# Verification of Real-Time Systems
# Cache Persistence Analysis

Jan Reineke

Department of Computer Science
Saarland University
Saarbrücken, Germany

Advanced Lecture, Summer 2015

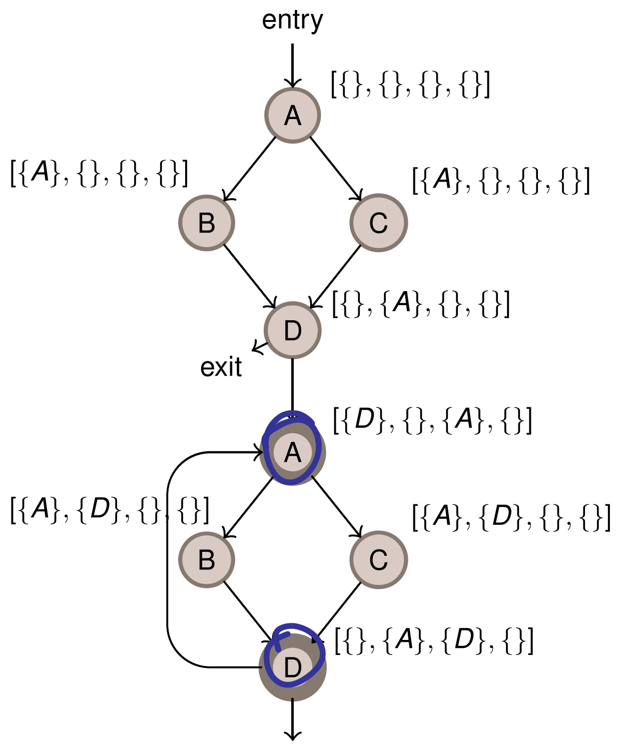
# Recap: Context-Sensitive Analysis

- 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.

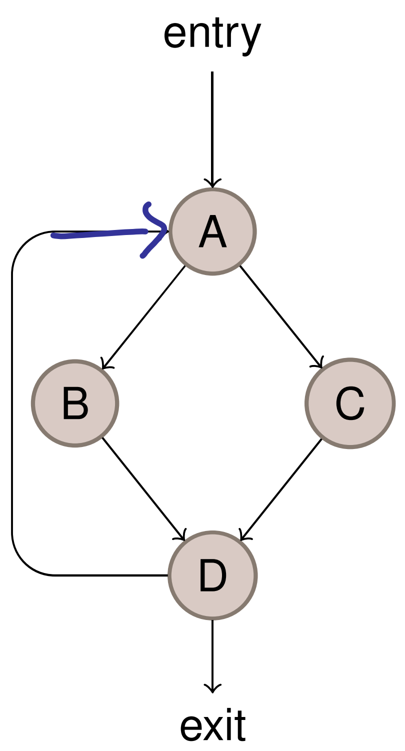
  $\longrightarrow$ Persistence Analysis



# Notion of Persistence

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- 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.

$\longrightarrow$ Trace collecting semantics

# 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 $P$: $Col(P) \subseteq Traces$,

where *Traces* denotes the set of all alternating sequences of states and events.

# Trace Collecting Semantics: Domain

SAARLAND UNIVERSITY
COMPUTER SCIENCE

*Trace collecting semantics* of program $P$: $Col(P) \subseteq Traces$,

where *Traces* denotes the set of all alternating sequences of states and events.

What are *states* and *events* here?

CACHE STATES

ACCESSES TO MEMORY
+
HIT/MISSES

# 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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

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 $\mathcal{I}$:

$$Col(P) = lfp_{\mathcal{I}}^{\subseteq}\ next$$

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

$$\underline{next(S)} = \{t.\sigma_n \underline{e_n \sigma_{n+1}} \mid t.\underline{\sigma_n} \in S \wedge (\sigma_n, e_n, \sigma_{n+1}) \in \mathcal{T}\}$$

# 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
  - $\mathcal{E}_{\mathcal{C}} = \{\mathit{hit}, \mathit{miss}, \perp\}$ are the "cache events".
- Transition relation $\mathcal{T}$:

$$\mathcal{T} = \{((m, c), (e_m, e_c), (m', c')) \mid m' = \mathit{upd}_{\mathcal{M}}(m) \wedge e_m = \mathit{eff}_{\mathcal{M}}(m) \\ \wedge\ c' = \mathit{upd}_{\mathcal{C}}(c, e_m) \wedge\ e_c = \mathit{eff}_{\mathcal{C}}(c, e_m)\},$$

