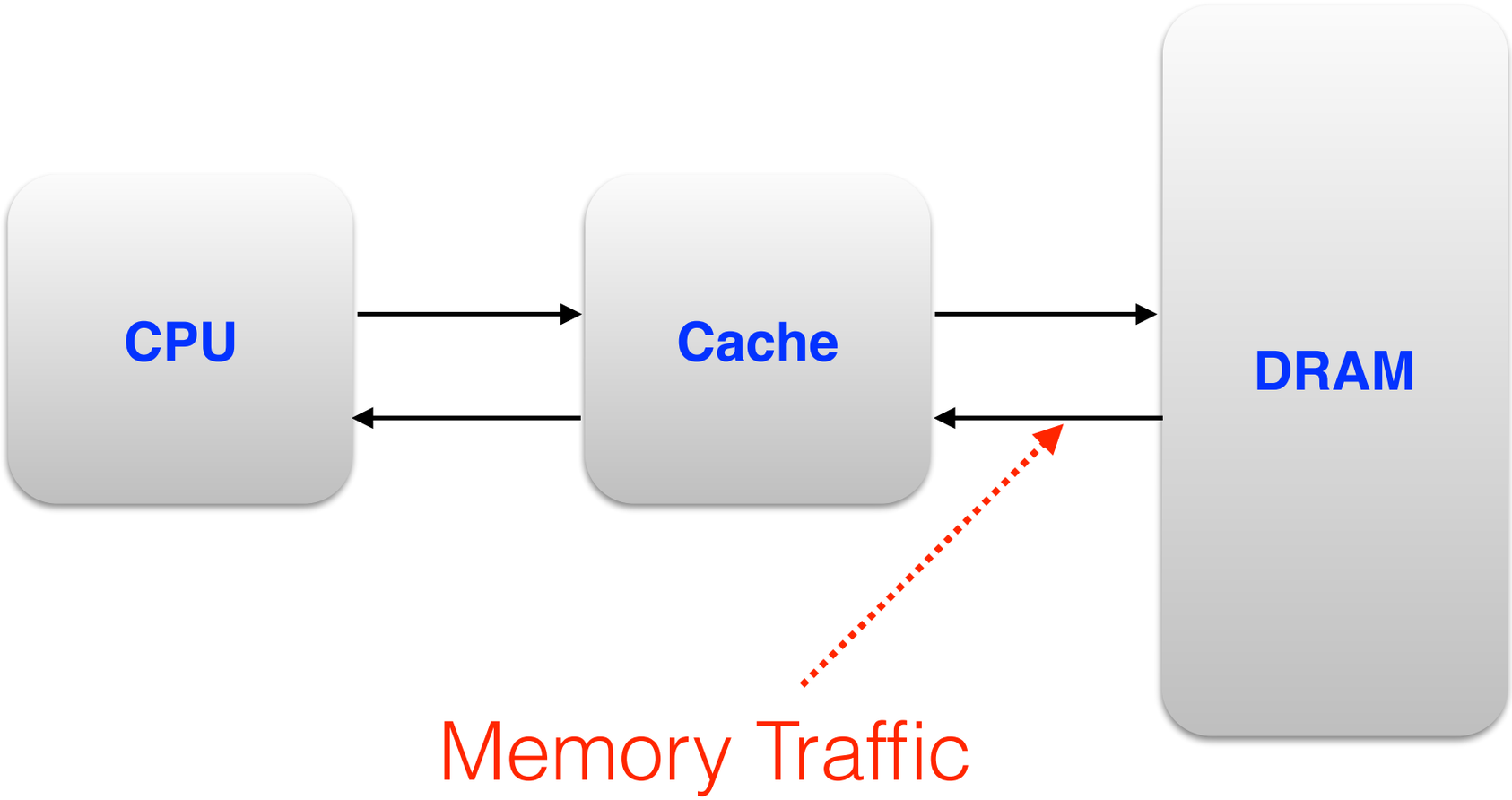


State-of-the-art in Detecting Performance Loss



Detecting performance loss

Memory Performance



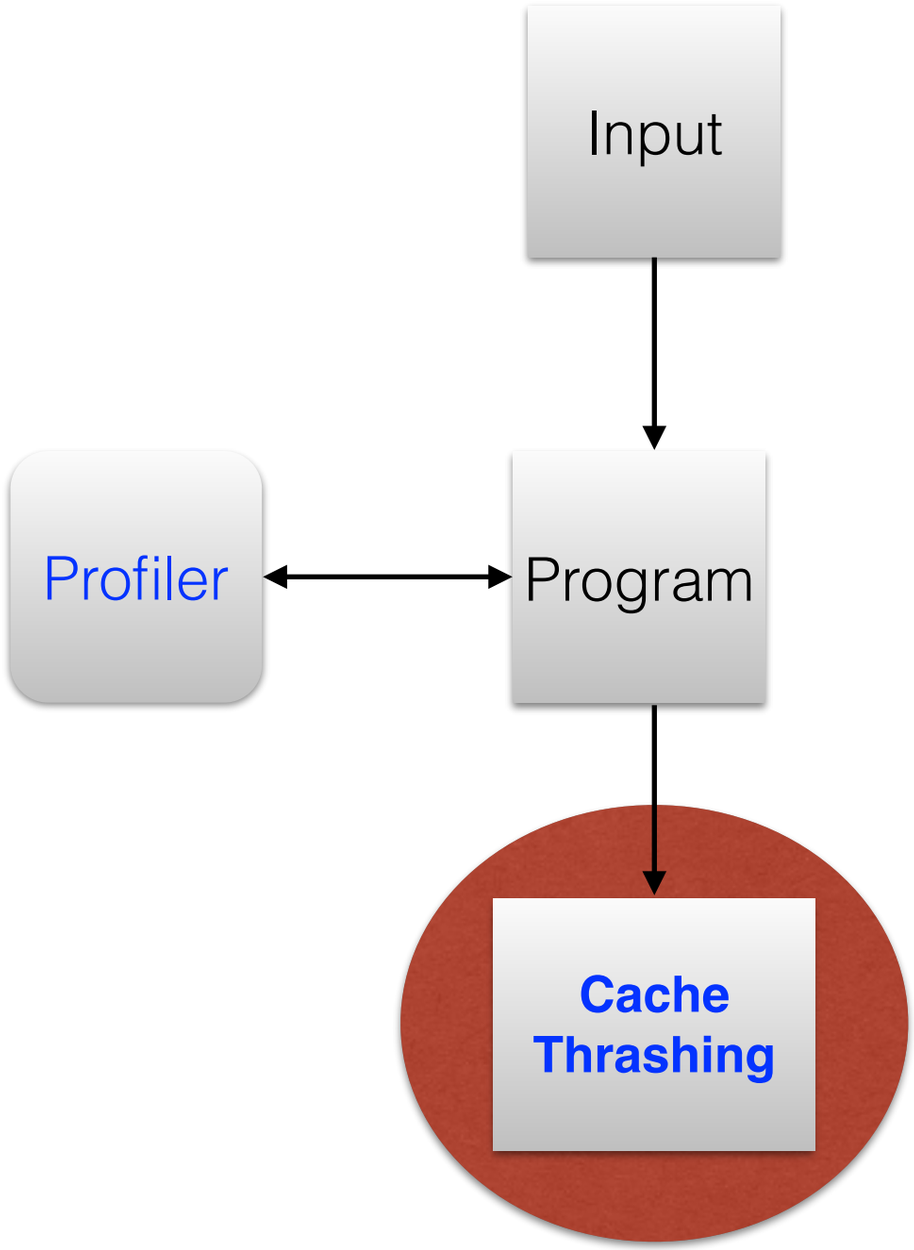
Typically, DRAM is several magnitudes slower than caches

Performance Loss

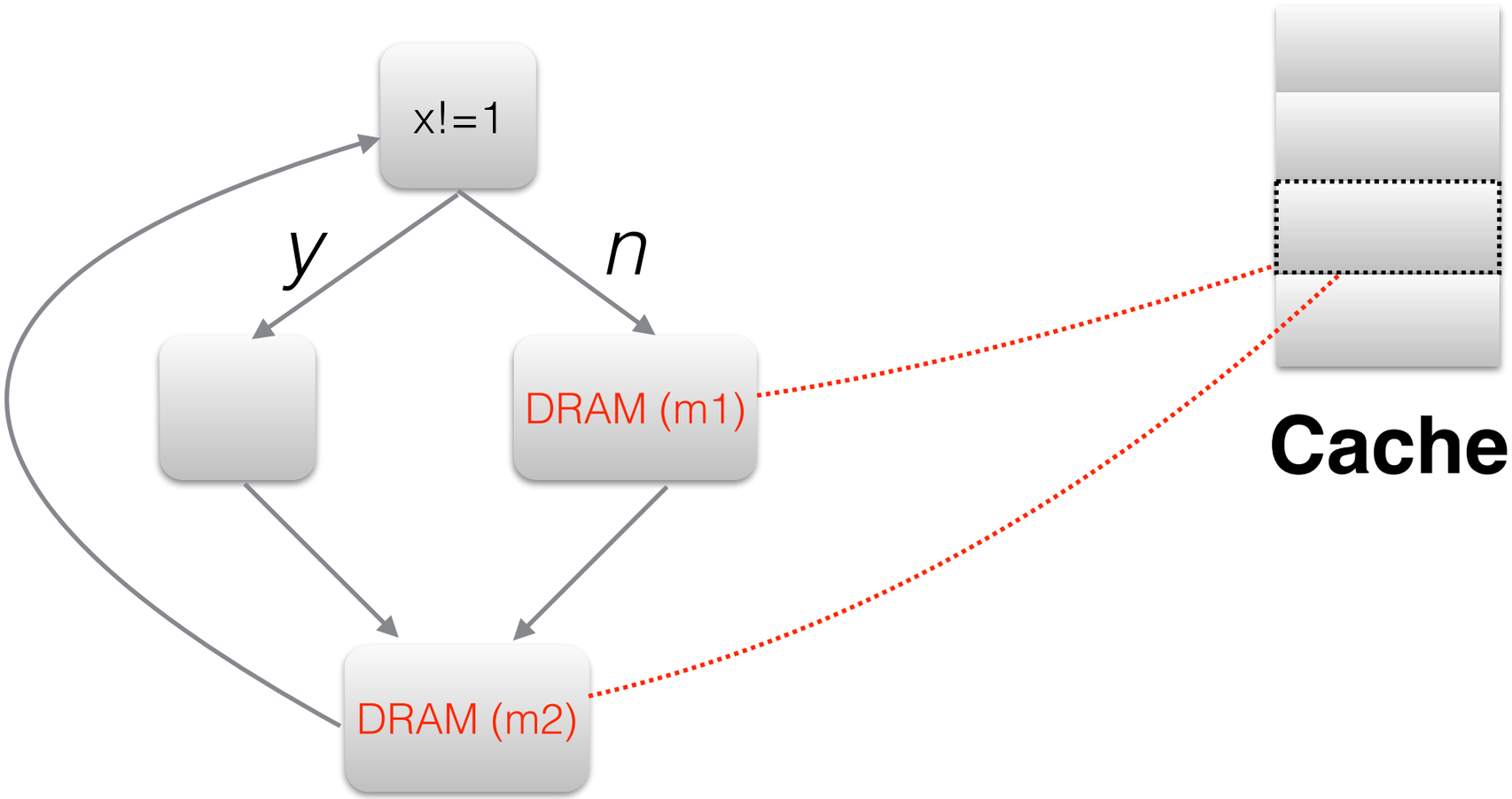
- Memory Performance
  - How do we define?
    - Many cache misses (How many is bad enough?)
- Our approach
  - Detect ***cache thrashing***

State-of-the-art in Detecting Performance Loss

Detecting performance loss



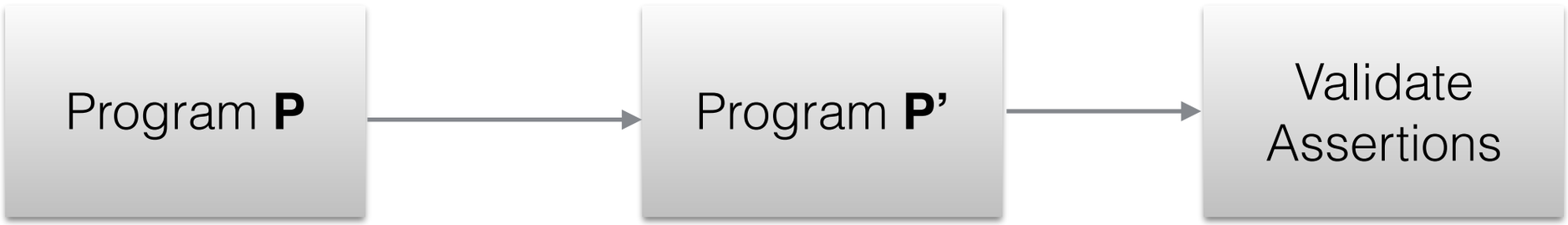
Cache Thrashing Scenario



m1 replaces m2 from the cache and vice versa

Test Generation

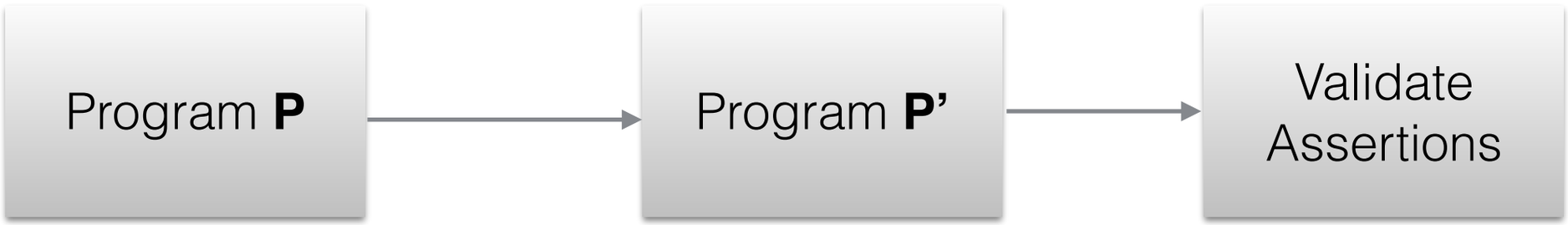
Encodes Cache Thrashing
of “Program P” as Assertions



Find Test Inputs
Invalidating Assertions

Test Generation

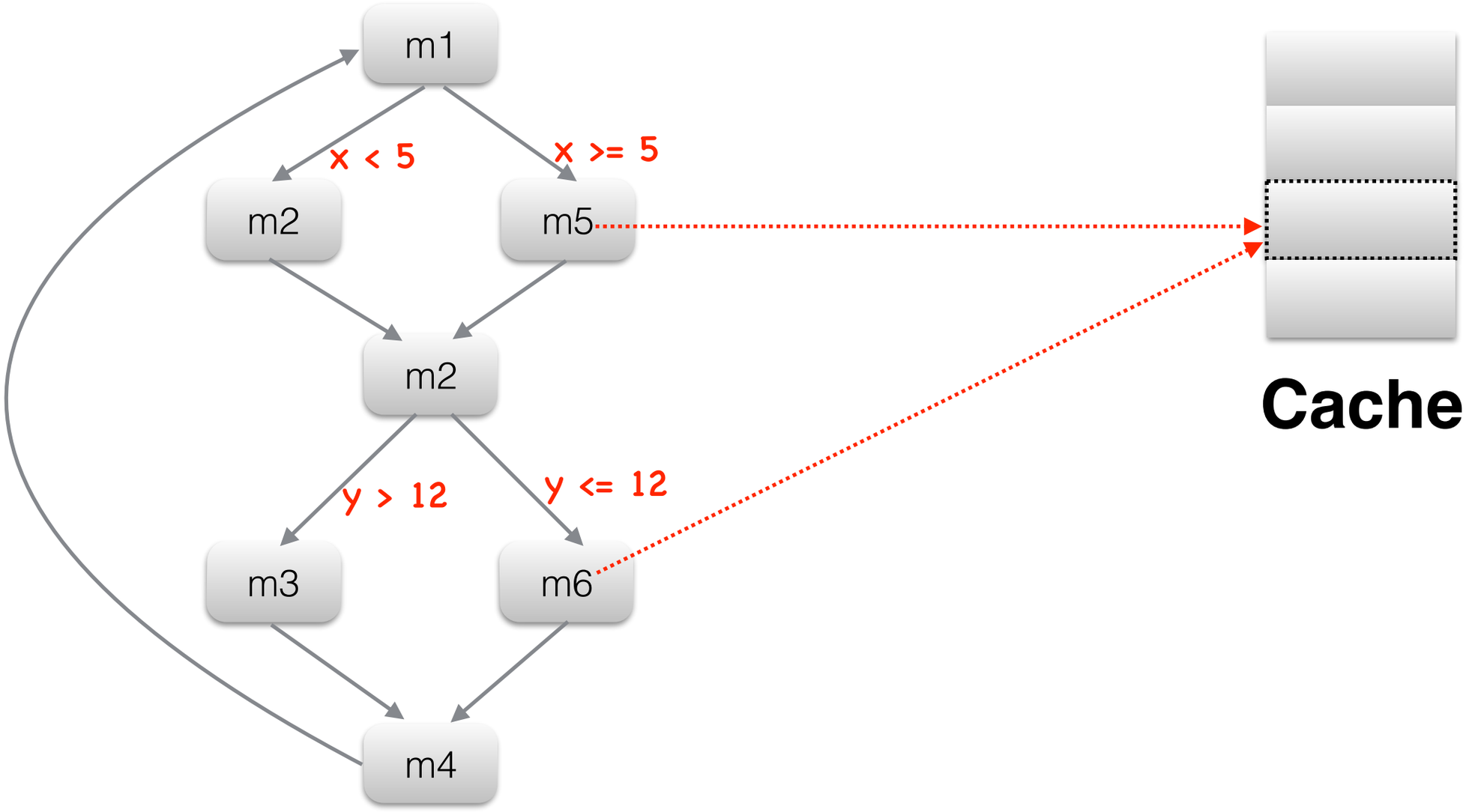
Encodes Cache Thrashing
of “Program P” as Assertions



