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 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!

Program
Cache Config.
Test Inputs

Program Profiling

Performance Issues

No guarantees for completeness

Vs

Program
Cache Config.

Our Framework

Performance Issues +
Symbolic Formula

Test Inputs
Key Idea

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

Stage I: Reduces the search space for exploration

Stage II: Explores the reduced search space & generate test cases

Non-functional properties encoded as assertions

Test Cases
Static Analysis

Program → Static Analysis → Cache

Classification of Memory Block:
- always hit (AH)
- persistent (PS)
- always miss (AM)
- not classified (NC)

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

RTSS 2013, Vancouver
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

Set 1
Set 2
Cache

RTSS 2013, Vancouver
Encode each thrashing scenario as an assertion at appropriate program location

Thrashing Sets
{m1, m2}
Generating Assertions

Direct-mapped cache

Conflicts in cache

Conflict due to m6

Conflict due to m5

No cache thrashing

assert (C_m5 <= 0 ∨ C_m6 <= 0)

x > 5

y > 12

y <= 12

x <= 5

m4

m7

m8

m9

m6

m5

m10

RTSS 2013, Vancouver
Dynamic Exploration

Exploration is performed to check the violation of Instrumented assertions.

No False Positives
Exploration by Greedy Strategy

\[(x \leq 5) \land (y \leq 12) \land (z \leq 1) \Rightarrow \alpha\text{ assertions checked}
\]

\[(x > 5) \land (y \leq 12) \land (z \leq 1) \land (y > 12) \Rightarrow 0\text{ additional assertions checked}
\]

\[(x \leq 5) \land (y \leq 12) \land (z > 1) \Rightarrow \beta\text{ assertions checked}
\]

Exploration performed using the Control Dependency Graph (CDG)
Test Generation

Results are generated in the format

\[ \langle \{m_1,m_2\} , x > 5 \rangle \]

Used to generate test cases

RTSS 2013, Vancouver
Recap

Instrumentation automatically adds assertions to the program. Cache analysis by static analysis reports violated assertions. Explore a path leading to assertions (symbolic exec). Test Suite includes:

- Always hit (AH)
- Persistent (PS)
- Always miss (AM)
- Not classified (NC)

CHMC (cache hit-miss classification) always hit (AH), persistent (PS), always miss (AM), not classified (NC). Assert m1 & m2 do not lead to thrashing.
Experiments

- Instruction Caches
- Data Caches

- RTS' 00 (Theiling et al)
- RTAS' 11 (Our work)
Evaluation

**Assertion Coverage (AC)**

\[
\text{Assertion Coverage (AC)} = \frac{\text{Unique assertions checked}}{\text{Unique assertions instrumented}} \times 100
\]

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

**Thrashing Potential (TP)**

\[
\text{Thrashing Potential (TP)} = \frac{\text{Unique assertions violated}}{\text{Unique assertions instrumented}} \times 100
\]

Gives an idea about the thrashing potential for a program, for a given cache configuration.
Results – Instruction Caches

- Papabench - AC
- Papabench - TP
- Nsichneu - AC
- Nsichneu - TP
- Jetbench - AC
- Jetbench - TP

Percentage vs. Time (300 seconds)
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

Smaller, Less associative cache — Lower cache thrashing — Fast access, lower power consumption — Big, highly associative cache

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

- 2 KB 1-Way
- 2 KB 2-Way
- 4 KB 2-Way
- 8 KB 2-Way
- 8 KB 4-Way

RTSS 2013, Vancouver
**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

- **Testing Functionality**
  - PLDI 2005, OSDI 2008

- **Profiling**
  - Not Complete

- **Partitioning I/P Space**
  - LCTES 2013
  - Requires manual effort
  - May have false positives

Our Work

- **Testing Performance**
  - Complete
  - Automated
  - No False Positives

RTSS 2013, Vancouver
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**