Verification of Real-Time Systems
Cache Persistence Analysis

Jan Reineke

Department of Computer Science
Saarland University
Saarbrücken, Germany

Advanced Lecture, Summer 2015
Recap: Context-Sensitive Analysis

- **Problem:**
  - The first iteration of a loop will always result in cache misses.
  - Similarly for the first execution of a function.
- **Solution:**
  - Virtually Unroll Loops: Distinguish the first iteration from others
  - Distinguish function calls by calling context.

Virtually unrolling the loop once:
- Accesses to A and D are provably hits after the first iteration
- Accesses to B and C can still not be classified. Within each execution of the loop, they may only miss once.

→ Persistence Analysis
Notion of Persistence

- Intuition:
  “Block $b$ is persistent if it can only cause one cache miss in any execution.”

- What is an appropriate concrete semantics that captures this property?
Trace Collecting Semantics

A semantics that consists of cache states is not sufficient!

Need to consider sequences of states and the events (hits/misses to particular cache blocks) that occur between them.

→ Trace collecting semantics
Trace Collecting Semantics: Domain

Trace collecting semantics of program $P$: $Col(P) \subseteq \text{Traces}$, where $\text{Traces}$ denotes the set of all alternating sequences of states and events.
Trace Collecting Semantics: Domain

Trace collecting semantics of program $P$: $Col(P) \subseteq Traces$,
where $Traces$ denotes the set of all alternating sequences of states and events.
Trace Collecting Semantics: Domain

*Trace collecting semantics* of program $P$: $\text{Col}(P) \subseteq \text{Traces}$,
where $\text{Traces}$ denotes the set of all alternating sequences of states and events.

What are *states* and *events* here?

- **States**: cache
- **Events**: access, hit, miss
Trace Collecting Semantics: Definition

A program \( P = (\Sigma, \mathcal{I}, \mathcal{E}, \mathcal{T}) \) consists of the following components:
- \( \Sigma \) - a set of program states
- \( \mathcal{I} \subseteq \Sigma \) - a set of initial states
- \( \mathcal{E} \) - a set of events
- \( \mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma \) - a transition relation
Trace Collecting Semantics:
Definition

A program $P = (\Sigma, \mathcal{I}, \mathcal{E}, \mathcal{T})$ consists of the following components:
- $\Sigma$ - a set of *program states*
- $\mathcal{I} \subseteq \Sigma$ - a set of *initial* states
- $\mathcal{E}$ - a set of *events*
- $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ - a *transition relation*

The trace collecting semantics can be formally defined as the least fixed point of the *next* operator containing $\mathcal{I}$:

$$Col(P) = \text{lfp}_\mathcal{I} \text{next}$$

where *next* describes the effect of one computation step:

$$\text{next}(S) = \{ t.\sigma_n e_n \sigma_{n+1} \mid t.\sigma_n \in S \land (\sigma_n, e_n, \sigma_{n+1}) \in \mathcal{T} \}$$
Trace Collecting Semantics: Instantiation for Caches

- States $\Sigma = \mathcal{M} \times \mathcal{C}$, where
  - $\mathcal{M}$ are the logical memory states, and
  - $\mathcal{C}$ are the cache states.

- Initial states $\mathcal{I} = \mathcal{I}_\mathcal{M} \times \mathcal{I}_\mathcal{C}$.

- Events $\mathcal{E} = \mathcal{E}_\mathcal{M} \times \mathcal{E}_\mathcal{C}$, where
  - $\mathcal{E}_\mathcal{M} = \mathcal{B} \cup \{\perp\}$ are the memory blocks the program may access, and
  - $\mathcal{E}_\mathcal{C} = \{\text{hit}, \text{miss}, \perp\}$ are the “cache events”.

- Transition relation $\mathcal{T}$:

  \[
  \mathcal{T} = \left\{ ((m, c), (e_m, e_c), (m', c')) \mid m' = \text{upd}_\mathcal{M}(m) \land e_m = \text{eff}_\mathcal{M}(m) \land c' = \text{upd}_\mathcal{C}(c, e_m) \land e_c = \text{eff}_\mathcal{C}(c, e_m) \right\},
  \]
Trace Collecting Semantics: Instantiation for Caches