Find Test Inputs
Invalidating Assertions

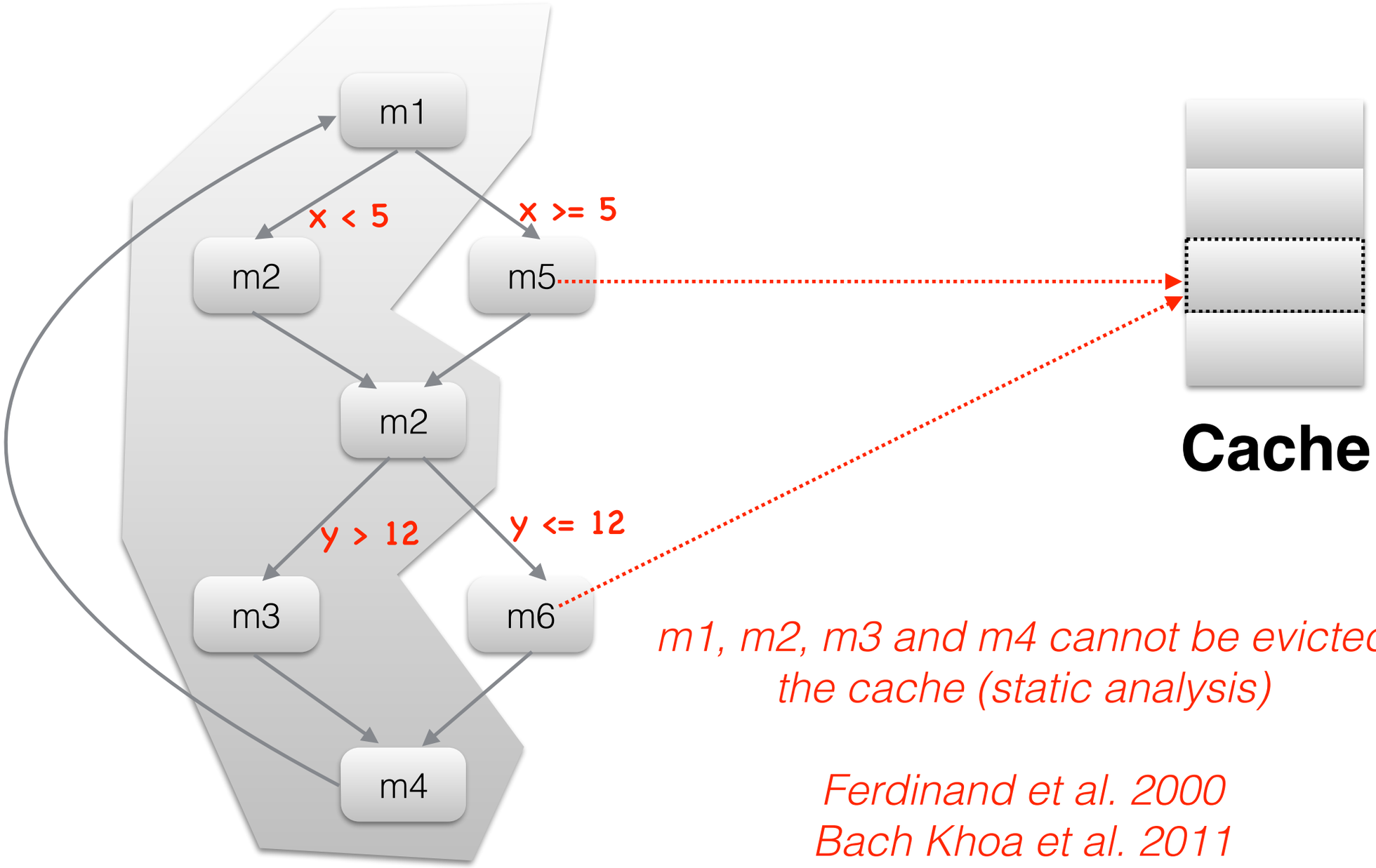
In other words, we reduce **memory performance testing** into
an equivalent **functionality testing** problem

Encoding Cache Thrashing Scenario



m5 and m6 map to the same cache set

Encoding Cache Thrashing Scenario

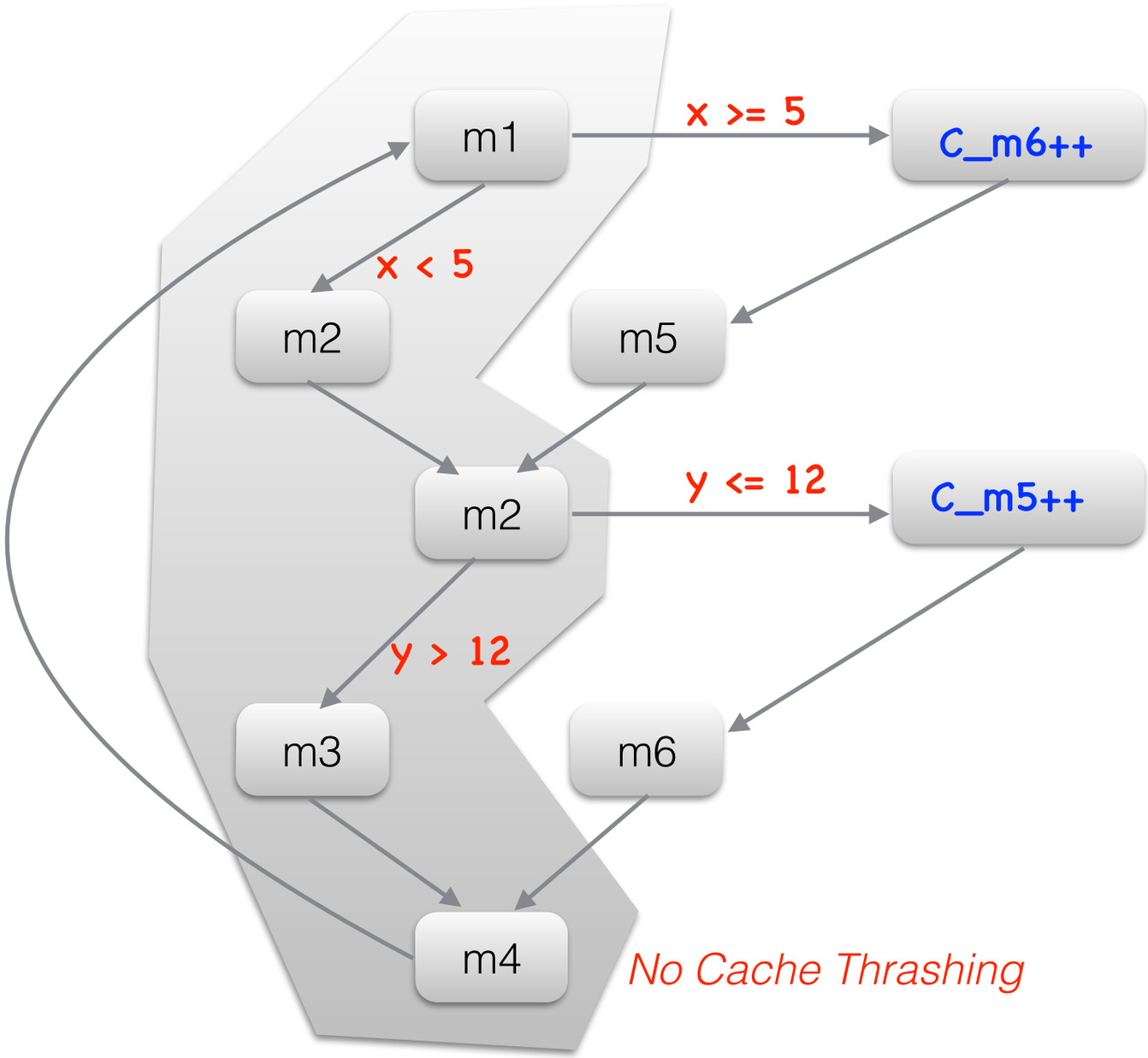


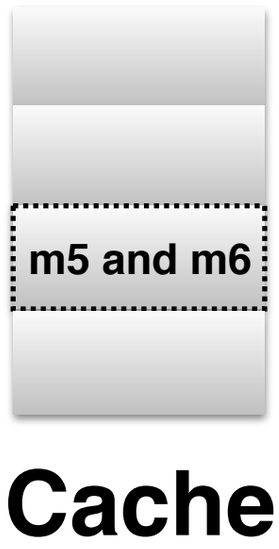
m1, m2, m3 and m4 cannot be evicted the cache (static analysis)

