

The Semantic Foundations and a Landscape of Cache-Persistence Analyses*


Jan Reineke

Saarland University

Saarland Informatics Campus

Saarbrücken, Germany

reineke@cs.uni-saarland.de

 <http://orcid.org/0000-0002-3459-2214>

Abstract

We clarify the notion of cache persistence and contribute to the understanding of persistence analysis for caches with least-recently-used replacement.

To this end, we provide the first formal definition of persistence as a property of a trace semantics. Based on this trace semantics we introduce a semantics-based, i.e., abstract-interpretation-based persistence analysis framework.

We identify four basic persistence analyses and prove their correctness as instances of this analysis

framework.

Combining these basic persistence analyses via two generic cooperation mechanisms yields a lattice of ten persistence analyses.

Notably, this lattice contains all persistence analyses previously described in the literature. As a consequence, we obtain uniform correctness proofs for all prior analyses and a precise understanding of how and why these analyses work, as well as how they relate to each other in terms of precision.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture, Theory of computation → Caching and paging algorithms, Hardware → Safety critical systems

Keywords and Phrases caches, persistence analysis, WCET analysis

Digital Object Identifier 10.4230/LITES-v005-i001-a003

Received 2017-10-23 **Accepted** 2018-05-15 **Published** 2018-08-02

1 Introduction

Due to the large processor-memory gap, essentially all modern processors employ some form of memory hierarchy, consisting of smaller but faster memories, such as caches, on the one hand, and larger but slower memories, such as DRAM-based main memory, on the other hand. Memory hierarchies of general-purpose processors usually contain one or multiple levels of caches. Caches are small but fast hardware-managed memories that store a subset of the contents of main memory. Memory accesses that “hit” the cache may be served at a much lower latency than those accesses that “miss” the cache and as a consequence have to be served from slow main memory. The execution time of a program thus heavily depends on how effective the processor’s caches are.

For safety-critical systems, it is imperative to demonstrate before deployment that the system will always behave as intended. Many safety-critical systems are real-time systems, i.e., in order to function correctly, they have to perform their actions within limited amounts of wall-clock time. A major task in verifying a system’s real-time behavior is to analyze each software component’s worst-case execution time (WCET). Due to the large influence of caches on execution times, WCET analyses have to soundly and precisely characterize a software component’s cache behavior. To this end, various static cache analyses have been developed. Simply assuming each memory access to yield a cache miss would result in extremely pessimistic execution-time bounds.

* This work was supported by the Deutsche Forschungsgemeinschaft as part of the project PEP.



© Jan Reineke;

licensed under Creative Commons Attribution 3.0 Germany (CC BY 3.0 DE)

Leibniz Transactions on Embedded Systems, Vol. 5, Issue 1, Article No. 3, pp. 03:1–03:52



Leibniz Transactions on Embedded Systems

LITES Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ **Listing 1** Example program motivating persistence analysis.

```
for (int i=0; i < N; i++) {
    if (read_sensor())
        a++;
    else
        b++;
}
```

Static cache analysis comes in two flavours [19]: 1. *Classifying Cache Analysis* aims to classify *individual* memory accesses as cache hits or misses. 2. *Quantitative Cache Analysis* aims to determine bounds on the number of misses resulting from a *set* of memory accesses in a program.

Classifying cache analysis for caches with least-recently-used (LRU) replacement has been well-understood since the introduction of may and must analysis by Ferdinand et al. [11, 12] in the late 1990s. May and must analysis are formalized and proven correct in the framework of abstract interpretation [3]. Both may and must analysis answer questions about the *set of reachable cache states*. For example, must analysis determines whether a memory block is guaranteed to be cached in all cache states that may be reached at a given program point. If that is the case, an access to such a block must result in a cache hit.

One instance of quantitative cache analysis is *persistence analysis*. Persistence analysis collectively considers all memory accesses in a program, or a fragment of a program such as a loop, that access the *same* memory block. Various slightly different interpretations of cache persistence exist in the literature [1, 12], which we discuss later in this article. Intuitively, a memory block is persistent if, during any possible program execution, all memory accesses referring to this block may cumulatively result in at most one cache miss.

