Design and Analysis of Time-Critical Systems
Cache Analysis and Predictability

Jan Reineke @ Saarland University

ACACES Summer School 2017
Fiuggi, Italy
An Analogy

Calvin and Hobbes by Bill Watterson
November 26, 1986
Can We Predict Software Behavior?

- **Security:**
  - Does this app leak my private data?
  - Are buffer overflow attacks possible?

- **Verification:**
  - Can this program crash?
  - Does it behave according to specification?

- **Timing Analysis:**
  - What is the maximum runtime of this program?
  - **Important Subproblems:**
    - Which address will be accessed?
    - **Will this memory access result in a cache hit?**
Prediction is Impossible!

Alan Turing: „The halting problem is undecidable.“

Henry Gordon Rice: „Any non-trivial property of programs is undecidable.“
Approximative Solutions

Program

```
// Perform the convolution.
for (int i=0; i<10; i++) {
    x[i] = a[i]*b[i-i];
    // Notify listeners.
    notify(x[i]);
}
```

Analysis for Property X

Yes
Don't know
No
Underapproximation: Testing

Test the program on a finite subset of its inputs:

all behaviours of the program

test cases

Undetected fault

Detected fault

“Program testing can be used to show the presence of bugs, but never to show their absence!”

Dijkstra (1970)
Sound Static Program Analysis (Abstract Interpretation)

Over-approximate all possible behaviors:

all behaviours of the program

Infeasible behaviours

Warning: False positive

All violations detected
Static Analysis in a Nutshell

Abstract Behaviors

Concrete Behaviors

Set of initial states + inputs
How to achieve „safe approximation“?

1. Every abstract state $s^\#$ represents a set $\text{conc}(s^\#)$ of concrete states:
How to achieve „safe approximation“?

2. *Local Consistency*:

„abstract successors“ over-approximate „concrete successors“

*may represent more states → imprecision*
How to achieve „safe approximation“?

Concrete Behaviors

Abstract Behaviors

safe approximation

Local Consistency

Local Consistency
Application: Static Cache Analysis

Cache: **Small**, but **fast** memory that buffers part of main memory

Goal of static cache analysis is to classify memory accesses as **guaranteed cache hits**
Concrete Cache States: Least-Recently-Used Replacement

**Notion of Age**

<table>
<thead>
<tr>
<th></th>
<th>0:</th>
<th>1:</th>
<th>2:</th>
<th>3:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

*most-recently-used*  
*least-recently-used*

<table>
<thead>
<tr>
<th></th>
<th>0:</th>
<th>1:</th>
<th>2:</th>
<th>3:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>B</td>
<td>F</td>
<td>D</td>
</tr>
</tbody>
</table>

*most-recently-used*  
*least-recently-used*
Concrete Cache Behavior: Least-Recently-Used Replacement

How do concrete cache states change upon memory accesses?

Access to C: "hit"

Access to E: "miss"
Brainstorming: Predicting Cache Hits

Want to soundly predict cache hits.

How to compactly represent sets of concrete states for this purpose?
Abstract “Must” Cache States

Compactly represent sets of concrete states:

- **0:** \{\}
- **1:** \{B\}
- **2:** \{A,C\}
- **3:** \{D\}

Upper bound on ages of memory blocks

Concrete states in which:
- B's age is at most 1,
- A's and C's age is at most 2,
- D's age is at most 3.
Abstract “Must” Cache States

Compactly represent sets of concrete states:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{B}</td>
<td>{A,C}</td>
<td>{D}</td>
</tr>
</tbody>
</table>

Upper bound on ages of memory blocks
Abstract “Must” Cache States

Compactly represent sets of concrete states:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

conc
Abstract “Must” Cache Behavior

How do abstract cache states change upon memory accesses?

Access to C: “definite hit”

Access to E: “potential miss”
Abstract "Must" Cache Behavior: Local Consistency

Access to C: "hit"

Access to C

conc
Integration with Pipeline Analysis

During microarchitectural analysis, **analysis states are pairs**:

(pipeline state, cache state)

Upon uncertainty, e.g. whether a cache hit or miss happens, "split".

If two analysis states agree on the pipeline state, the cache states are "joined".
Properties of the Join

- **Soundness**: Must not lose any concrete states:
  \[
  \text{conc}(A) \subseteq \text{conc}(A \cup B) \\
  \text{conc}(B) \subseteq \text{conc}(A \cup B)
  \]

- **Precision**: Want the “best” representation that is sound, i.e. the one that represents the fewest concrete states.
Join for Abstract “Must” Cache States

\[
\{A\} \cup \{C\} = \{C,A\}
\]

→ Take the **maximum** of the age bounds!
Example Analysis: A Loop

No hits can be predicted! Why?
Context-sensitive Analysis:
Virtual Loop Peeling (Unrolling)

The problem:
- The first iteration of a loop will always result in cache misses
- Similarly for the first execution of a recursive function

A solution:
- Distinguish the first iteration of a loop from others
- Distinguish function calls by calling context
Context-sensitive Analysis: Example

- Accesses to A and D are provably hits after the first iteration
- Accesses to B and C can still not be classified, but they can miss at most once
  ➔ Need Persistence Analysis

Note: Loop is only peeled virtually, i.e. during analysis not in the binary!
Brainstorming: Predicting Cache Misses

Want to soundly predict cache misses.