- States $\Sigma = \mathcal{M} \times \mathcal{C}$, where
  - $\mathcal{M}$ are the logical memory states, and
  - $\mathcal{C}$ are the cache states.
- Initial states $\mathcal{I} = \mathcal{I}_M \times \mathcal{I}_C$.
- Events $\mathcal{E} = \mathcal{E}_M \times \mathcal{E}_C$, where
  - $\mathcal{E}_M = \mathcal{B} \cup \{\bot\}$ are the memory blocks the program may access, and
  - $\mathcal{E}_C = \{\text{hit, miss, } \bot\}$ are the “cache events”.
- Transition relation $\mathcal{T}$:

\[
\mathcal{T} = \{(m, c), (e_m, e_c), (m', c')) | m' = \text{upd}_M(m) \wedge e_m = \text{eff}_M(m) \\
\quad \wedge c' = \text{upd}_C(c, e_m) \wedge e_c = \text{eff}_C(c, e_m) \},
\]

What are $\text{upd}_M$, $\text{eff}_M$, $\text{upd}_C$, and $\text{eff}_C$?
Notions of Persistence in Terms of Trace Collecting Semantics

There are at least three notions in the literature:

\[\text{persistent}(P, b) := \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \text{Col}(P) : \]
\[\forall i, j : (e_i = e_j = (b, \text{miss})) \Rightarrow i = j.\]

\[\text{firstmiss}(P, b) := \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \text{Col}(P) : \]
\[\forall i : (e_i = (b, \text{miss})) \Rightarrow \exists j < i : (e_j \neq (b, \text{hit}) \land e_j \neq (b, \text{miss})).\]

\[\text{noeviction}(P, b) := \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \text{Col}(P) : \]
\[\forall i : (e_i = (b, \text{miss})) \Rightarrow \forall j > i : (b \in \sigma_j).\]
Notions of Persistence in Terms of Trace Collecting Semantics

There are at least three notions in the literature:

\[
persistent(P, b) := \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \text{Col}(P) : \\
\forall i, j : (e_i = e_j = (b, \text{miss})) \Rightarrow i = j.
\]

\[
\text{firstmiss}(P, b) := \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \text{Col}(P) : \\
\forall i : (e_i = (b, \text{miss})) \Rightarrow \forall j < i : (e_j \neq (b, \text{hit}) \land e_j \neq (b, \text{miss})).
\]

\[
\text{noeviction}(P, b) := \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \text{Col}(P) : \\
\forall i : (e_i = (b, \text{miss})) \Rightarrow \forall j > i : (b \in \sigma_j).
\]

Is one notion weaker/stronger than the others?
Abstractions of the Trace Collecting Semantics

The *trace collecting semantics* is not computable. → Need to apply abstractions.

Generic approach:

- Associate abstractions of sets of “cache traces” with program points.
- Need good abstractions for sets of “cache traces” that allow to predict persistence.
Generic Approach for Instruction Caches

“Abstract sticky cache trace collecting semantics” associates abstraction of “cache traces” with program points:

$$\text{StickyCol}(P_{\text{ins}}): \mathcal{L} \rightarrow \widehat{C_{\text{tr}}}.$$ 

Can be defined as least fixed point of $\text{next}_{\text{ins}}$:

$$\text{StickyCol}(P_{\text{ins}}) = \text{lfp}_{\text{Init}} \text{next}_{\text{ins}},$$

where $\text{next}_{\text{ins}}$ is defined as follows:

$$\text{next}_{\text{ins}}(\hat{S}) = \lambda l' \in \mathcal{L}. \big\{ \text{upd}_{\text{tr}}(\hat{S}(l), b) \mid b = \text{eff}_{\mathcal{L}}(l, l') \big\}.$$
Abstractions for Cache Traces

1. “Set-wise conflict counting”: Determines for each cache set the set of memory blocks accessed that map to this set

2. “Block-wise conflict counting”: Determines for each memory block the set of conflicting memory blocks accessed since the last access the the block itself

3. “Conditional must-analysis”: Determines for each memory block its maximal age under the condition that the block has already been accessed.
Set-wise conflict counting
(Here: for fully-associative cache)

Domain: \( \hat{CS} := \mathcal{P}(B) \) (sets of memory blocks)

Update: \( \text{upd}(cs, b) := cs \cup \{b\} \)

Join: \( cs_1 \cup cs_2 := cs_1 \cup cs_2 \)

Concretization: \( \gamma(cs) := \)

Classification: \( \text{persistent}(cs, b) := \)
Set-wise conflict counting
(Here: for fully-associative cache)