# Trace Collecting Semantics: Instantiation for Caches

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- 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
  - $\mathcal{E}_{\mathcal{M}} = \mathcal{B} \cup \{\bot\}$ are the memory blocks the program may access, and
  - $\mathcal{E}_{\mathcal{C}} = \{\mathit{hit}, \mathit{miss}, \bot\}$ are the "cache events".
- Transition relation $\mathcal{T}$:

$$\mathcal{T} = \{((m, c), (e_m, e_c), (m', c')) \mid m' = \mathit{upd}_{\mathcal{M}}(m) \wedge e_m = \mathit{eff}_{\mathcal{M}}(m) \wedge c' = \mathit{upd}_{\mathcal{C}}(c, e_m) \wedge e_c = \mathit{eff}_{\mathcal{C}}(c, e_m)\},$$

What are $\mathit{upd}_{\mathcal{M}}$, $\mathit{eff}_{\mathcal{M}}$, $\mathit{upd}_{\mathcal{C}}$, and $\mathit{eff}_{\mathcal{C}}$?

# Notions of Persistence in Terms of Trace Collecting Semantics



There are at least three notions in the literature:

$$
\begin{aligned}
\mathit{persistent}(P, b) &:= \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \mathit{Col}(P) : \\
&\qquad \forall i, j : (e_i = e_j = (b, \mathit{miss})) \Rightarrow i = j. \\
\mathit{firstmiss}(P, b) &:= \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \mathit{Col}(P) : \\
&\qquad \forall i : (e_i = (b, \mathit{miss})) \Rightarrow \forall j < i : (e_j \neq (b, \mathit{hit}) \wedge e_j \neq (b, \mathit{miss})). \\
\mathit{noeviction}(P, b) &:= \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \mathit{Col}(P) : \\
&\qquad \forall i : (e_i = (b, \mathit{miss})) \Rightarrow \forall j > i : (b \in \sigma_n).
\end{aligned}
$$

# Notions of Persistence in Terms of Trace Collecting Semantics

SAARLAND UNIVERSITY
COMPUTER SCIENCE

There are at least three notions in the literature:

$$\begin{aligned}
\mathit{persistent}(P, b) &:= \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \mathit{Col}(P) : \\
&\qquad \forall i, j : (e_i = e_j = (b, \mathit{miss})) \Rightarrow i = j. \\
\mathit{firstmiss}(P, b) &:= \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \mathit{Col}(P) : \\
&\qquad \forall i : (e_i = (b, \mathit{miss})) \Rightarrow \forall j < i : (e_j \neq (b, \mathit{hit}) \wedge e_j \neq (b, \mathit{miss})). \\
\mathit{noeviction}(P, b) &:= \forall t = \sigma_0 e_0 \sigma_1 e_1 \ldots e_{n-1} \sigma_n \in \mathit{Col}(P) : \\
&\qquad \forall i : (e_i = (b, \mathit{miss})) \Rightarrow \forall j > i : (b \in \sigma_n).
\end{aligned}$$

Is one notion weaker/~~stronger~~ than the others?

# Abstractions of the Trace Collecting Semantics

SAARLAND UNIVERSITY
COMPUTER SCIENCE

The *trace collecting semantics* is not computable.

$\longrightarrow$ Need to apply abstractions.

Generic approach:

- Associate abstractions of sets of "cache traces" with program points.

$\rightarrow$ 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:

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

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

$$\widehat{StickyCol}(P_{ins}) = lfp^{\sqsubseteq}_{\widehat{Init}} \widehat{next}_{ins},$$

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

$$\widehat{next}_{ins}(\widehat{S}) = \lambda l' \in \mathcal{L}. \bigsqcup_{(l,l')\in E} \{\widehat{upd}_{tr}(\widehat{S}(l), b) \mid b = eff_{\mathcal{L}}(l, l')\}.$$

# Abstractions for Cache Traces

SAARLAND UNIVERSITY
COMPUTER SCIENCE

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)