Consider the program in Listing 1 for a motivating example. Assume that variables **a** and **b** are kept in two distinct memory blocks. Further, assume that in each loop iteration it is equally possible for the program to take the then- and the else-branch of the conditional, as the outcome of `read_sensor()` depends on external inputs. Then it is impossible to classify the memory accesses to **a** or **b** in any loop iteration as guaranteed cache hits and a WCET analysis would have to pessimistically account for misses upon all memory accesses. However, provided the cache is large enough to hold **a** and **b** simultaneously, among all memory accesses to **a** (and similarly to **b**) only the very first may result in a cache miss¹. Both **a** and **b** are persistent, and WCET analysis can safely account for at most two misses.

Various persistence analyses have been proposed in the literature starting from Mueller’s [20, 1, 31, 21] and Ferdinand’s [11, 12] work in the 1990s up until today [2, 6, 17, 23, 22, 8, 7, 32]. In our opinion, however, persistence analysis is so far not as well-understood as classifying cache analysis. In particular, even though persistence analysis clearly determines semantic properties of programs, it has never been formalized and proven correct as a semantics-based program analysis, i.e., as an abstract interpretation of an appropriate semantics in which persistence is expressible. Instead, persistence analyses have so far been described and argued correct in rather ad hoc manners. Possibly as a consequence of this lack of foundations, a flaw in one of the early persistence analyses [12] was long overlooked.

In this article we seek to fill this gap by providing a solid semantic underpinning for persistence analysis. We observe that persistence is a property of *traces* rather than *states*. Thus, semantics that capture sets of reachable states – such as those used as a basis for may and must analysis,

¹ Assuming that neither **a** nor **b** are evicted by `read_sensor()`.

and in fact most other static program analyses – are not appropriate to understand and prove correct persistence analyses. In Section 2, we define a *trace collecting semantics*, which captures all possible cache traces of a program, i.e., alternating sequences of cache states and memory accesses. On this basis, we are then able to provide the first formal definition of the various cache persistence notions found in the literature.

After discussing standard abstractions and simplifications in Section 3, we introduce a generic abstract-interpretation-based persistence analysis framework in Section 4. This framework defines the components of a persistence analysis and provides conditions on these components that are sufficient to guarantee the correctness of the persistence classifications of the analysis. As is usual in abstract interpretation, the framework uses concretization functions to capture the relation between concrete and abstract semantics. The key difference to prior work on abstract-interpretation-based cache analysis is that concretization functions map to sets of cache traces rather than sets of cache states, as persistence is a property of traces rather than states. To analyze the relative precision of two different persistence analyses, we also provide conditions on the components of two arbitrary analyses A and B that are sufficient to show that A is more precise than B , i.e., if B classifies a memory block as persistent then so does A .

A generic analysis framework is only useful if it has interesting instantiations. In Section 5, we identify four basic persistence analyses. Using the framework introduced in Section 4 we prove their correctness and determine their relative precision. Then we introduce two generic cooperation mechanisms that enable the exchange of analysis information between different persistence analyses in order to obtain more precise combined analyses. Combining the four basic persistence analyses using these two cooperation mechanisms yields a lattice of ten persistence analyses of varying precision. These ten analyses include, to the best of our knowledge, *all* persistence analyses previously described in the literature. Thus we obtain uniform correctness proofs for all these analyses and a precise understanding of how and why these analyses work, as well as how they relate to each other in terms of precision. In Section 6, we discuss how the persistence analyses from the literature map to the lattice of persistence analyses developed in Section 5.

Due to uncertainty about the memory accesses induced by loads and stores in a program, persistence analysis is more challenging for data caches than for instruction caches. We briefly describe a generic approach to data-cache persistence analysis in Section 7. Finally, we conclude the article by summarizing our results and discussing future work in Section 8.

This article may be read in different ways depending on a reader’s intent:

- Readers primarily interested in understanding the intuition of the various persistence analyses may focus their attention on Section 5. To enable readers to quickly obtain a basic understanding of the state of the art, many of the correctness proofs have been moved to the appendix.
- Readers who would like to understand the semantic foundations of persistence analysis in detail, will have to study Sections 2 to 4 more carefully. Further, they may selectively read the detailed correctness proofs in the appendix. These proofs are linked from the respective theorems and lemmas of Section 5.
- Readers interested in a historical perspective may focus on Sections 5.3 and 6.

2 A Formal Definition of Cache Persistence

In this section, we provide a formal definition of cache persistence. As persistence is a semantic property of a program’s execution traces, we first introduce a generic trace collecting semantics in Section 2.1. As persistence involves cache behavior, we require a semantics taking into account caches. We show how to instantiate the generic trace collecting semantics accordingly in Section 2.2. Finally, in Section 2.3 we formally capture the various notions of persistence found in the literature as properties of the semantics introduced in the two preceding sections.