Domain: \( \hat{CS} := \mathcal{P}(\mathcal{B}) \) (sets of memory blocks)
Update: \( \hat{udp}(cs, b) := cs \cup \{b\} \)
Join: \( cs_1 \sqcup cs_2 := cs_1 \cup cs_2 \)

Concretization: \( \gamma(cs) := \{ c_1(b_1, h_i), c_2 \ldots c_r \mid \forall i < n : b_i \in cs \land c_{i+1} = udp(c_i, b_i) \} \)

Classification: \( \text{persistent}(cs, b) := \)
Set-wise conflict counting
(Here: for fully-associative cache)

Domain: \( \widehat{CS} := \mathcal{P}(B) \) (sets of memory blocks)
Update: \( \widehat{upd}(cs, b) := cs \cup \{b\} \)
Join: \( cs_1 \cup cs_2 := cs_1 \cup cs_2 \)

Concretization: \( \gamma(cs) := \{c_1(b_1, h_i)c_2 \ldots c_n \mid \forall i < n: b_i \in cs \land c_{i+1} = upd(c_i, b_i)\} \)

Classification: \( \widehat{persistent}(cs, b) := |cs| \leq k \)
Set-wise conflict counting
(Here: for fully-associative cache)

Domain: \( \widehat{CS} := \mathcal{P}(B) \) (sets of memory blocks)
Update: \( \widehat{upd}(cs, b) := cs \cup \{b\} \)
Join: \( cs_1 \cup cs_2 := cs_1 \cup cs_2 \)

Concretization: \( \gamma(cs) := \{c_1(b_1, h_1)c_2 \ldots c_n | \forall i < n : b_i \in cs \land c_{i+1} = upd(c_i, b_i)\} \)

Classification: \( \widehat{persistent}(cs, b) := |cs| \leq k \)

How to lift classification to all program points?
Set-wise conflict counting: Example 1

Which blocks are determined to be persistent for associativity 4?
What about associativity 3?
Block-wise conflict counting
(Here: for fully-associative cache)

Domain: \( \overline{BCS} := \mathcal{B} \rightarrow \mathcal{P}(\mathcal{B}) \)

Update: \( \overline{upd}(bcs, b) := \lambda b'. \begin{cases} \emptyset & : b' = b \\ bcs(b') \cup \{b\} & : \text{else} \end{cases} \)

Join: \( bcs_1 \sqcup bcs_2 := \lambda b : bcs_1(b) \cup bcs_2(b) \)

Concretization: \( \gamma(bcs) := \)

Classification: \( \overline{\text{persistent}}(cs, b) := \)
Block-wise conflict counting
(Here: for fully-associative cache)

Domain: \[ \widehat{BCS} := B \rightarrow \mathcal{P}(B) \]

Update: \[ \widehat{upd}(bcs, b) := \lambda b'. \begin{cases} \emptyset & : b' = b \\ bcs(b') \cup \{ b \} & : \text{else} \end{cases} \]

Join: \[ bcs_1 \sqcup bcs_2 := \lambda b : bcs_1(b) \cup bcs_2(b) \]

Concretization: \[ \gamma(bcs) := \{ c_1(b_1, h_i)c_2 \ldots c_n \mid \forall i < n : \]
\[ c_{i+1} = \widehat{upd}(c_i, b_j) \land (\forall j, i < j \leq n : b_i \neq b_j) \rightarrow \]
\[ (\forall j, i < j \leq n : b_j \in bcs(b_j)) \} \]

Classification: \[ \widehat{persistent}(cs, b) := \]
Block-wise conflict counting
(Here: for fully-associative cache)

Domain: \( \widehat{BCS} := \mathcal{B} \rightarrow \mathcal{P}(\mathcal{B}) \)

Update: \( \widehat{upd}(bcs, b) := \lambda b'. \begin{cases} \emptyset & : b' = b \\ bcs(b') \cup \{b\} & : \text{else} \end{cases} \)

Join: \( bcs_1 \sqcup bcs_2 := \lambda b : bcs_1(b) \cup bcs_2(b) \)

Concretization: \( \gamma(bcs) := \{ c_1(b_1, h_i)c_2 \ldots c_n | \forall i < n : c_{i+1} = upd(c_i, b_i) \wedge (\forall j, i < j \leq n : b_i \neq b_j) \rightarrow (\forall j, i < j \leq n : b_j \in bcs(b_j)) \} \)

Classification: \( \widehat{\text{persistent}}(cs, b) := |bcs(b)| < k \)
Block-wise conflict counting: Example