Ferdinand et al. 2000
Bach Khoa et al. 2011

m5 and m6 map to the same cache set

Encoding Cache Thrashing Scenario

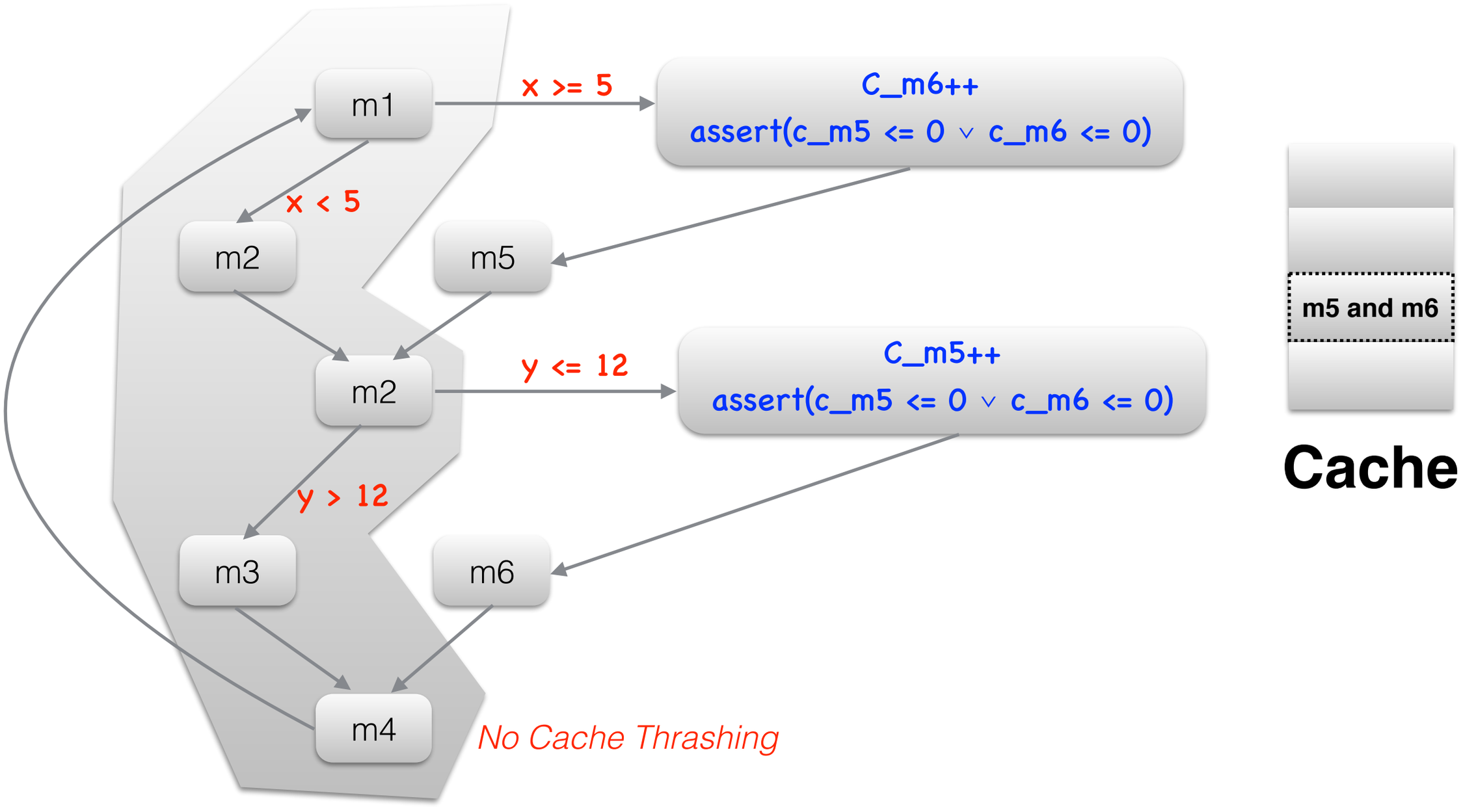




No Cache Thrashing

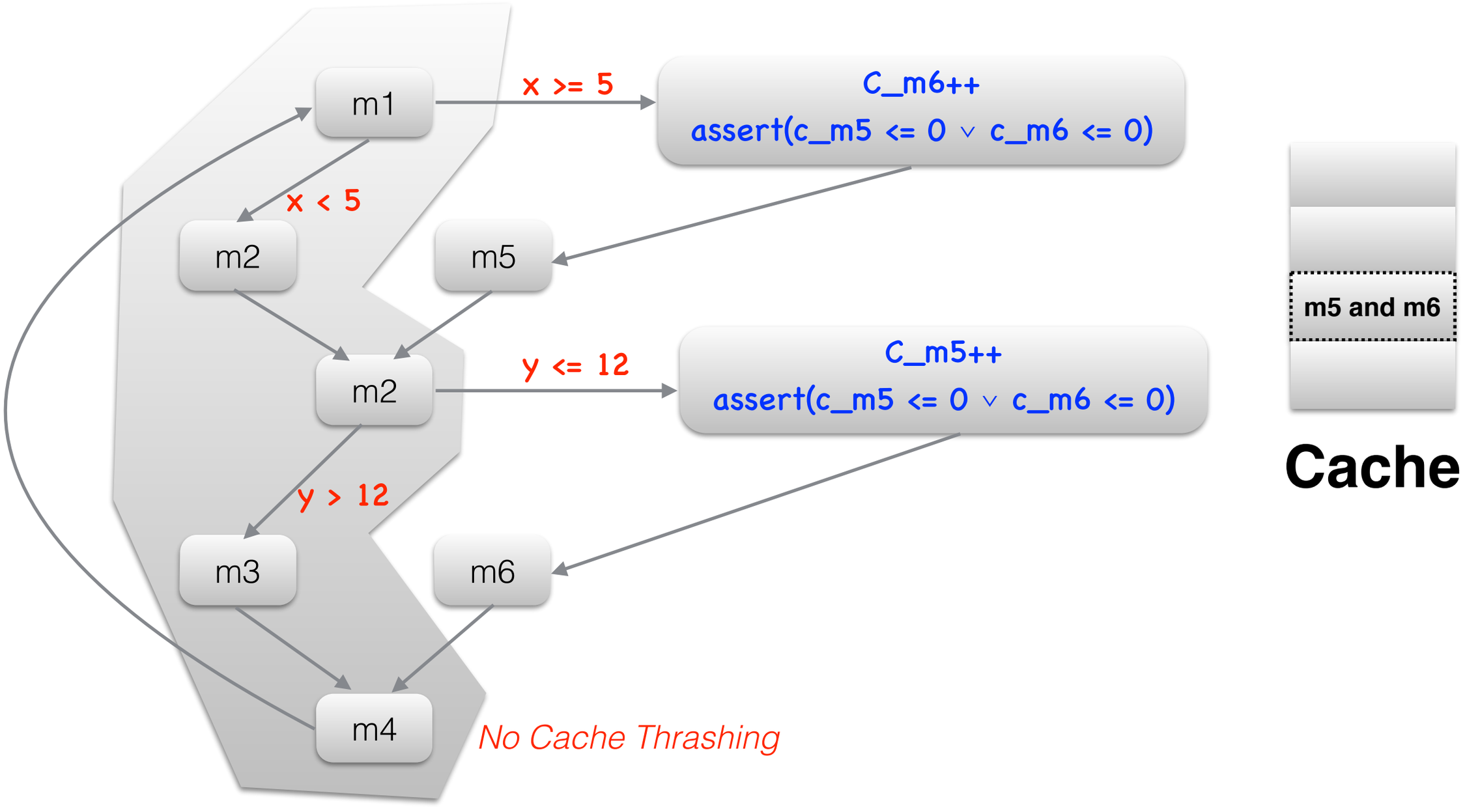
m5 and m6 map to the same cache set

Encoding Cache Thrashing Scenario



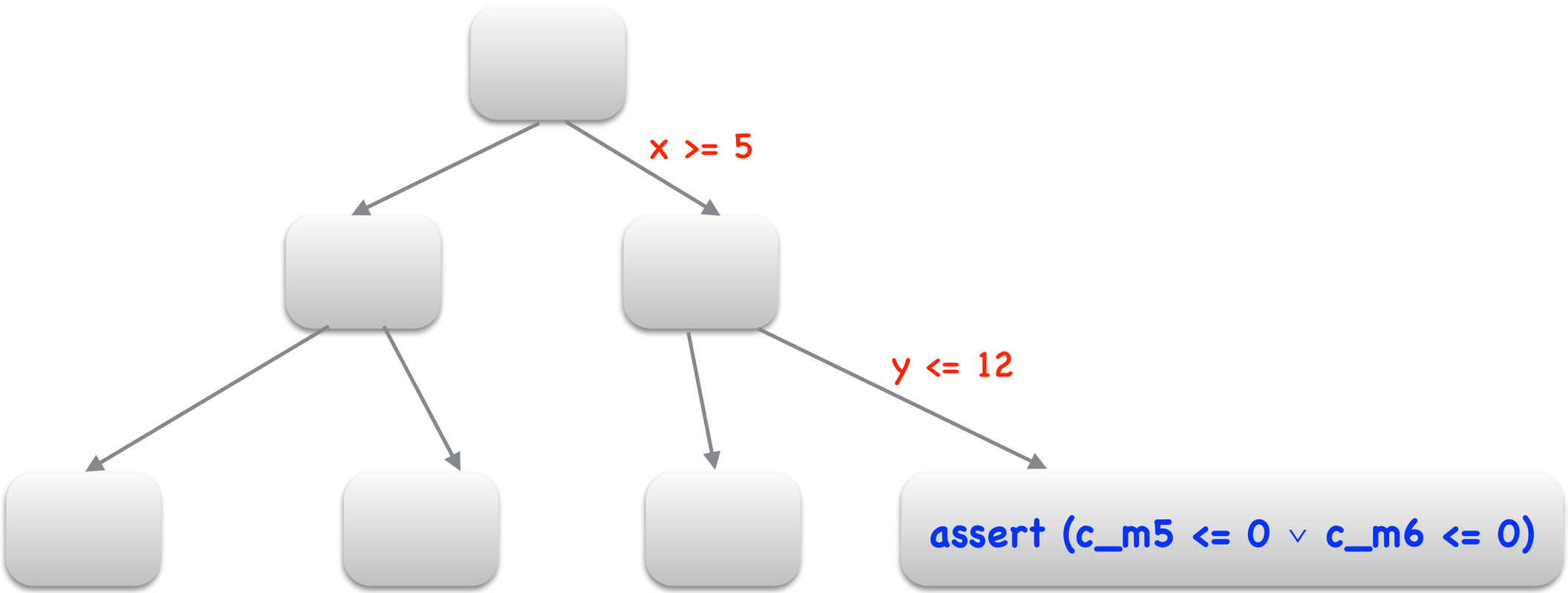
m5 and m6 map to the same cache set

Encoding Cache Thrashing Scenario



Test Generation is Performed on the modified program

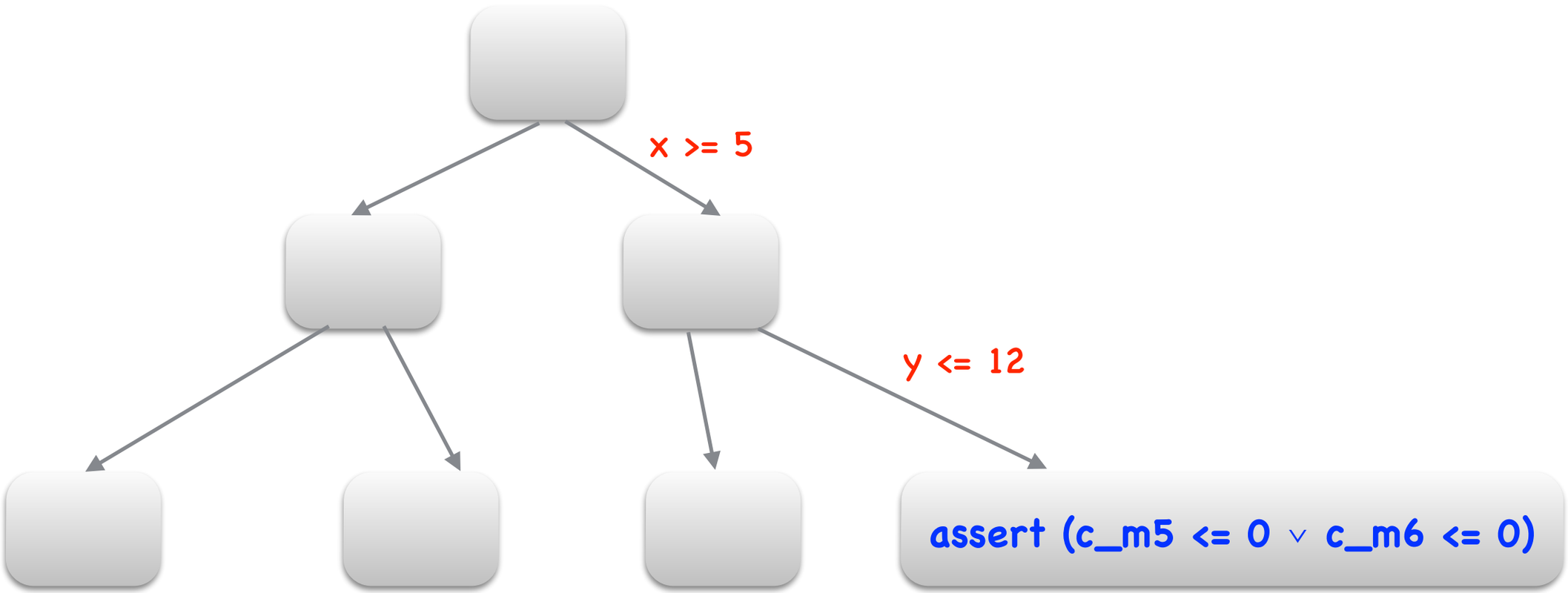
Test Generation Approach



Dynamic Symbolic Execution
Guidance via Control Dependency Graph

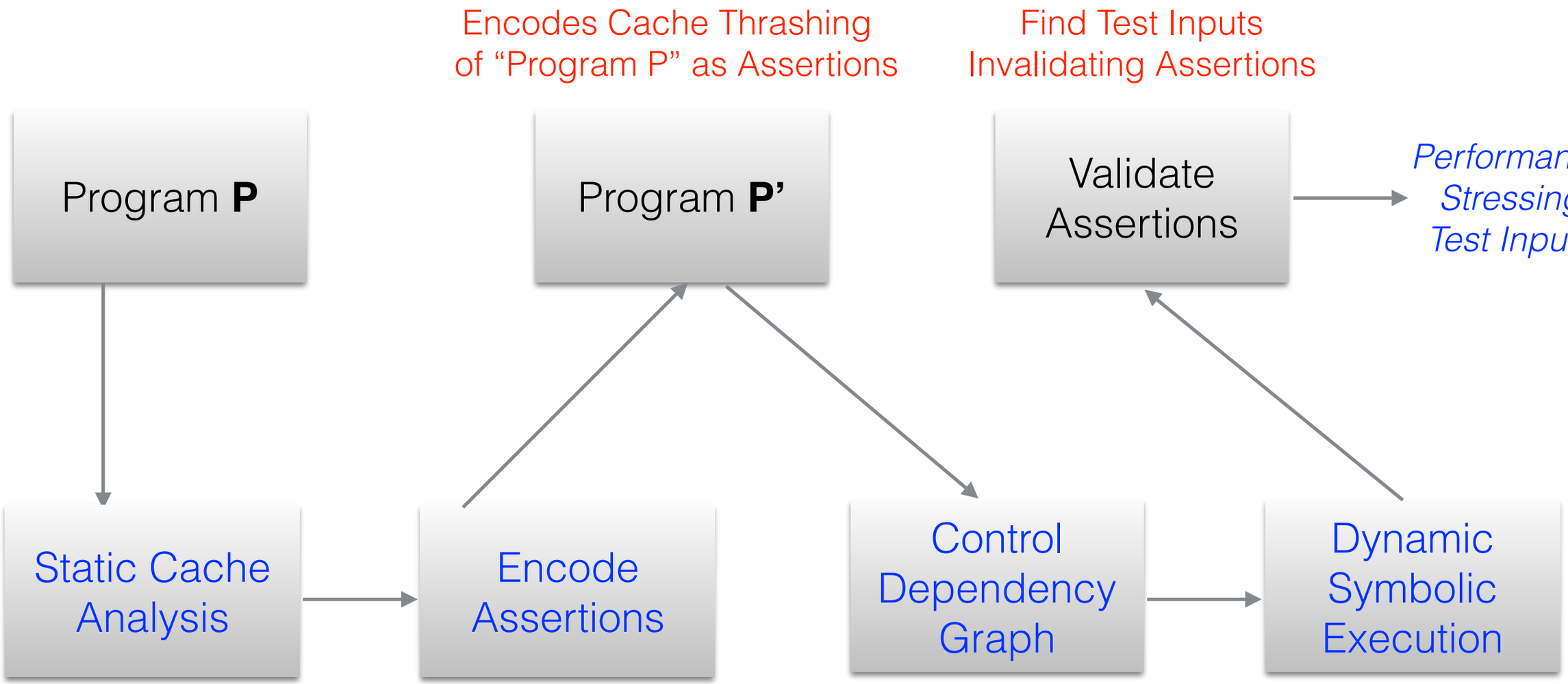
(reaching path to Assertions)

Test Generation Approach

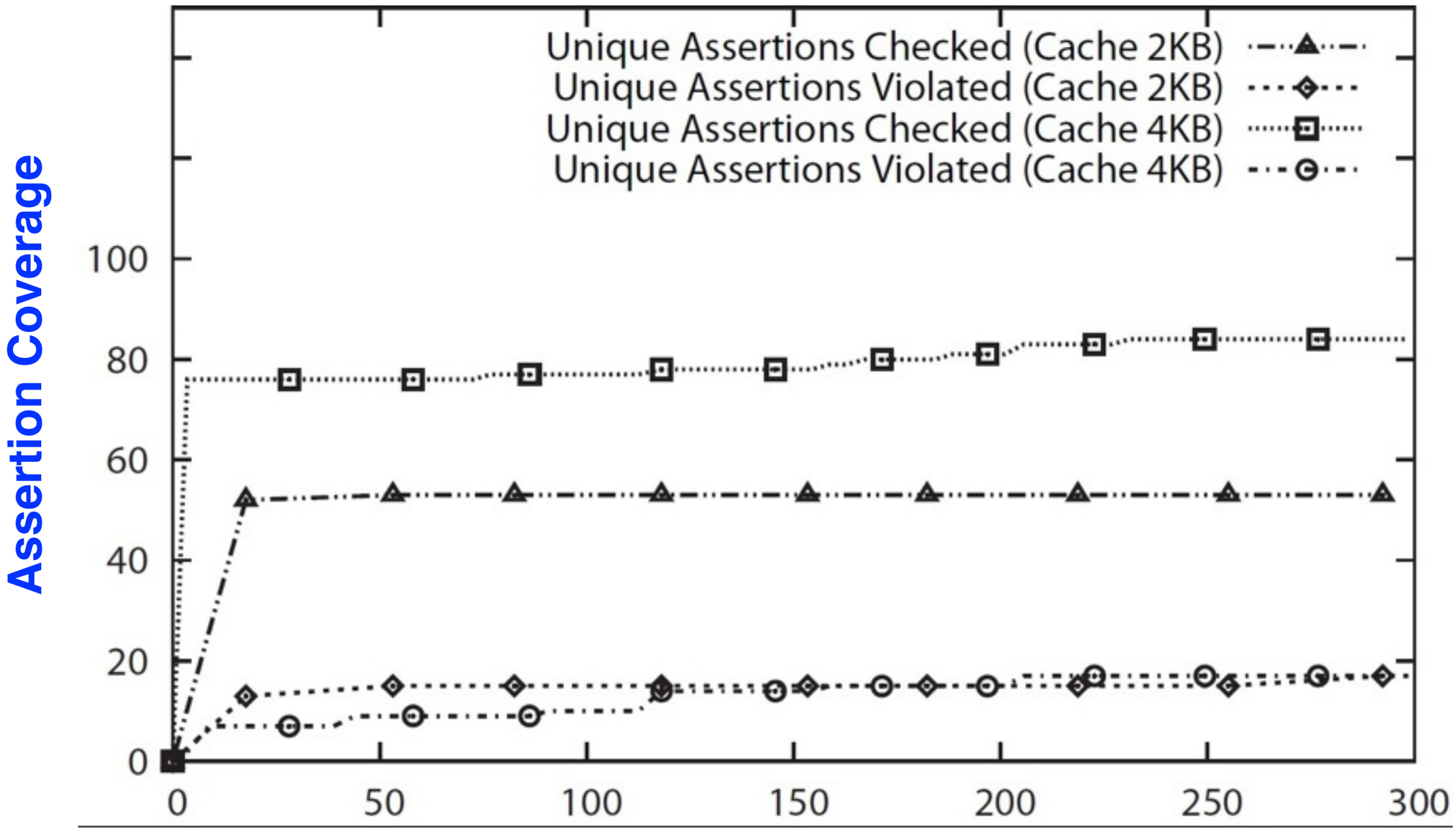


Generate inputs satisfying (x >= 5 /\ y <= 12)

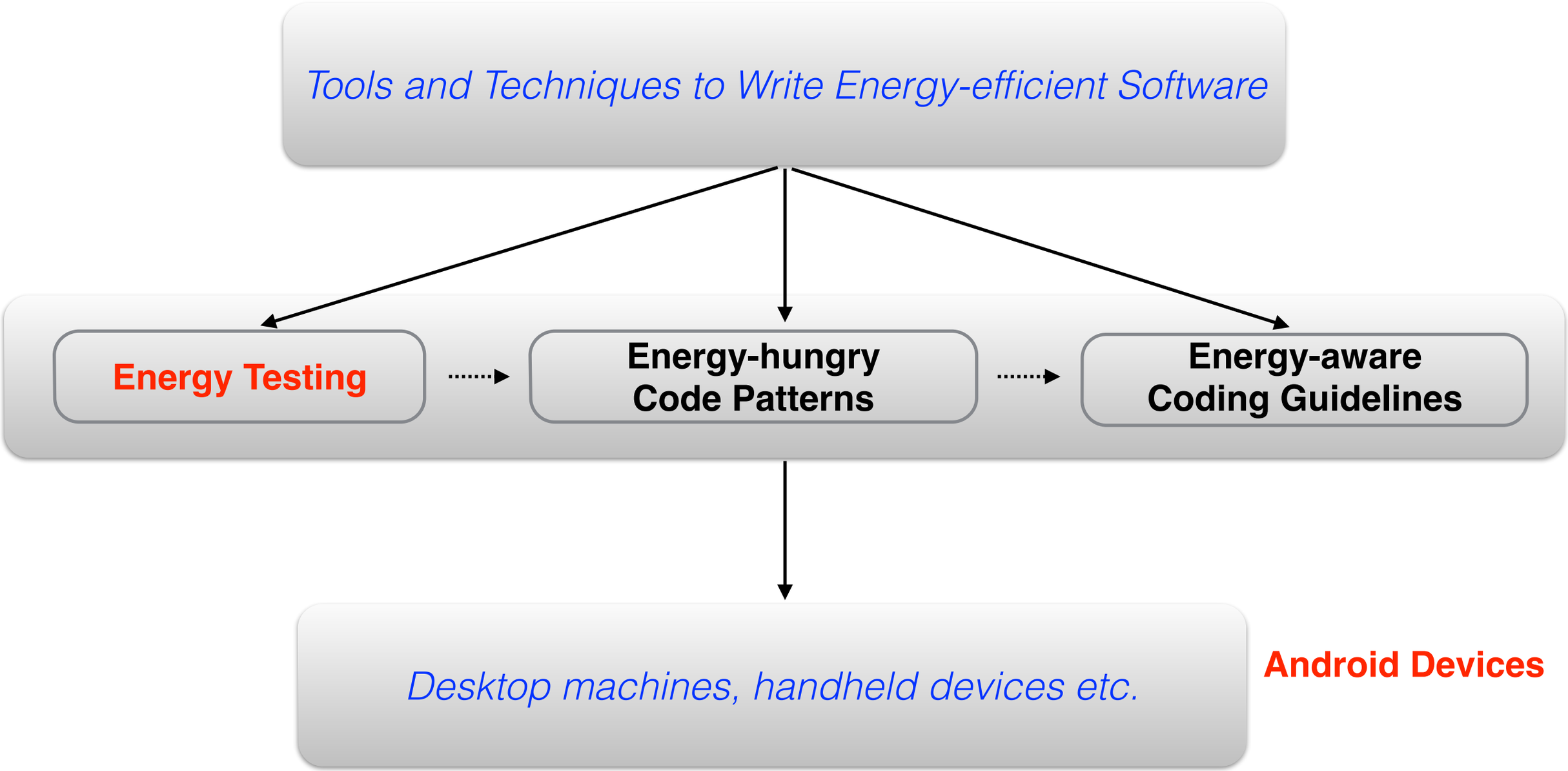
Summary


Encodes Cache Thrashing
of “Program P” as Assertions
Find Test Inputs
Invalidating Assertions
Program **P**
Program **P’**
Validate
Assertions
*Performance
Stressing
Test Inputs*
Static Cache
Analysis
Encode
Assertions
Control
Dependency
Graph
Dynamic
Symbolic
Execution

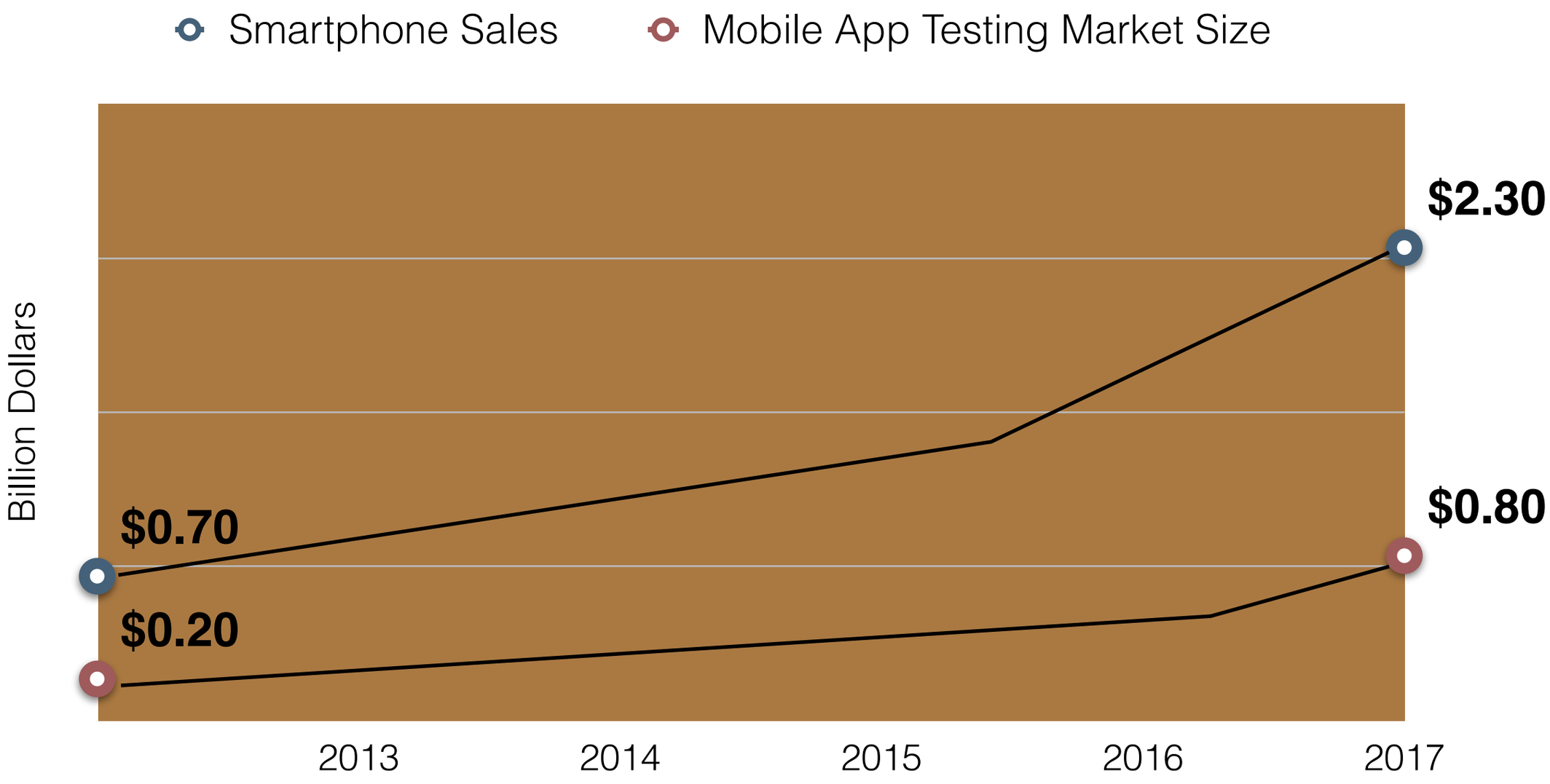
Evaluation


Unique Assertions Checked (Cache 2KB)
Unique Assertions Violated (Cache 2KB)
Unique Assertions Checked (Cache 4KB)
Unique Assertions Violated (Cache 4KB)
Assertion Coverage
0
20
40
60
80
100
0
50
100
150
200
250
300
Time (in Seconds)

Overview


Tools and Techniques to Write Energy-efficient Software
Energy Testing
Energy-hungry Code Patterns
Energy-aware Coding Guidelines
Desktop machines, handheld devices etc.
Android Devices