SAARLAND UNIVERSITY
COMPUTER SCIENCE

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

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

Join: $cs_1 \sqcup cs_2 := cs_1 \cup cs_2$

Concretization: $\gamma(cs) :=$

Classification: $\widehat{persistent}(cs, b) :=$

# Set-wise conflict counting
# (Here: for fully-associative cache)

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

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

Classification: $\widehat{persistent}(cs, b) :=$

# Set-wise conflict counting (Here: for fully-associative cache)

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

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

Join: $cs_1 \sqcup cs_2 := cs_1 \cup cs_2$

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

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

# Set-wise conflict counting
# (Here: for fully-associative cache)



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

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

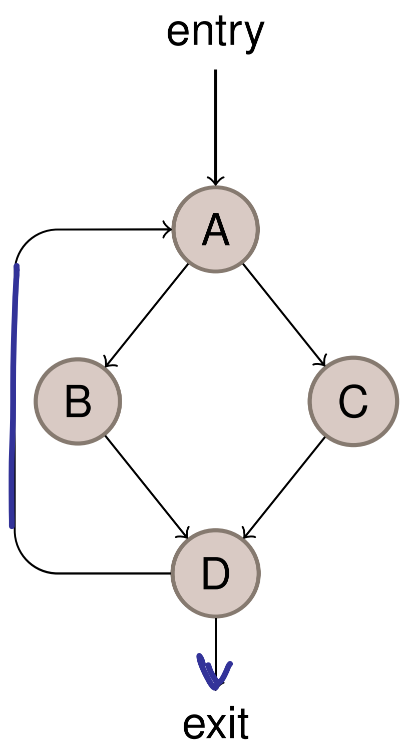
Join: $cs_1 \sqcup cs_2 := cs_1 \cup cs_2$

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

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

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: $\widehat{BCS} := \mathcal{B} \to \mathcal{P}(\mathcal{B})$

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: $\gamma(bcs) :=$

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

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

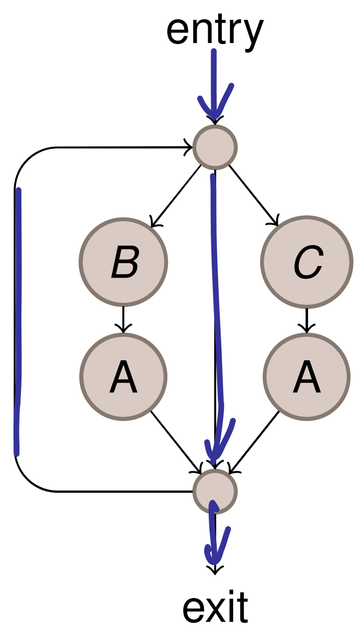
Join: $bcs_1 \sqcup bcs_2 := \lambda b : bcs_1(b) \cup bcs_2(b)$

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

Classification: $\widehat{persistent}(cs, b) := |bcs(b)| < k$

# Block-wise conflict counting: Example

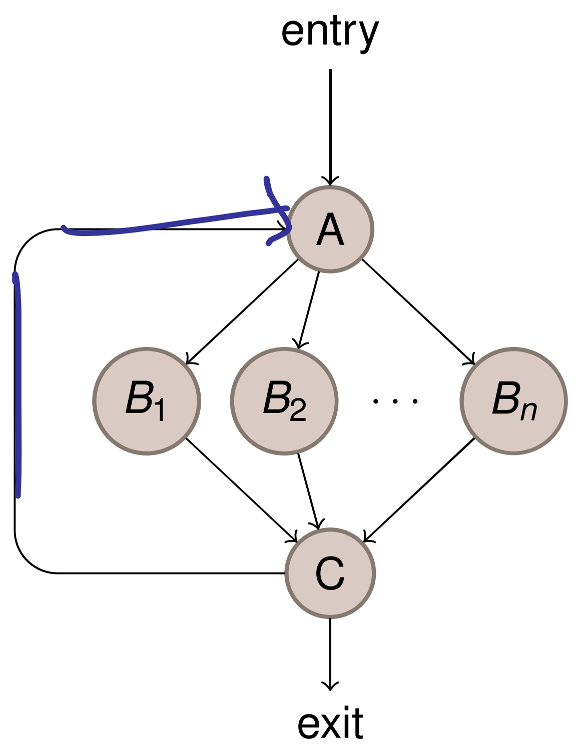
SAARLAND UNIVERSITY
COMPUTER SCIENCE




Which blocks are determined to be persistent for associativity ~~3~~ 2 under block- and set-wise conflict counting?