2.1 Programs, Computations, Trace Collecting Semantics

A program $P = \langle \Sigma, \mathcal{I}, \mathcal{E}, \mathcal{T} \rangle$ consists of the following components:

- Σ - 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*, which captures how a computation of the program may step through its state space.

An *execution trace* t of P is an alternating sequence of states and events $t = \sigma_0 e_0 \sigma_1 e_1 \dots \sigma_n$ such that $\sigma_0 \in \mathcal{I}$ and for all $i \in \{0, \dots, n-1\}$, $\langle \sigma_i, e_i, \sigma_{i+1} \rangle \in \mathcal{T}$. The set of all execution traces of P is its *trace collecting semantics* $Col(P) \subseteq Traces$, where $Traces$ denotes the set of all alternating sequences of states and events. When considering terminating programs, the trace collecting semantics can be formally defined as the least fixpoint² of the *next* operator containing \mathcal{I} :

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

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

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

In the definition of *next* above, $t.\sigma_n$ denotes a trace that ends in state σ_n following its prefix t . Similarly, $t.\sigma_n e_n \sigma_{n+1}$ is a trace obtained by extending $t.\sigma_n$ by $e_n \sigma_{n+1}$.

In other words, $Col(P)$ is the least solution, i.e., the smallest set of traces X that satisfies the equation $X = \mathcal{I} \cup next(X)$.

Cousot and Cousot [5] give a detailed proof of why the set of all finite execution traces of a program is indeed captured by the least fixpoint of a *next* operator as in our definition above.

2.2 Taking Caches Into Account

For reasoning about caches, we need to consider a semantics in which the state of the cache is part of the program state. To this end, the state will consist of two components: (1) the logical memory state in \mathcal{M} (representing the values of memory locations and CPU registers) and (2) the cache state in \mathcal{C} . So $\Sigma = \mathcal{M} \times \mathcal{C}$. The set of initial states is the product of the set of initial memory states and the set of initial cache states, i.e., $\mathcal{I} = \mathcal{I}_{\mathcal{M}} \times \mathcal{I}_{\mathcal{C}}$.

We define the transition relation on this domain based on four functions that model the evolution of the logical memory and the cache as well as their interaction:

1. The *memory update* is a function $update_{\mathcal{M}}: \mathcal{M} \rightarrow \mathcal{M}$ that captures the logical memory state the system transitions into from a given logical memory state.
2. The *memory effect* is a function $eff_{\mathcal{M}}: \mathcal{M} \rightarrow \mathcal{E}_{\mathcal{M}}$ that determines the memory block, if any, that is accessed when transitioning from a given logical memory state. We denote the set of memory blocks by \mathcal{B} . Thus, the set of memory events is defined as $\mathcal{E}_{\mathcal{M}} = \mathcal{B} \cup \{\perp\}$, where \perp denotes that no memory block is accessed.
3. The *cache update* is a function $update_{\mathcal{C}}: \mathcal{C} \times \mathcal{E}_{\mathcal{M}} \rightarrow \mathcal{C}$ that determines the successor cache state given a memory effect.

² Here, and later, we denote by $lfp_{\mathcal{I}}^{\subseteq} next$, the least fixpoint of the function *next* that is greater than or equal to \mathcal{I} . This is the same as the least fixpoint of the function $next_{\mathcal{I}}(X) := \mathcal{I} \cup next(X)$.

4. The *cache effect* is a function $eff_C: \mathcal{C} \times \mathcal{E}_M \rightarrow \mathcal{E}_C$ that determines whether or not the memory access results in a cache hit or a cache miss. Thus, the set of cache events is defined as $\mathcal{E}_C = \{hit, miss, \perp\}$, where \perp is used when the memory effect is \perp , and so no memory block is actually accessed.

Events are pairs of memory events and cache events, i.e., $\mathcal{E} = \mathcal{E}_M \times \mathcal{E}_C$.

The definition of persistence as a property of traces in Section 2.3, as well as the persistence analysis framework developed in Sections 3 and 4 applies to arbitrary replacement policies. Particular replacement policies can be captured by appropriately defining $update_C$ and eff_C . Below, we provide definitions of these two functions for the LRU strategy, denoted as $update_C^{LRU}$ and eff_C^{LRU} . This is because all the persistence analysis instances that we will introduce in Section 5 apply to caches with LRU replacement. In the following, whenever we make statements that hold for arbitrary caches, we use $update_C$ and eff_C . Whenever our statements refer to LRU, we use $update_C^{LRU}$ and eff_C^{LRU} .