How to compactly represent sets of concrete states for this purpose?
Predicting Cache Misses: Abstract “May” Cache States

<table>
<thead>
<tr>
<th>0:</th>
<th>{A}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>{B, D}</td>
</tr>
<tr>
<td>2:</td>
<td>{C, E, F}</td>
</tr>
<tr>
<td>3:</td>
<td>{}</td>
</tr>
</tbody>
</table>

Concrete states in which:
- A’s age is at least 0,
- B’s and D’s age is at least 1, and
- C’s, E’s, and F’s age is at least 2.

How is this useful in WCET analysis?

Lower bound on ages of memory blocks

Blocks with a lower bound greater or equal to the cache size are guaranteed not to be in the cache.
Abstract “May” Cache Behavior

How do abstract cache states change upon memory accesses?

Access to C: “potential hit“

Access to E: “definite miss“

Notice subtle difference to “must” update!
Join for Abstract “May” Cache States

\[
\{A,C\} \cup \{C\} = \{A,C\}
\]

\[
\{F\} \cup \{E\} = \{F\}
\]

\[
\{D\} \cup \{A\} = \{D\}
\]

→ Take the minimum of the age bounds!
Precision of WCET analysis determined by amount of uncertainty
Uncertainty in cache analysis depends on replacement policy
Uncertainty in Cache Analysis

1. Initial cache contents unknown.
2. Different paths lead to these points.
3. Cannot resolve address of $z$.

$\Rightarrow$ Amount of uncertainty determined by ability to recover information
Cache Predictability Metrics

Evict

Fill

[seq: h, . . . , e, f, g, h]

Complete Uncertainty

Complete Certainty
Interpretation of Predictability Metrics

- **Evict:**
  - Number of accesses until any analysis can start predicting guaranteed cache misses

- **Fill:**
  - Number of accesses until any analysis can precisely predict any access as hit or miss

→ The two metrics bound the precision of any static cache analysis. Can thus serve as benchmark for analyses.
Popular Cache Replacement Policies

- Least-Recently-Used (LRU) used in Intel Pentium I, MIPS 24K/34K, AMD Opteron
- First-In First-Out (FIFO) used in Motorola PowerPC 56x, Intel XScale, ARM9, ARM11
- Not Most-Recently-Used (NMRU) used in Intel Itanium
- Pseudo-LRU (PLRU) used in Intel Pentium II-IV, Intel Core, Intel Xeon, PowerPC 75x
Observation: LRU Replacement is „Robust“

→ Predictability Metrics:
\[\text{Evict}(k) = \text{Fill}(k) = k\]

where \( k = \text{associativity} \)
Predictability Metrics: FIFO

- Like LRU in the miss case
- But: “Ignores” hits

In the worst case \( k-1 \) hits and \( k \) misses: \((k = \text{associativity})\)

\[ \text{Evict}(k) = 2k-1 \]

Another \( k \) accesses to obtain complete knowledge:

\[ \text{Fill}(k) = 3k-1 \]
Predictability Metrics: PLRU

Tree bits point to block to be replaced

Accesses „rejuvenate“ neighborhood
- Active blocks keep their (inactive) neighborhood in the cache

Analysis yields:
- $\text{Evict}(k) = \frac{k}{2} \log_2 k + 1$
- $\text{Fill}(k) = \frac{k}{2} \log_2 k + k - 1$
#### Evaluation of Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Evict(k)</th>
<th>Fill(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>(k)</td>
<td>(k)</td>
</tr>
<tr>
<td>FIFO</td>
<td>(2k-1)</td>
<td>(3k-1)</td>
</tr>
<tr>
<td>NMRU</td>
<td>(2k-2)</td>
<td>(3k-4)</td>
</tr>
<tr>
<td>PLRU</td>
<td>(k/2 \log k + 1)</td>
<td>(k/2 \log k + k-1)</td>
</tr>
</tbody>
</table>

1. LRU is **optimal** w.r.t. metrics → Use LRU when you have the choice.

2. How to build static analyses that are optimal w.r.t. metrics? → [SIGMETRICS, LCTES, SAS, WCET, ECRTS]
Conclusions

- Cache Analysis for Least-Recently-Used:
  - Efficiently represents sets of cache states by bounding the age of memory blocks
    - From above: “must” analysis
    - From below: “may” analysis
  - Requires context sensitivity for precision

- Cache Predictability Metrics:
  - Equate predictability with “forgetfulness”
  - LRU is particularly “forgetful”
Literature

Abstract Interpretation:

LRU:
- Ferdinand, Wilhelm: Efficient and precise cache behavior prediction for real-time systems, Real-Time Systems, 1999

Predictability:

Other Policies:

FIFO:
- Grund, Reineke: Precise and efficient FIFO-replacement analysis based on static phase detection, In: Proceedings ECRTS, 2010

Persistence:

NMRU:
- Guan, Lv, Yi, Yu: WCET analysis with MRU cache: challenging LRU for predictability, ACM TECS, 2014

Survey:
- Lv, Guan, Reineke, Wilhelm, Yi: A survey on static cache analysis for real-time systems, LITES 3(1), 2016