A ✓ ↯

# 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: $\widehat{CM} := \mathcal{B} \rightarrow \{0, \ldots, k\}$

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: $\gamma(cm) :=$

Classification: $\widehat{persistent}(cm, b) :=$

# Conditional Must Analysis (Here: for fully-associative cache)

SAARLAND UNIVERSITY
COMPUTER SCIENCE

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: $\gamma(cm) := \{c_1(b_1, h_i)c_2 \ldots c_n \mid \forall i < n :$
$c_{i+1} = upd(c_i, b_i) \wedge age(\underset{c_n}{\cancel{cm}}, b_i) \leq cm(b_i)\}$

Classification: $\widehat{persistent}(cm, b) :=$

# Conditional Must Analysis
# (Here: for fully-associative cache)

SAARLAND UNIVERSITY
COMPUTER SCIENCE

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

$c_m$

Classification: $\widehat{persistent}(cm, b) := cm(b) < k$

# 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: $\gamma(cm) := \{c_1(b_1, h_i)c_2 \ldots c_n \mid \forall i < n :$
$c_{i+1} = upd(c_i, b_i) \wedge age(\underset{c_n}{\cancel{cm}}, b_i) \leq cm(b_i)\}$

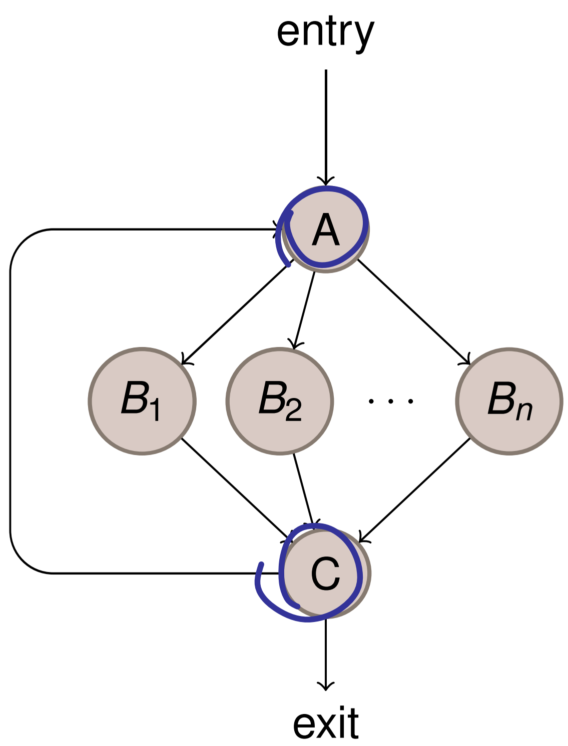
Classification: $\widehat{persistent}(cm, b) := cm(b) < k$

What should be the initial value for this domain?

$\mapsto 0$

# Conditional Must-Analysis: Example Revisited

SAARLAND UNIVERSITY
COMPUTER SCIENCE




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? No.

A B B B B A

# Refinments of the Persistence Notion

The persistence notion can be refined in several directions:

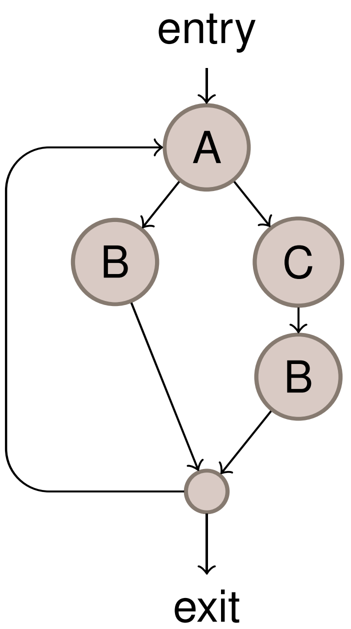
- A block may be persistent during the execution of an inner loop, but not in the execution of the program as a whole.
  $\longrightarrow$ Classify blocks to be persistent in particular *program scopes*.

# Refinments 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.
  $\longrightarrow$ 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:




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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

## 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?