Which blocks are determined to be persistent for associativity $\beta$ under block- and set-wise conflict counting?
Block-wise conflict counting: Example 2

Which blocks are determined to be persistent for associativity 3 under block- and set-wise conflict counting?
Conditional Must Analysis
(Here: for fully-associative cache)

Domain: \( \widehat{CM} := B \rightarrow \{0, \ldots, k\} \)

Update: \( \widehat{upd}(cm, b) := \lambda b'. \begin{cases} 0 : b' = b \\ cm(b) + 1 : \text{else} \end{cases} \)

Join: \( cm_1 \sqcup cm_2 := \lambda b : \max\{cm_1(b), cm_2(b)\} \)

Concretization: \( \gamma(cm) := \)

Classification: \( \widehat{persistent}(cm, b) := \)
Conditional Must Analysis
(Here: for fully-associative cache)

Domain: \( \mathcal{CM} := \mathcal{B} \rightarrow \{0, \ldots, k\} \)

Update: \( \overline{\text{upd}}(cm, b) := \lambda b'. \begin{cases} 0 & : b' = b \\ cm(b) + 1 & : \text{else} \end{cases} \)

Join: \( cm_1 \sqcup cm_2 := \lambda b : \max\{cm_1(b), cm_2(b)\} \)

Concretization: \( \gamma(cm) := \{c_1(b_1, h_i)c_2 \ldots c_n | \forall i < n : c_{i+1} = \text{upd}(c_i, b_i) \land \text{age}(cm, b_i) \leq cm(b_i)\} \)

Classification: \( \overline{\text{persistent}}(cm, b) := \)
Conditional Must Analysis
(Here: for fully-associative cache)

Domain: \[ \widehat{CM} := \mathcal{B} \rightarrow \{0, \ldots, k\} \]

Update: \[ \widehat{upd}(cm, b) := \lambda b'. \begin{cases} 
0 & : b' = b \\
cm(b) + 1 & : \text{else}
\end{cases} \]

Join: \[ cm_1 \sqcup cm_2 := \lambda b : \max\{cm_1(b), cm_2(b)\} \]

Concretization: \[ \gamma(cm) := \{c_1(b_1, h_i)c_2 \ldots c_n | \forall i < n : c_{i+1} = \widehat{upd}(c_i, b_i) \land \text{age}(cm, b_i) \leq cm(b_i)\} \]

Classification: \[ \widehat{\text{persistent}}(cm, b) := cm(b) < k \]
Conditional Must Analysis
(Here: for fully-associative cache)

Domain: \( \widehat{CM} := \mathcal{B} \to \{0, \ldots, k\} \)

Update: \( \widehat{upd}(cm, b) := \lambda b'. \begin{cases} 0 & : b' = b \\ cm(b) + 1 & : \text{else} \end{cases} \)

Join: \( cm_1 \sqcup cm_2 := \lambda b : \max\{cm_1(b), cm_2(b)\} \)

Concretization: \( \gamma(cm) := \{c_1(b_1, h_i)c_2 \cdots c_n \mid \forall i < n : c_{i+1} = upd(c_i, b_i) \land age(cm, b_i) \leq cm(b_i)\} \)

Classification: \( \overline{\text{persistent}}(cm, b) := cm(b) < k \)

What should be the initial value for this domain? \( \text{?} \)
Conditional Must-Analysis: Example Revisited

Which blocks are determined to be persistent for associativity 3 under conditional must-analysis? $A, C$

Is the conditional must-analysis strictly better than the other two? $\neg$
Refinements of the Persistence Notion

The persistence notion can be refined in several directions:

- A block may be persistent during the execution of an inner loop, but not in the execution of the program as a whole.
  - Classify blocks to be persistent in particular *program scopes.*
Refinements of the Persistence Notion

The persistence notion can be refined in several directions:

- A block may be persistent during the execution of an inner loop, but not in the execution of the program as a whole. → Classify blocks to be persistent in particular program scopes.
- Accesses to the same block in different program locations may have different “persistence properties”. A per-location classification can exploit this. Example:

![Diagram](image)

At associativity 2, the access to B on the left may only miss once, whereas the access to B on the right will miss every time.
Summary

Notion of Persistence
- Persistence can only be captured by a *trace semantics*.
- Can be refined by focusing on particular accesses and program scopes.

Persistence Analysis
- Based on abstractions of cache/memory access traces.
- For LRU the abstraction needs to answer the following question: How many distinct memory blocks have been accessed since the first access to a particular block?