Smartphone Market



Data obtained from IDC, Gartner and ABI Research

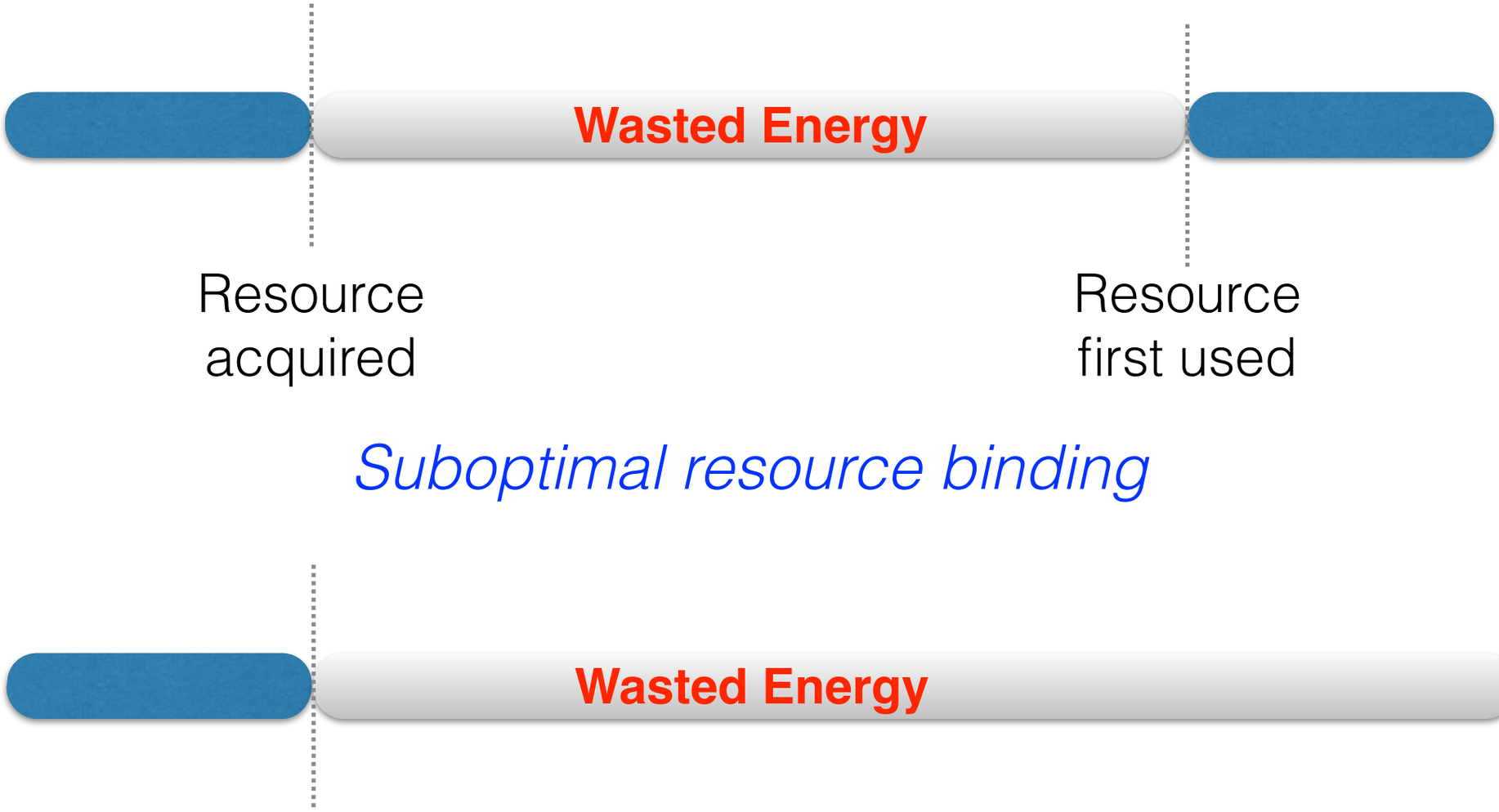
Energy Inefficiency

- How do we quantify energy inefficiency?
  - High energy consumption, what is ***high?***
- High energy consumption
  - High utilization of hardware components
  - *Low utilization of hardware components*
- Ratio Energy/Utilization

Energy Inefficiency

Cause/Source		
Hardware components	Resource leak	Suboptimal resource binding
Sleep state transition	Wakelock bug	Tail Energy hotspot
Background Service	Vacuous background service	Expensive background service
Defective Functionality	Immortality bug	Loop energy hotspot

Energy Inefficiency



A Broader Categorization

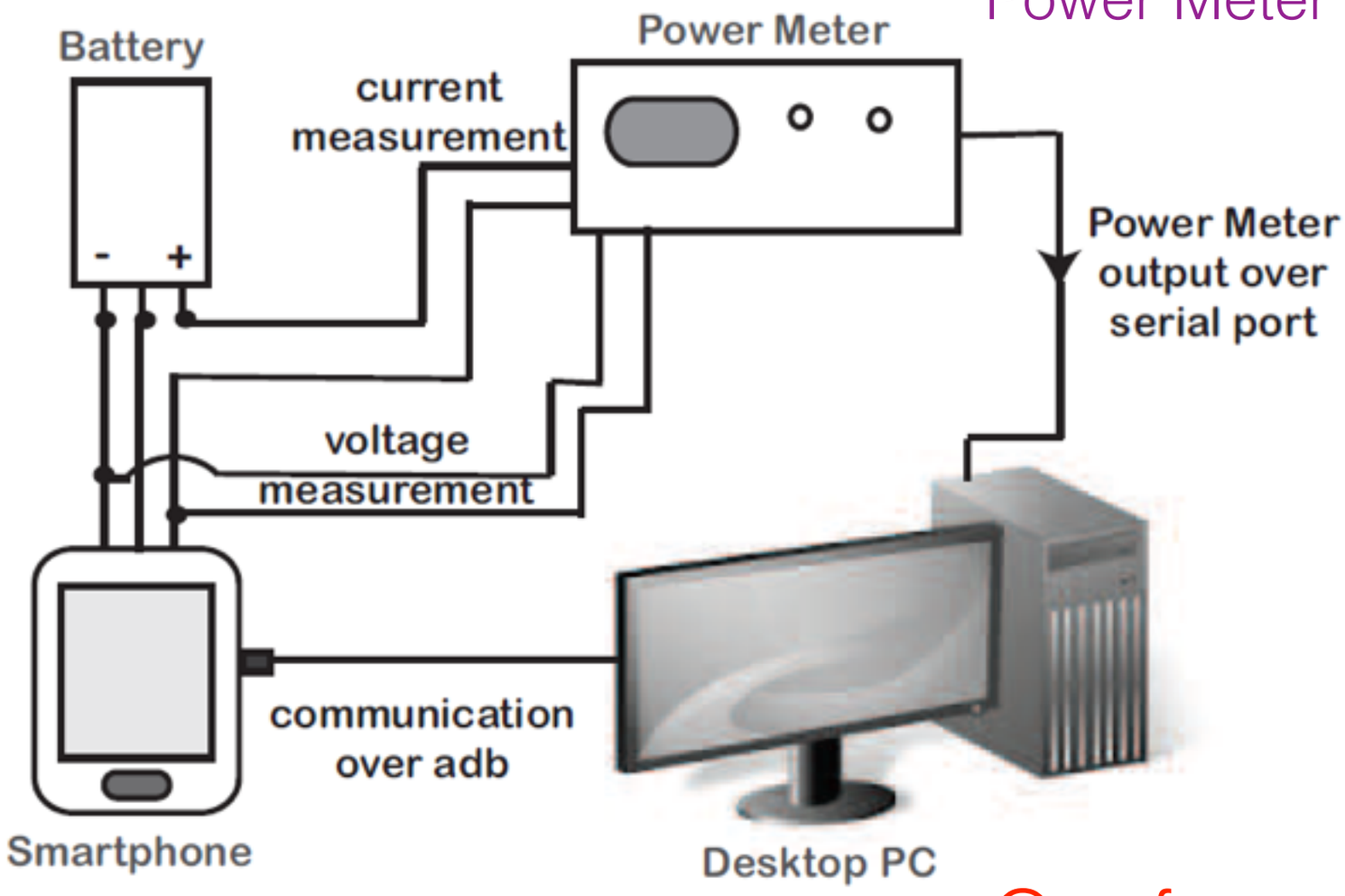
Cause/Source	Energy Bugs	Energy Hotspots
Hardware components	Resource leak	Suboptimal resource binding
Sleep state transition	Wakelock bug	Tail Energy hotspot
Background Service	Vacuous background service	Expensive background service
Defective Functionality	Immortality bug	Loop energy hotspot

Device does not return to idle

High energy consumption + low utilization

Measurement

Yokogawa Digital Power Meter

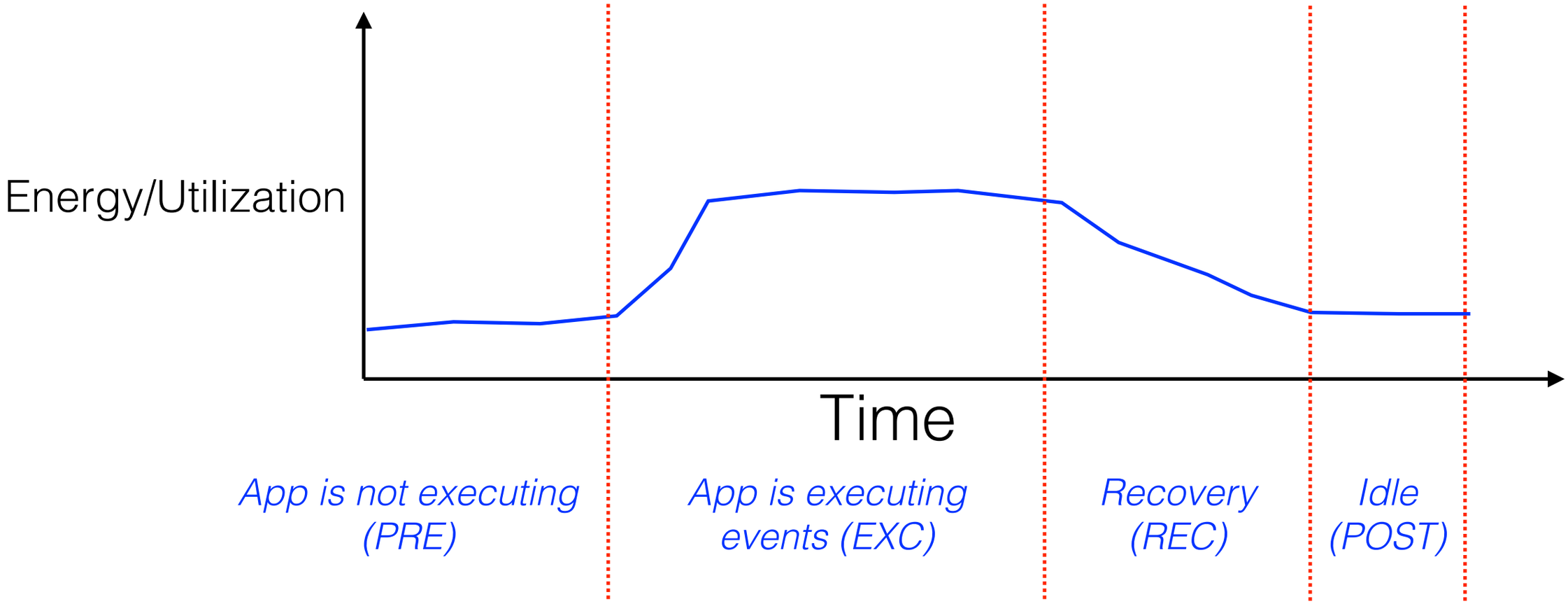


LG Optimus smartphone

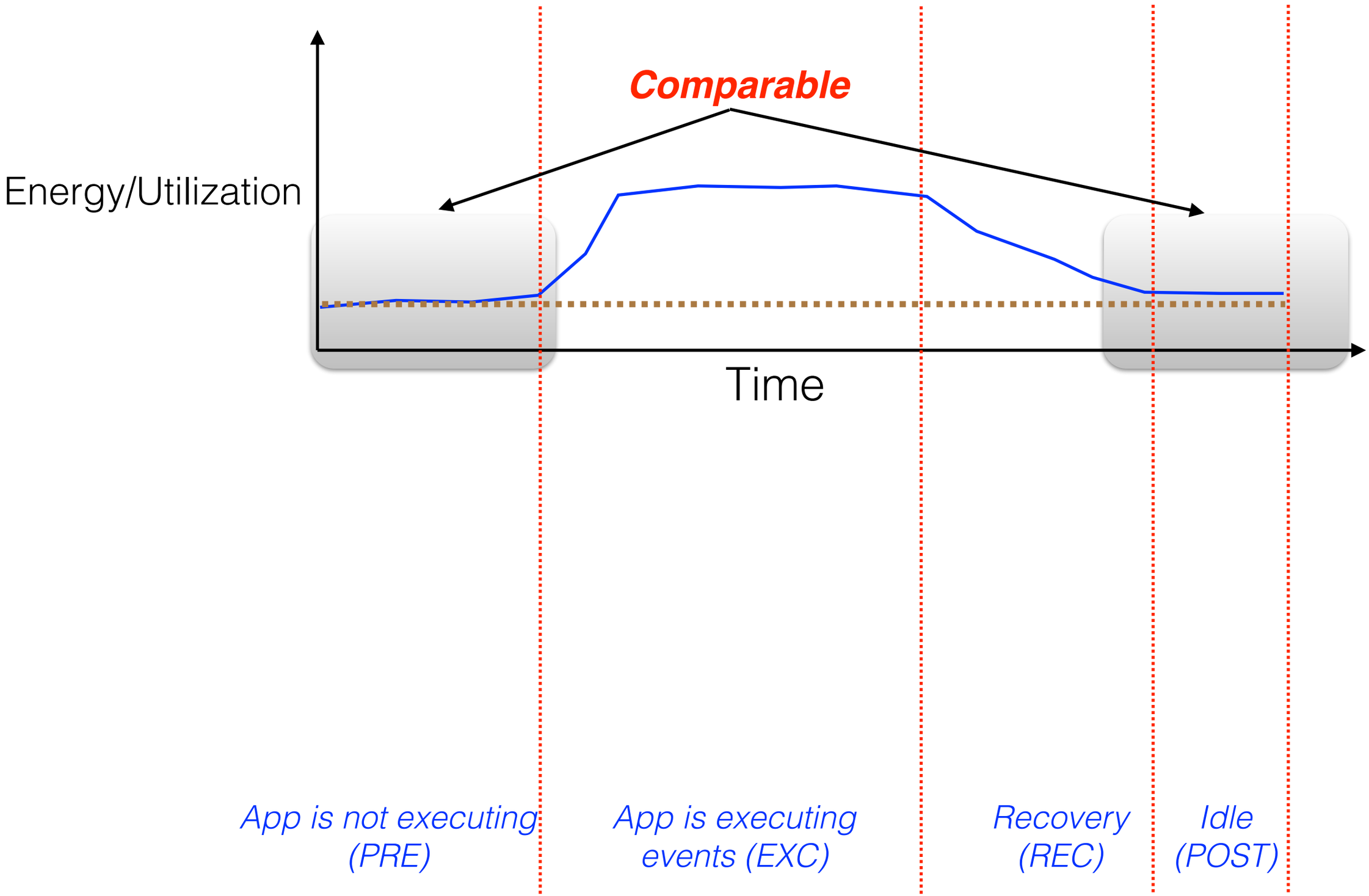
Our framework

Measurement

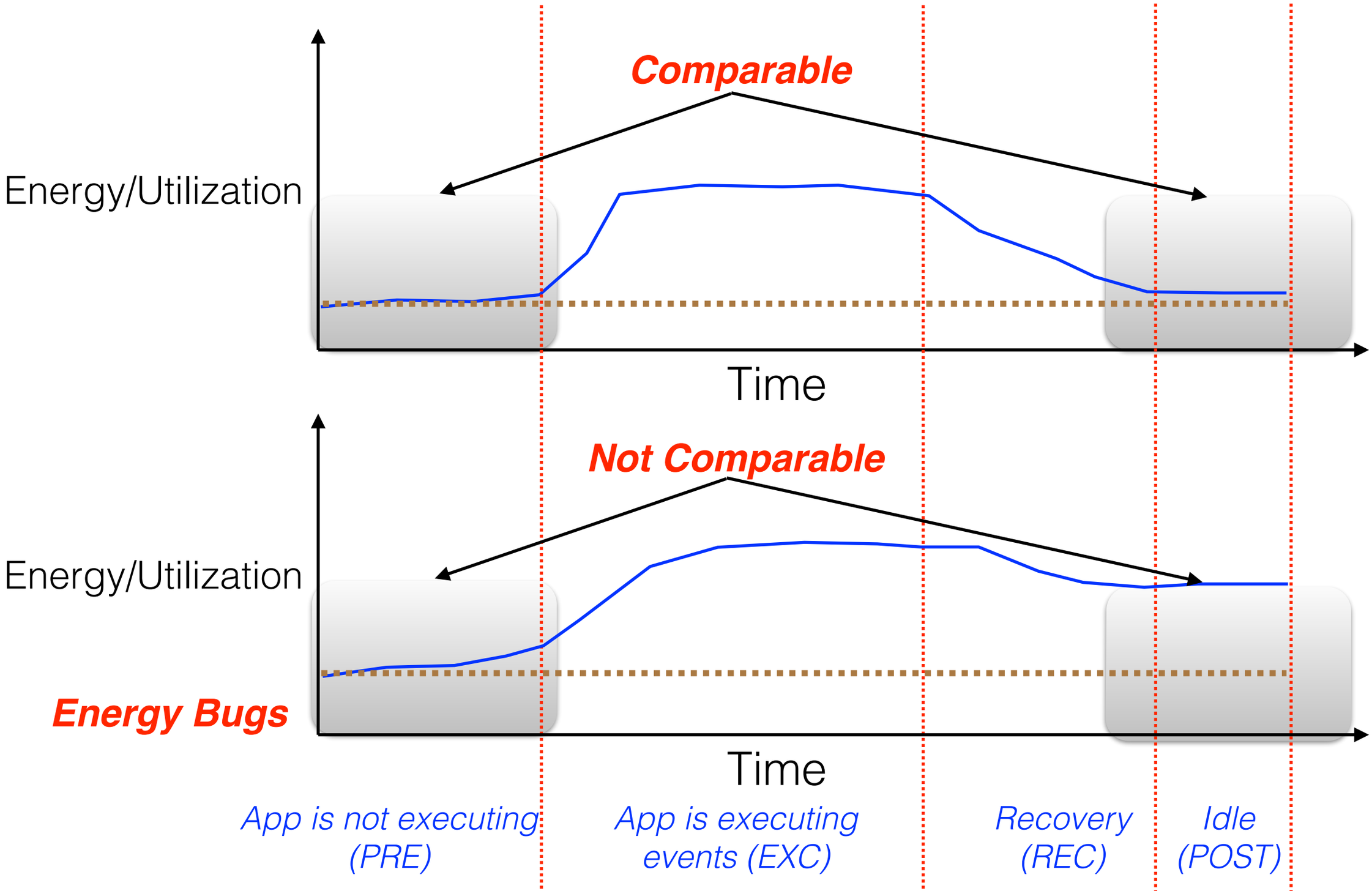
- Measuring Energy/Utilization ratio for an application



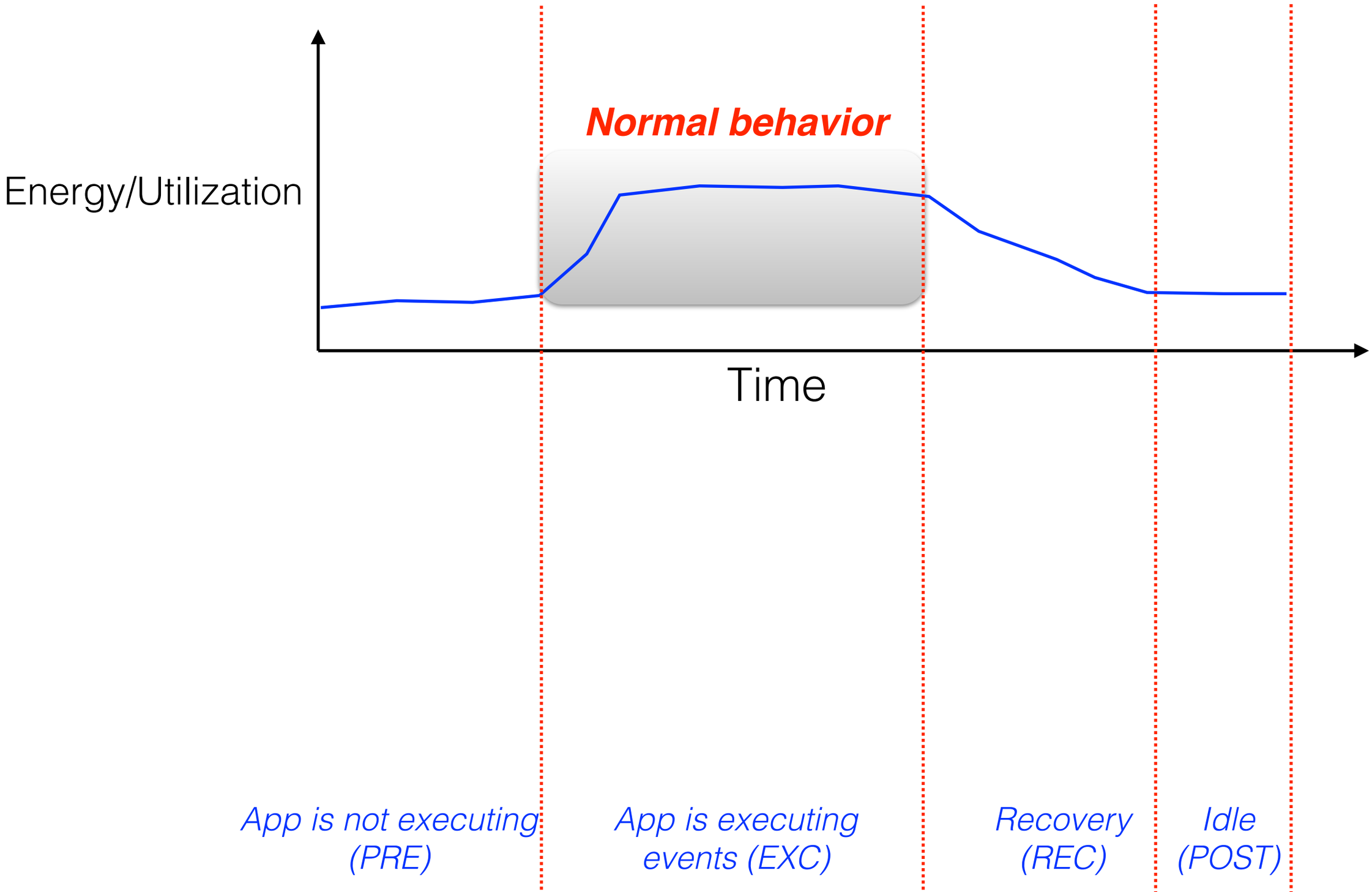
Energy Inefficiency



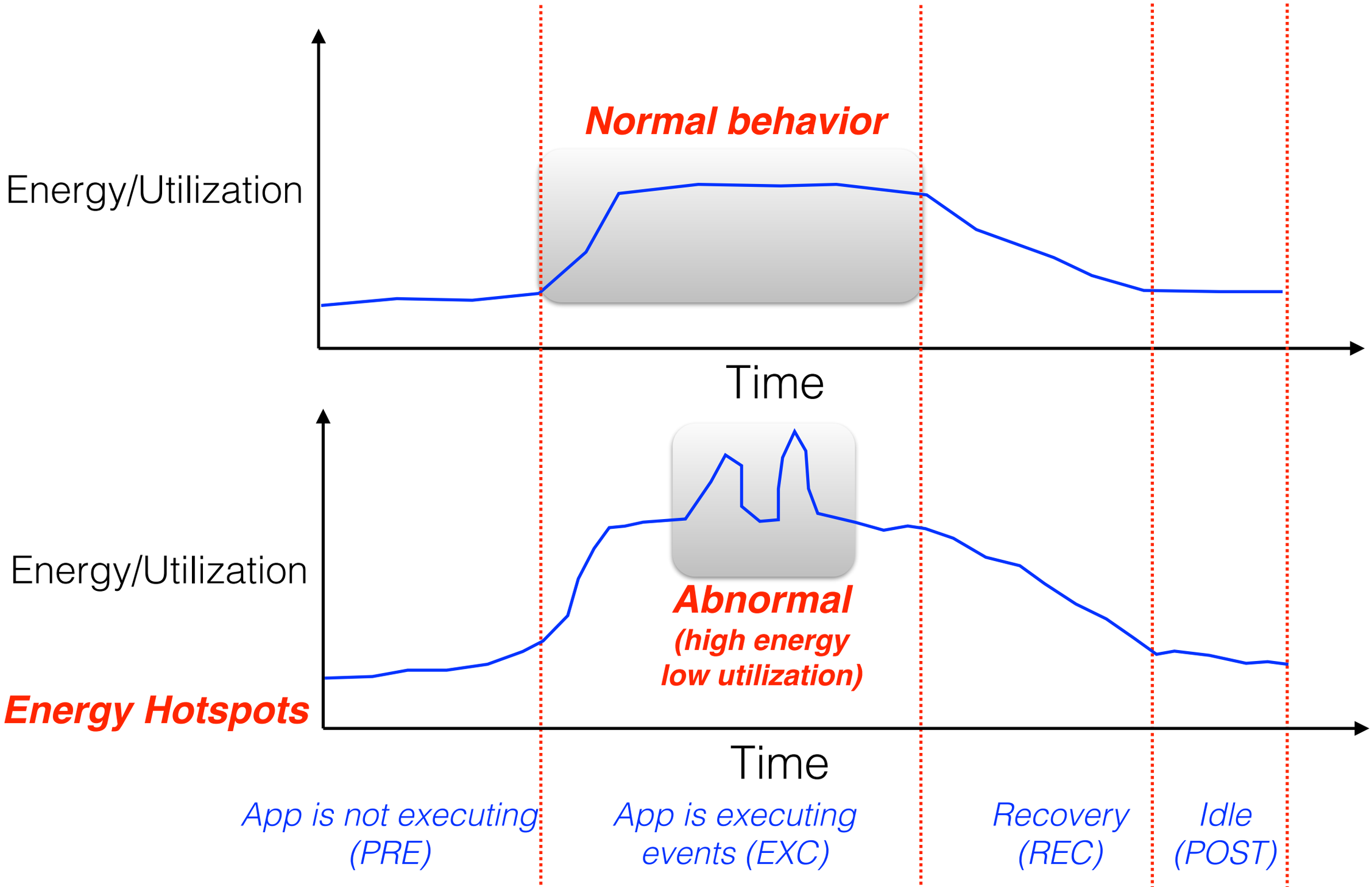
Energy Inefficiency



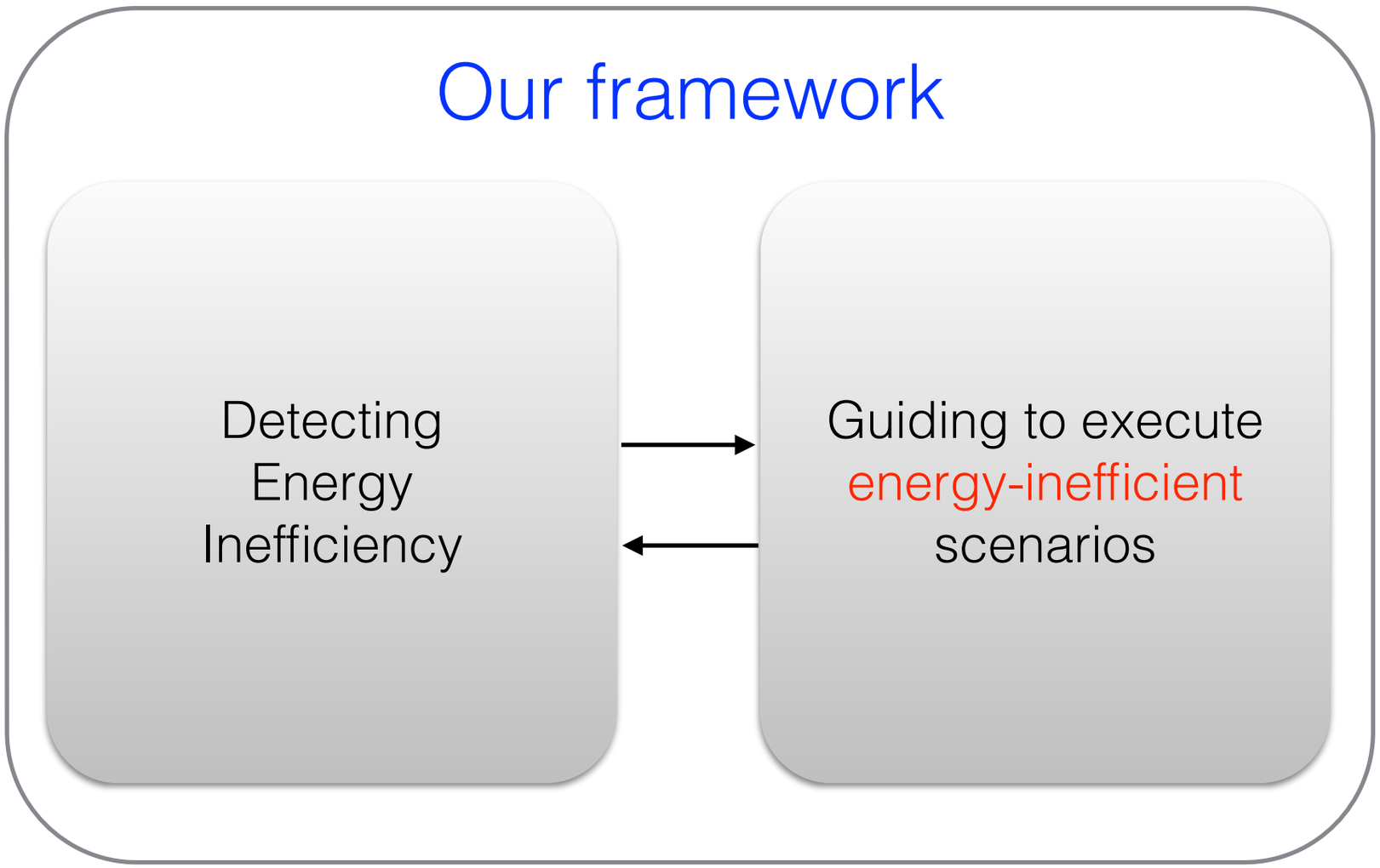
Energy Inefficiency



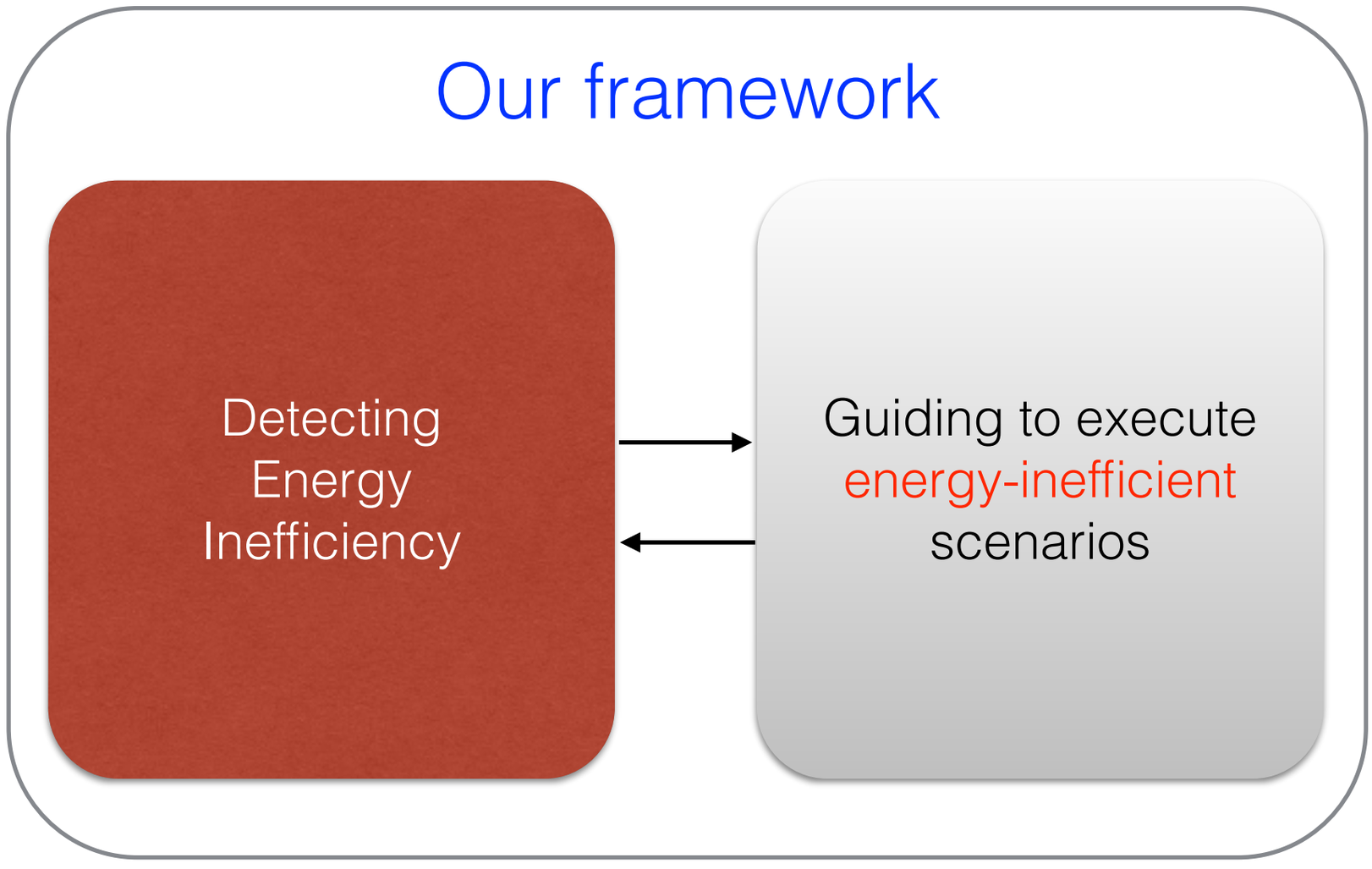
Energy Inefficiency



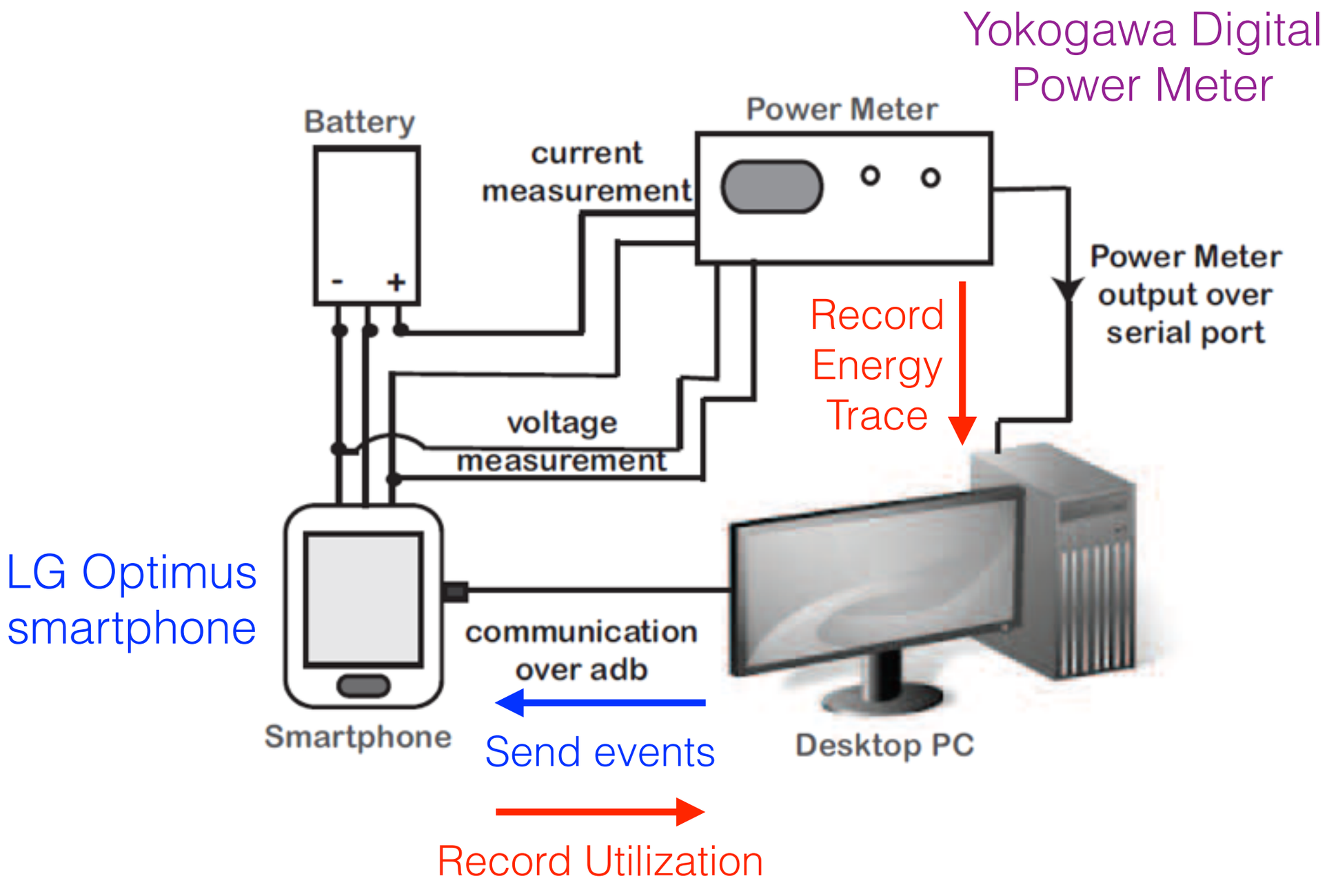
Test Generation



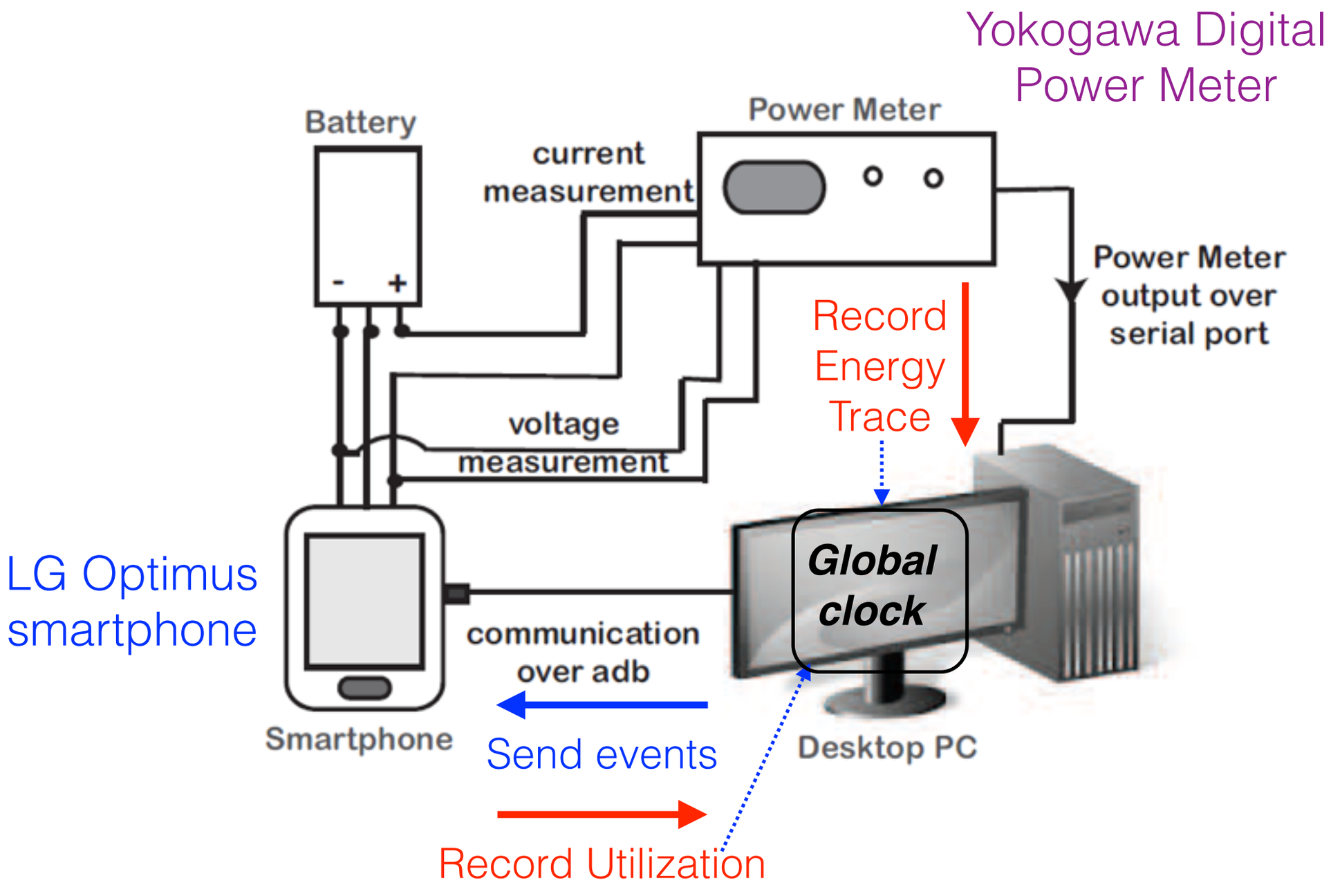
Test Generation



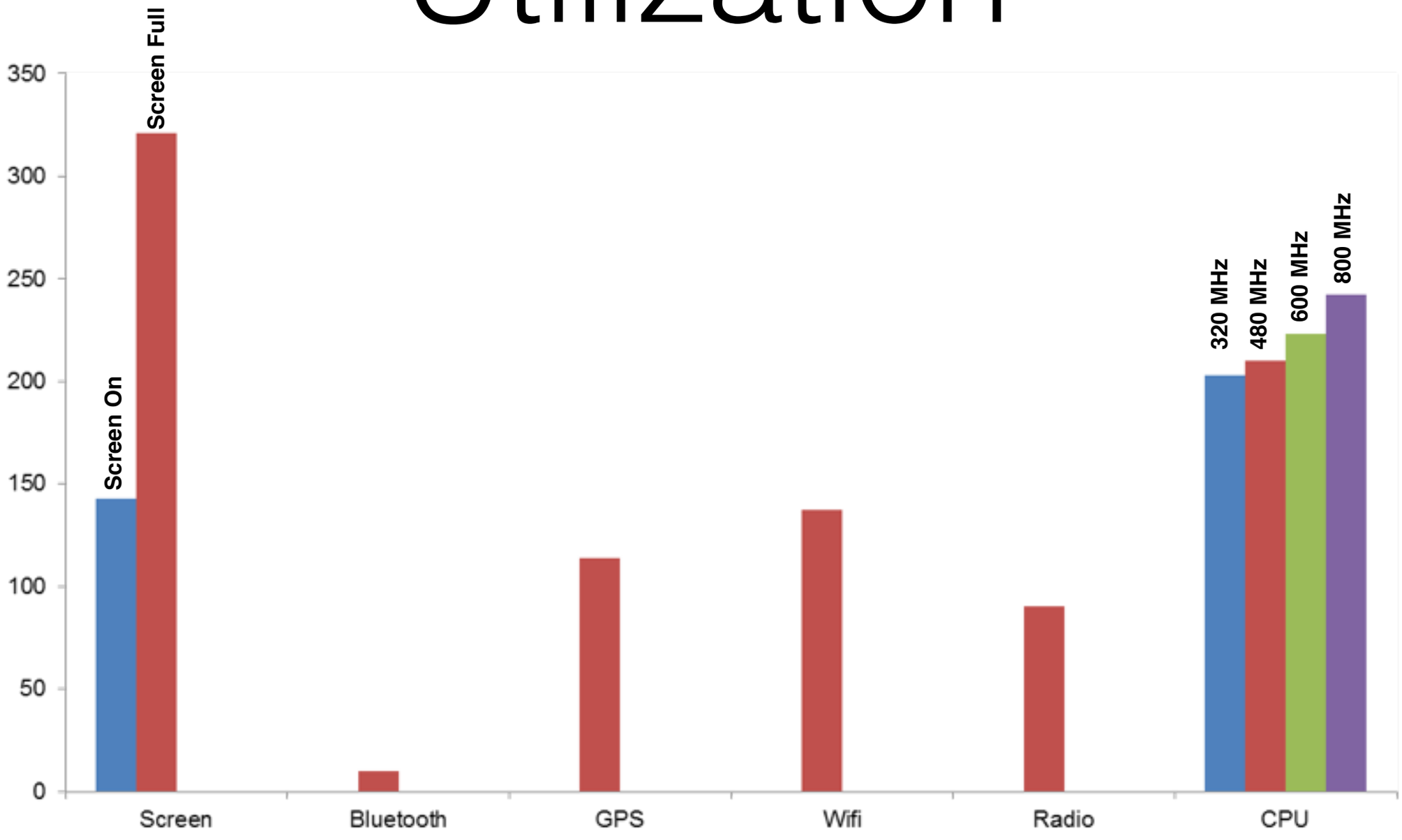
Measurement



Measurement



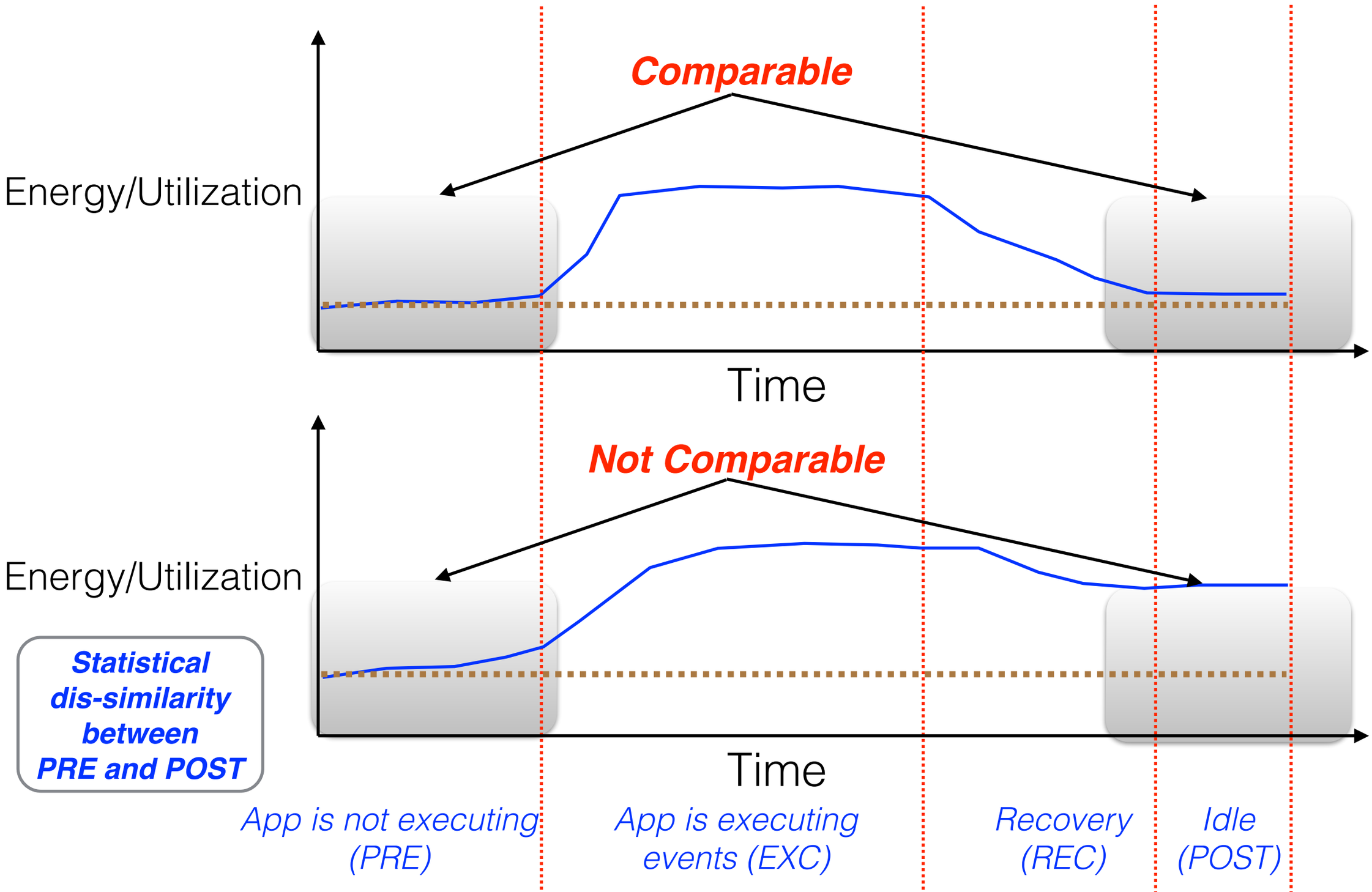
Utilization



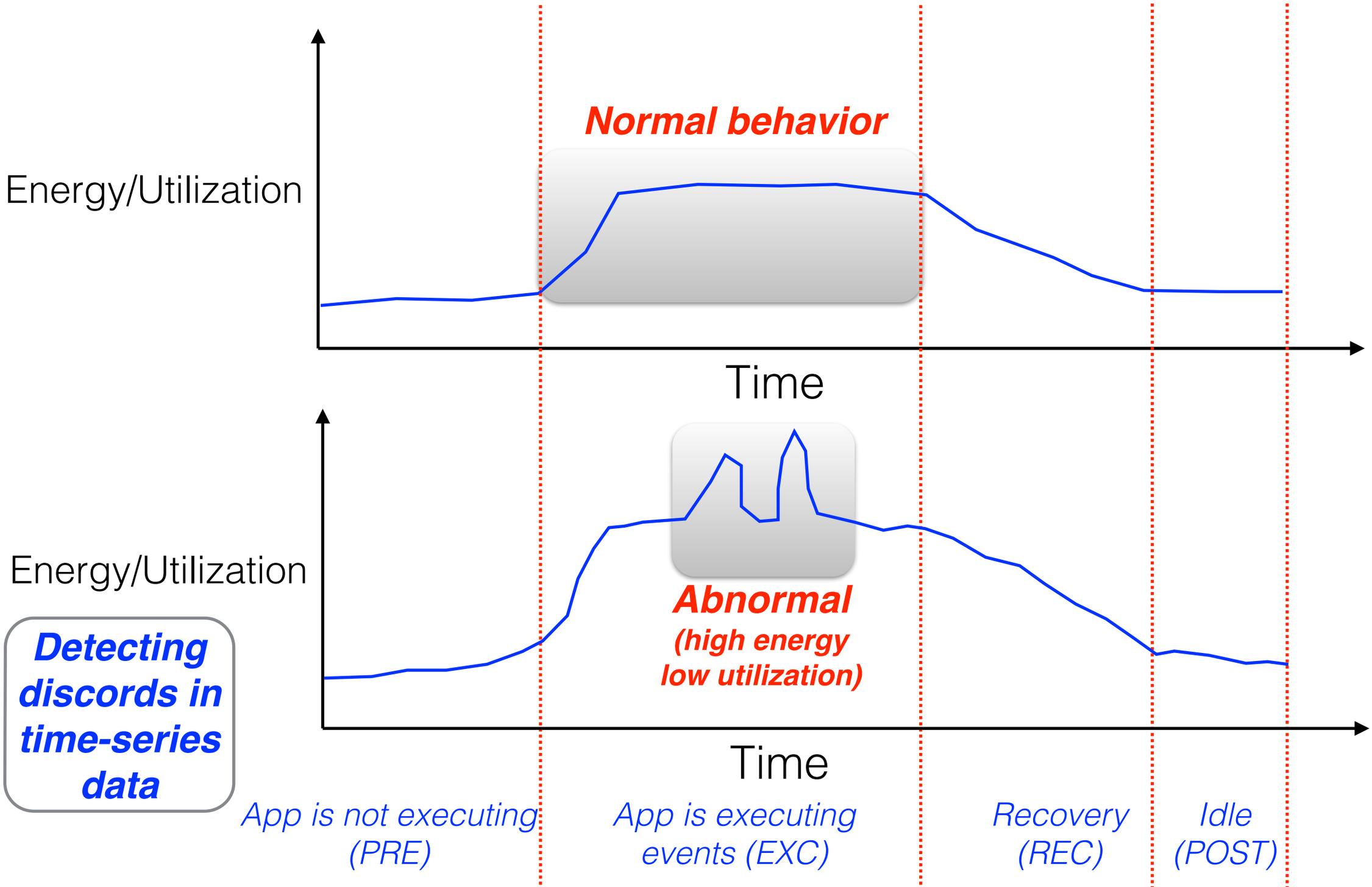
Energy consumption of different components is not even (GPS < CPU)

100% CPU does not consume same energy as GPS being on

Detecting Energy Bugs

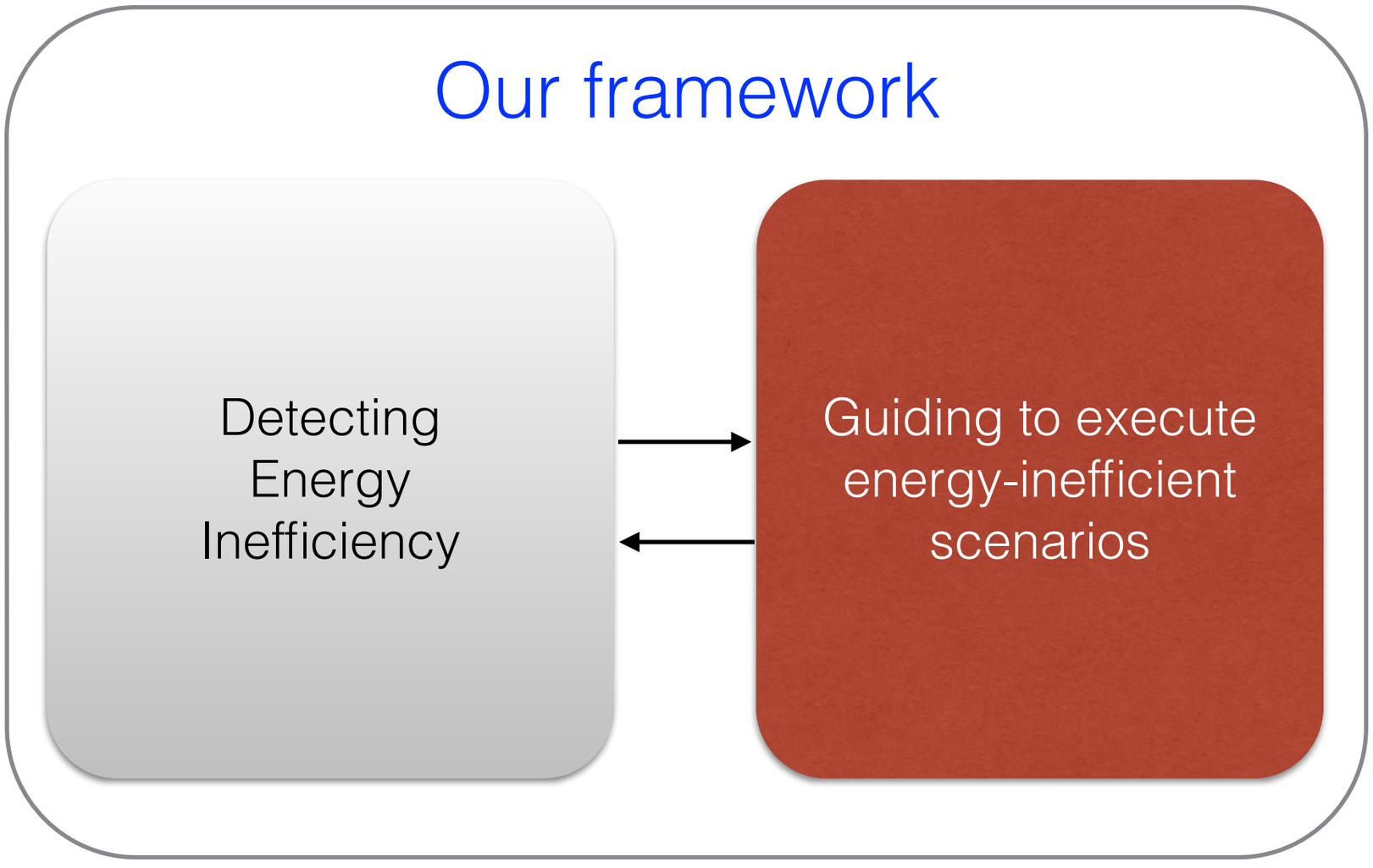


Detecting Energy Hotspots



Detecting discords in time-series data

Test Generation



Guided Exploration

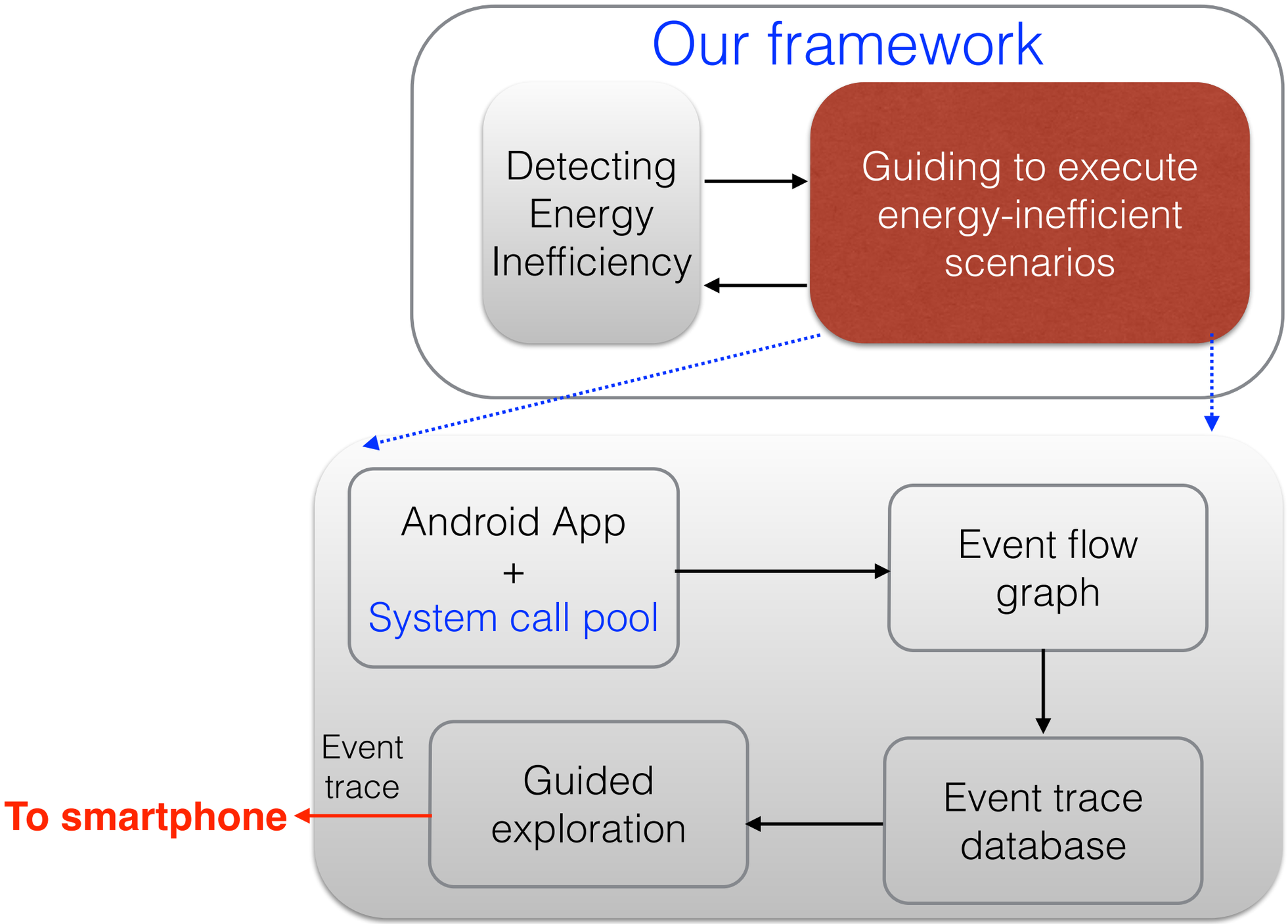
- Energy-inefficient execution
  - Which fragments are energy-inefficient?
  - What is an appropriate coverage metric?

A Broader Categorization

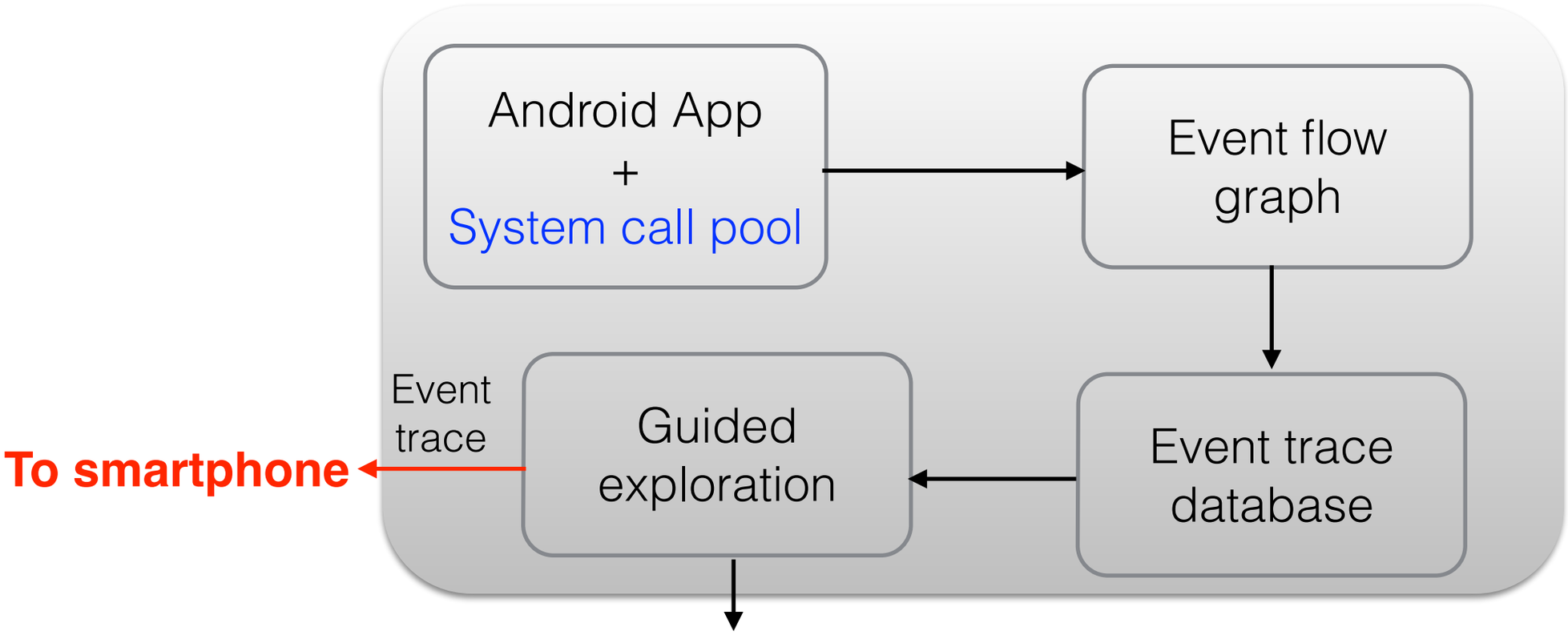
Cause/Source	Energy Bugs	Energy Hotspots
Hardware components	Resource leak	Suboptimal resource binding
Sleep state transition	Wakelock bug	Tail Energy hotspot
Background Service	Vacuous background service	Expensive background service
Defective Functionality	Immortality bug	Loop energy hotspot

Invoked via System Calls

Test Generation

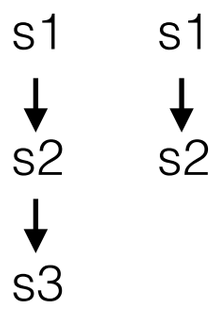


Test Generation



An ordering between trace T1 and T2

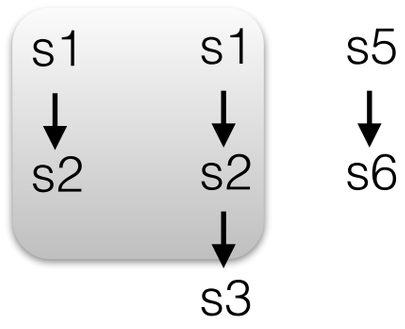
T1 > T2



More system calls

Buggy trace

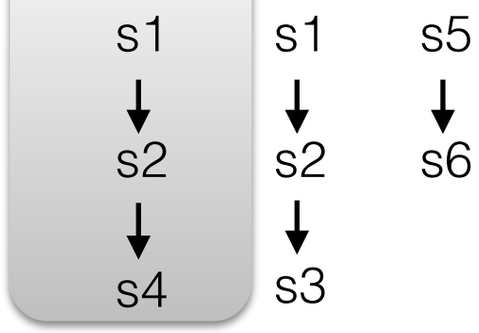
T1 > T2



Similarity with buggy trace

Executed trace

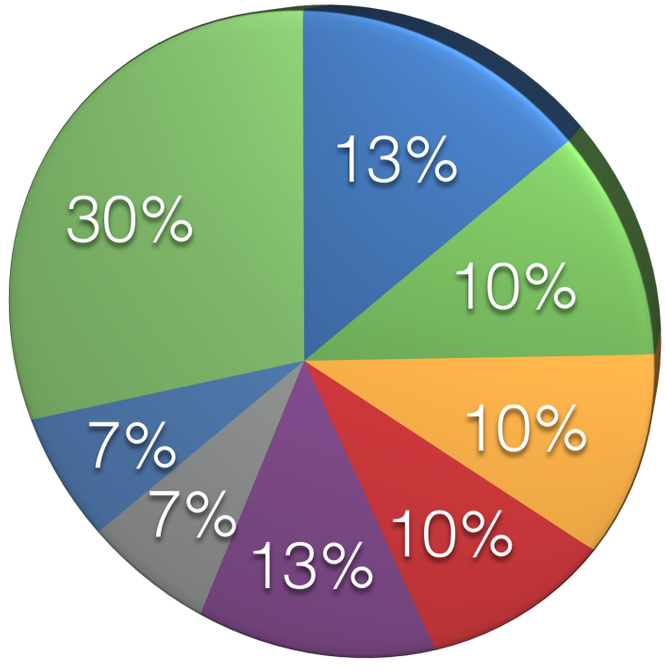
T1 < T2



Covering more system calls

Evaluation

- Travel/Transportation
- Lifestyle/Health
- Books/News
- Productivity
- Photography/Media
- Entertainment
- Puzzle
- Tools



Category of Android Apps Evaluated

Summary of Evaluation

App	Feasible traces	Energy Bugs	Energy Hotspots	Type	Reported before
Aripuca	502	Yes	No	Vacuous background service	No
Montreal Transit	64	No	Yes	Suboptimal resource binding and more	No
Sensor Test	2800	Yes	No	Immortality Bug	No
760 KFMB AM	26	Yes	Yes	Vacuous background service, suboptimal resource binding	No

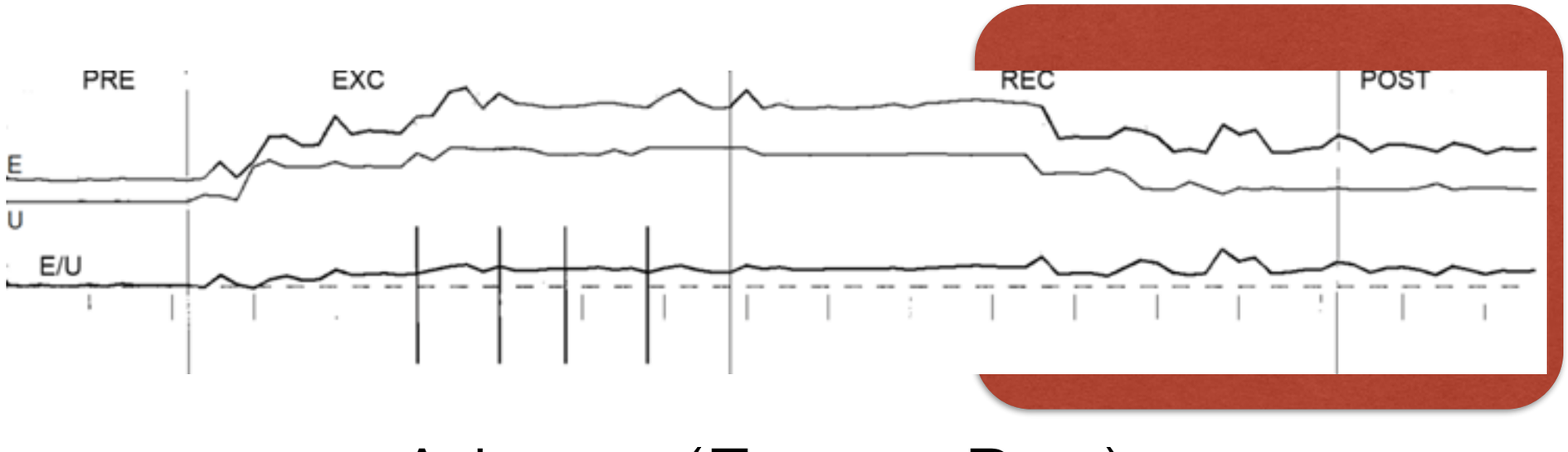
All Results are in the paper
(10 energy bugs and 3 energy hotspots found out of 30 tested apps)

Summary of Evaluation

App	System call	Statement	Lines of Code
Aagtl	100	21	11612
Android Battery Dog	100	17	463
Aripuca	100	15	4353
Kitchen Timer	100	30	1101
Montreal Transit	89	11	10925
NPR News	100	24	6513
OmniDroid	83	36	6130
Pedometer	100	56	849
Vanilla Music Player	86	20	4081

To cover all system calls, exploring only a small part of the program suffices
A substantial portion of the code is used for provide user feedback, compatibility over different OS

Case Studies



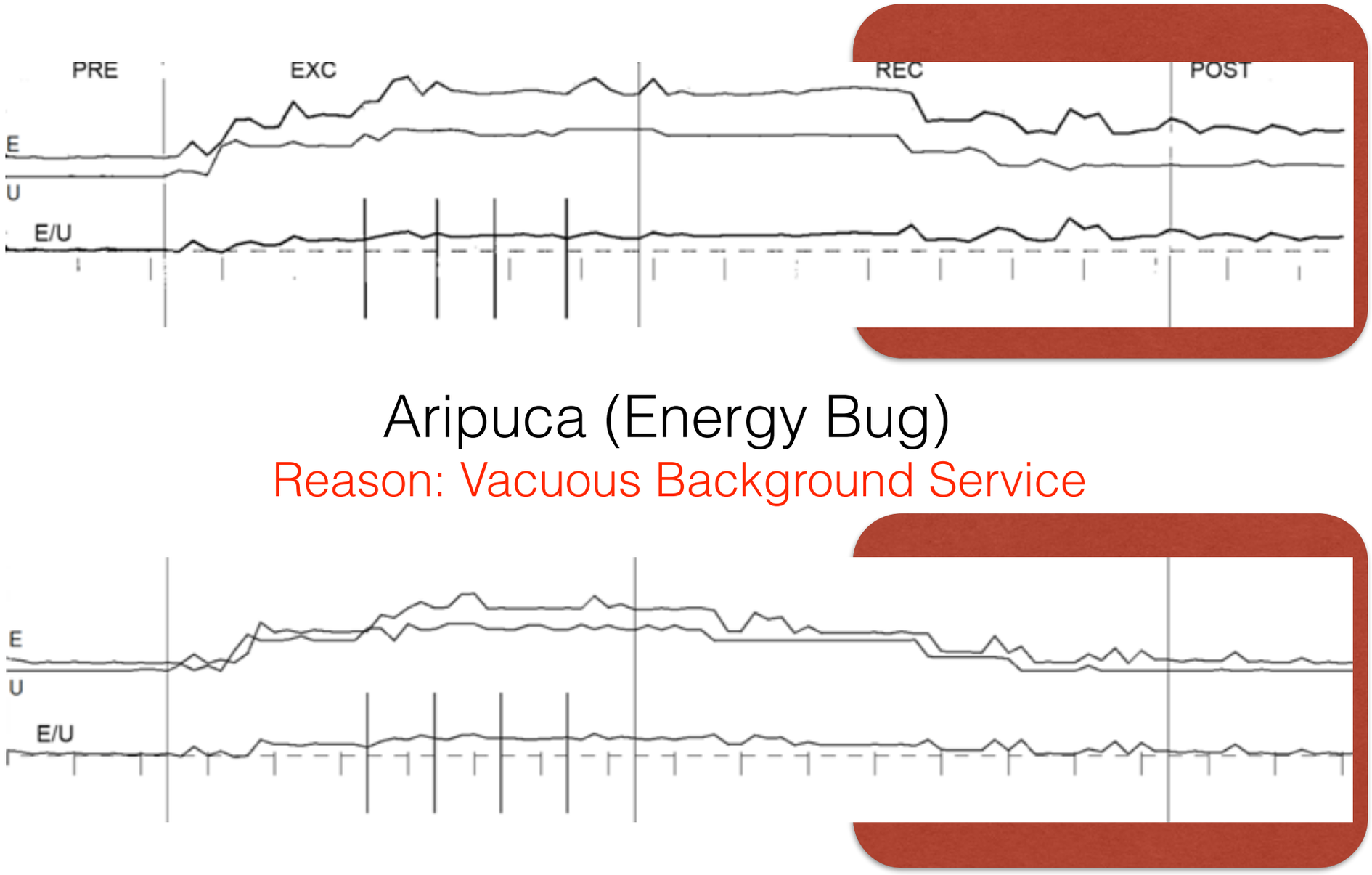
Aripuca (Energy Bug)

Reason: Vacuous Background Service

Fix:

```
serviceConnection.getService().stopLocationUpdates();
serviceConnection.getService().stopSensorUpdates();
```

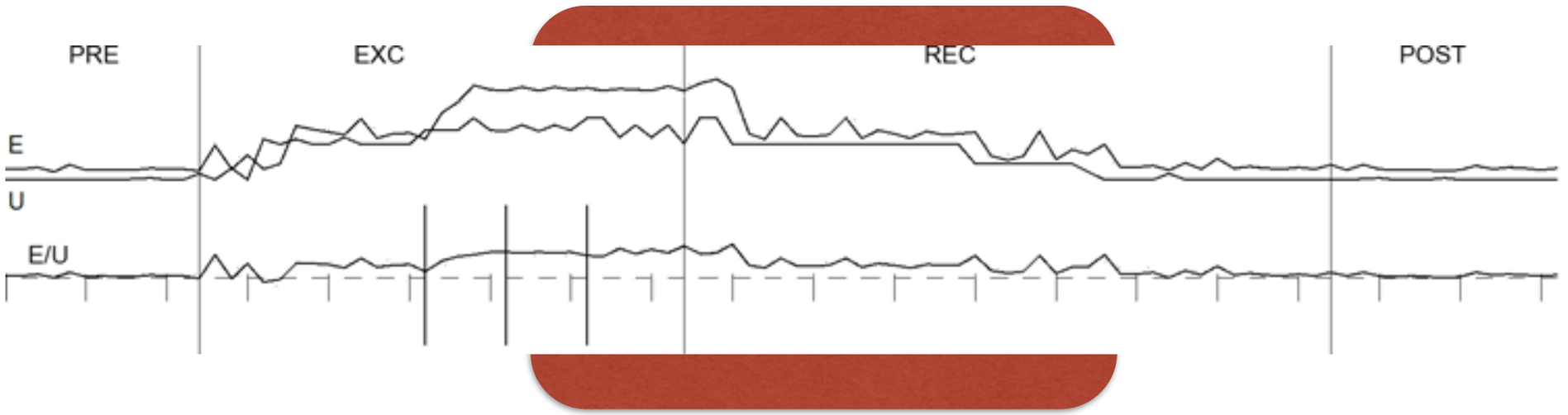
Case Studies



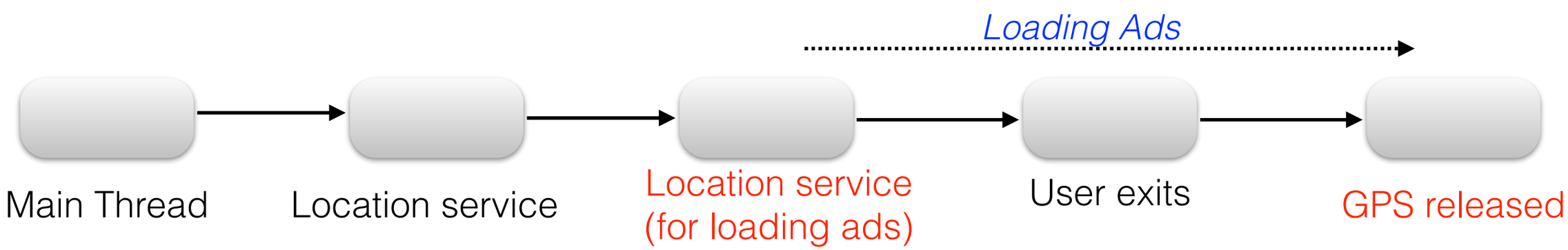
Aripuca (Energy Bug)

Reason: Vacuous Background Service

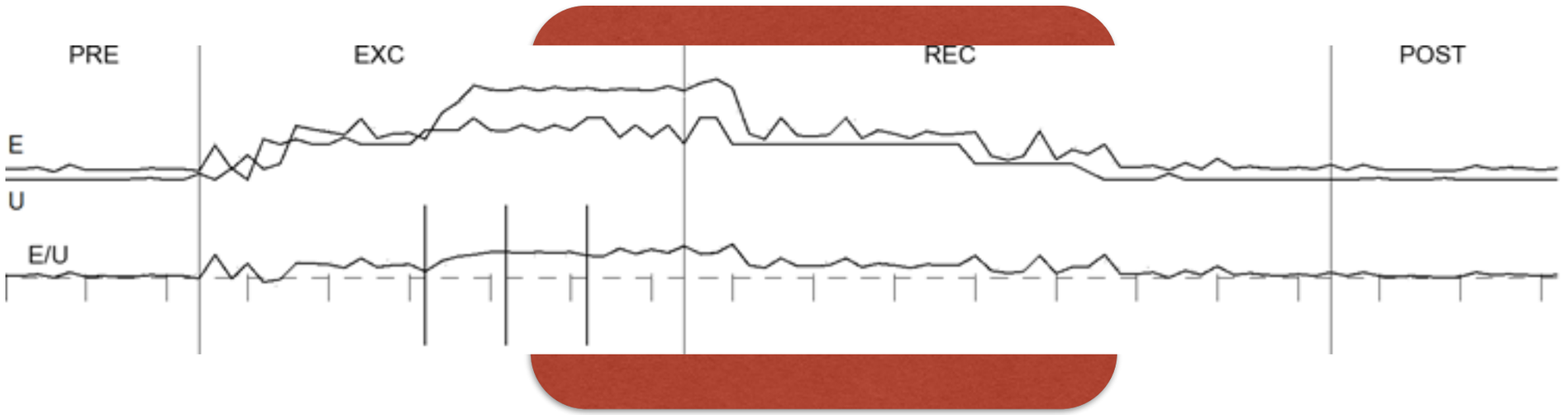
Case Studies



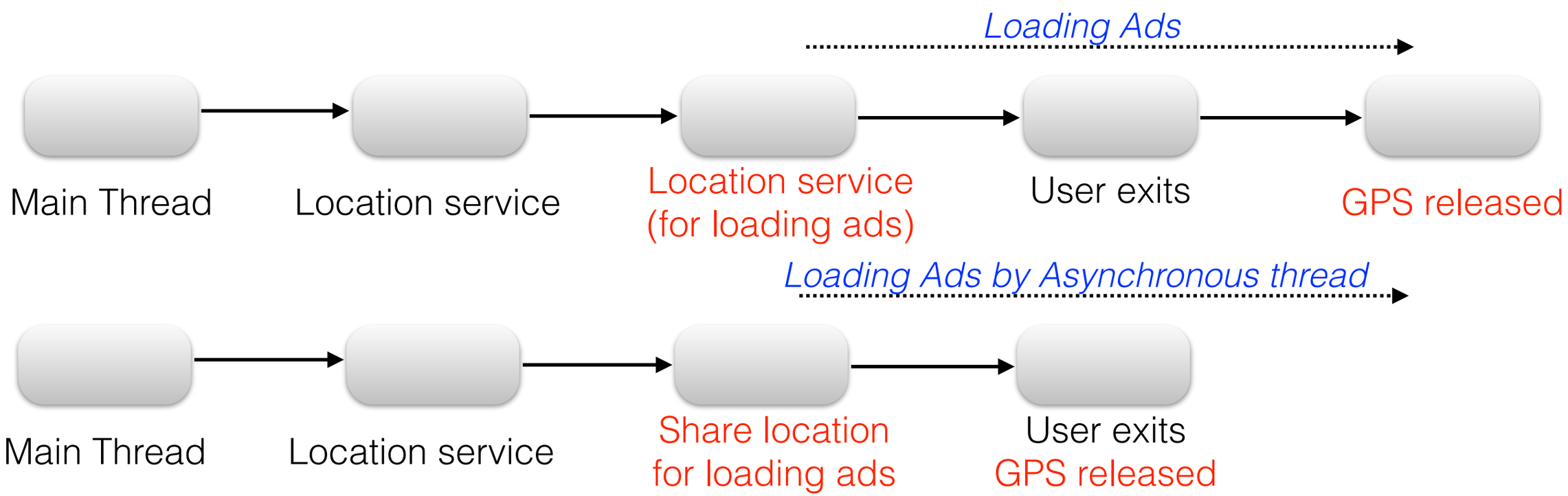
Montreal Transit (Energy hotspot for <5 sec)



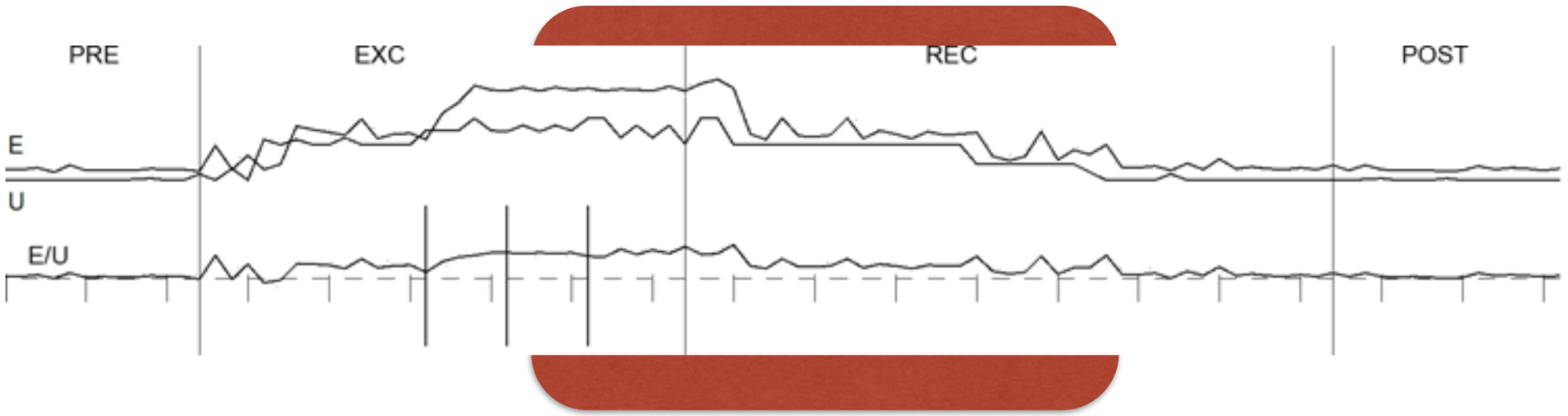
Case Studies



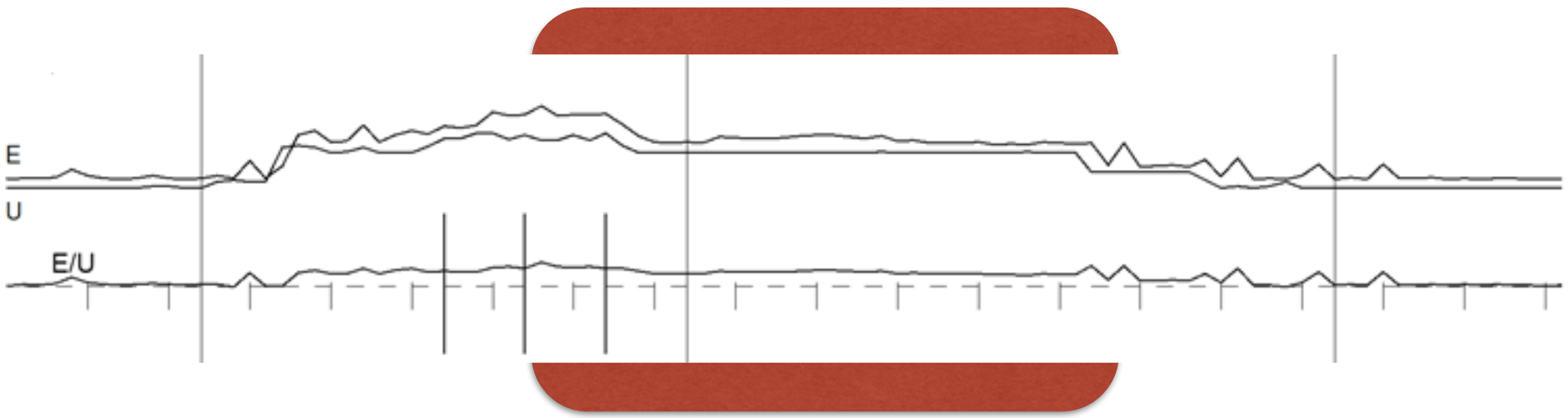
Montreal Transit (Energy hotspot for <5 sec)



Case Studies



Montreal Transit (Energy hotspot for <5 sec)



Montreal Transit (After fixing)

Summary

- Categorization of energy inefficiency
  - Energy bugs
  - Energy hotspots
- A guided exploration of event traces
  - Targeting system call coverage
- Evaluation with Android apps
  - Energy bugs and hotspots exist in several Android apps

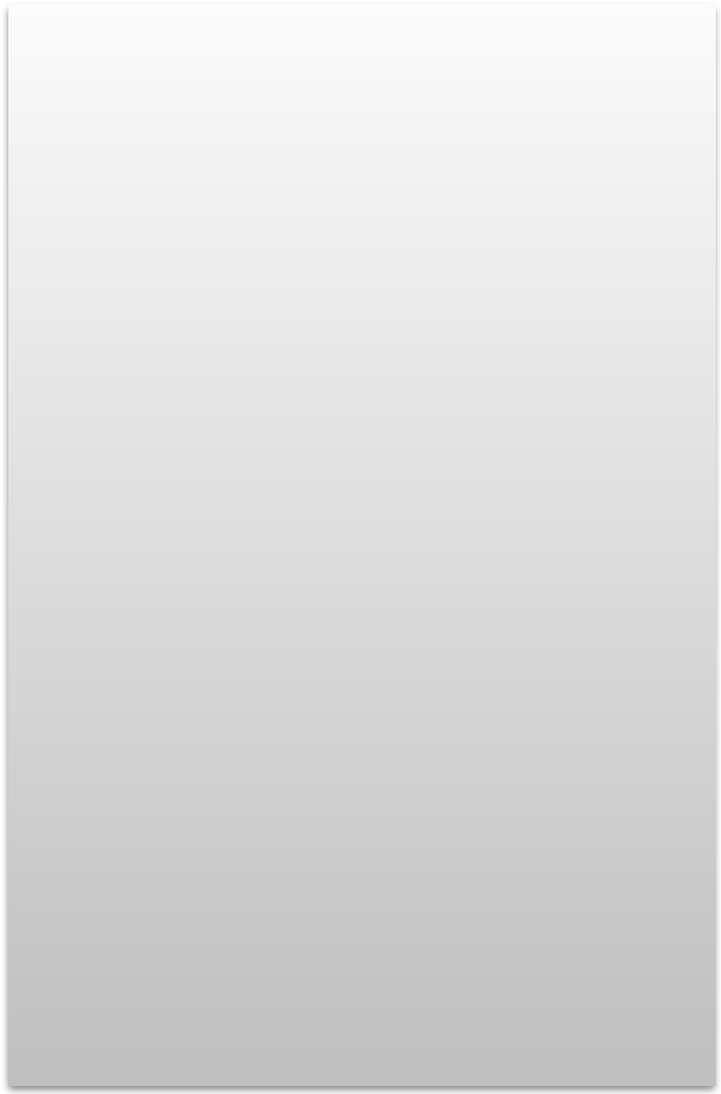
Is That All?

- Testing Non-functional Software Properties
  - Performance Testing
  - Energy Testing
- Far from being solved
  - Fresh look on the formal foundation of software testing
  - Automated debugging and fault localization
- *Information leaks via side channels (time, cache miss, power)*

Cache Side Channel

```
encrypt (message, key) {
 ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
mid[0][0] = lookup[key[0]][0];
mid[0][0] ^= lookup[key[1]][1];
 ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
}
```

Intermediate state of the encrypted message



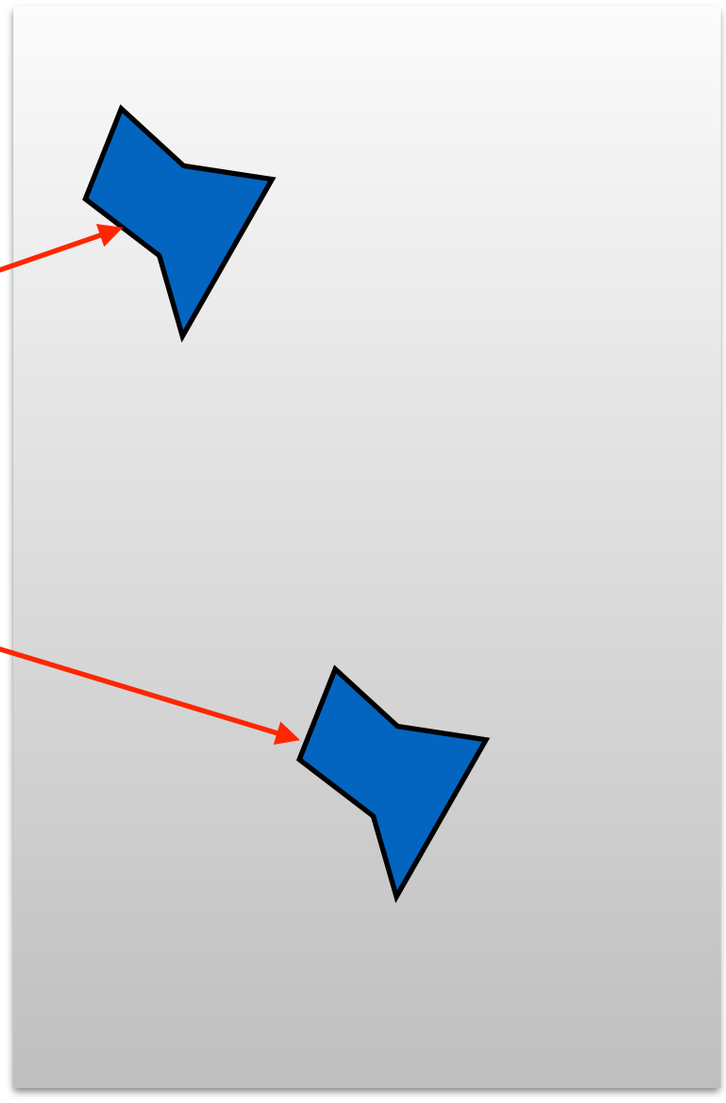
lookup Table
(in memory)

Cache Side Channel

```
encrypt (message, key) {
  ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
  mid[0][0] = lookup[key[0]][0];
  mid[0][0] ^= lookup[key[1]][1];
  ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
}
```

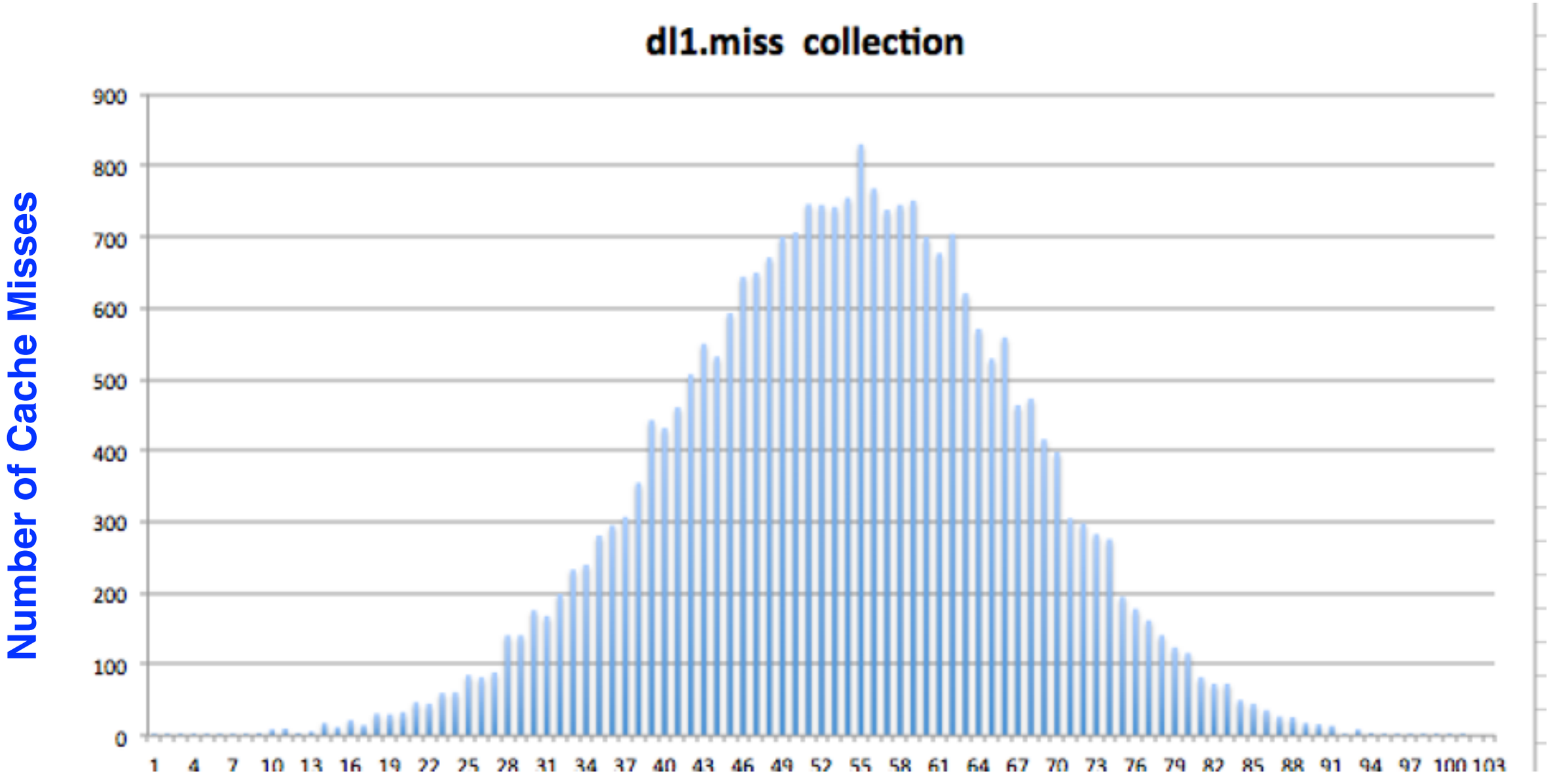
*Depends on the value of **Key***

Intermediate state of the encrypted message



lookup Table (in memory)

Cache Miss Distribution



- Abhijeet Banerjee, Sudipta Chattopadhyay and Abhik Roychoudhury. *Static Analysis Driven Cache Performance Testing*. IEEE Real-time Systems Symposium (**RTSS**), 2013 ***Best Paper Candidate***

- Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay and Abhik Roychoudhury. *Detecting Energy Bugs and Hotspots in Mobile Apps.* Foundation in Software Engineering (**FSE**), 2014

- Sudipta Chattopadhyay, Petru Eles and Zebo Peng. *Automated Software Testing of Memory Performance in Embedded GPUs.* International Conference on Embedded Software (**EMSOFT**), 2014