Upon a cache miss, LRU replaces the least-recently-used memory block. To this end, it tracks the ages of memory blocks within each cache set, where the youngest block has age 0 and the oldest cached block has age $k - 1$, where k is the associativity of the cache. Thus, the state of the cache can be modeled as a function that assigns an age to each memory block, where non-cached blocks are assigned age k . For simplicity of exposition, we consider a fully-associative cache³, in other words, all blocks map to the same cache set.

$$\mathcal{C} := \{c \in \mathcal{B} \rightarrow A \mid \forall a, b \in \mathcal{B} : a \neq b \Rightarrow (c(a) \neq c(b) \vee c(a) = c(b) = k)\},$$

where $A := \{0, \dots, k - 1, k\}$ is the set of ages. The constraint encodes that no two cached blocks can have the same age. For readability we omit the additional constraint that blocks of non-zero age are preceded by other blocks, i.e. that cache sets do not contain “holes”.

Below, we define the cache update and the cache effect for the cases where a memory access occurs, i.e., for the subset \mathcal{B} of their domain $\mathcal{E}_M = \mathcal{B} \cup \{\perp\}$. Both cache update and cache effect are naturally extended to the case where no memory access occurs.

The cache update for LRU is given by

$$update_C^{LRU}(c, b) := \lambda b' \in \mathcal{B}. \begin{cases} 0 & : b' = b, \\ c(b') + 1 & : c(b') < c(b), \\ c(b') & : c(b') \geq c(b). \end{cases}$$

The accessed block attains age 0 (case 1), blocks younger than the accessed block age by 1 (case 2), and the ages of other blocks are not affected (case 3).

The cache effect captures that a hit occurs whenever the age of the accessed block is less than the cache’s associativity:

$$eff_C^{LRU}(c, b) := \begin{cases} hit & : c(b) < k, \\ miss & : \text{else.} \end{cases}$$

Both $update_C^{LRU}$ and eff_C^{LRU} are naturally extended to the case where no memory access occurs. Then, the cache state remains unchanged and the cache effect is \perp .

³ Set-associative caches as they are typically found in actual caches can be considered as arrays of fully-associative caches. Thus, analyses that apply to fully-associative caches can be lifted to set-associative caches in a rather straightforward manner.

With this, we can now connect the components and obtain the global transition relation $\mathcal{T} \subseteq \Sigma \times \mathcal{E} \times \Sigma$ by

$$\mathcal{T} = \{ \langle \langle m, c \rangle, \langle e_m, e_c \rangle, \langle m', c' \rangle \rangle \mid m' = \text{update}_{\mathcal{M}}(m) \wedge e_m = \text{eff}_{\mathcal{M}}(m) \\ \wedge c' = \text{update}_{\mathcal{C}}(c, e_m) \wedge e_c = \text{eff}_{\mathcal{C}}(c, e_m) \},$$

which formally captures the asymmetric relationship between caches, logical memories, and events:

- The *memory update* determines the next memory states: $m' = \text{update}_{\mathcal{M}}(m)$, and the *memory effect* determines the memory block, if any, that is accessed on the transition: $e_m = \text{eff}_{\mathcal{M}}(m)$.
- Based on the memory effect, the *cache update* determines the next cache state: $c' = \text{update}_{\mathcal{C}}(c, e_m)$, and the *cache effect* determines whether or not the current memory access results in a hit or a miss: $e_c = \text{eff}_{\mathcal{C}}(c, e_m)$.

2.3 Persistence as a Property of Traces

Given a trace collecting semantics that takes caches into account as defined in the previous two sections, we are now ready to formally capture multiple notions of persistence found in the literature. Persistence is a property of traces, and thus, the following predicates determine for a given trace τ and a memory block b , whether b is persistent in τ according to a particular notion of persistence.

The most liberal notion of persistence is that a persistent block may cause at most one miss:

$$\text{AtMostOneMiss}(\sigma_0 e_0 \sigma_1 e_1 \dots e_{n-1} \sigma_n, b) := |\{i \mid e_i = \langle b, \text{miss} \rangle\}| \leq 1 \quad (1)$$

A stronger notion of persistence is that only the very first access to a persistent block may result in a miss [1]:

$$\text{FirstAccessIsAMiss}(\sigma_0 e_0 \sigma_1 e_1 \dots e_{n-1} \sigma_n, b) := \\ \forall i : (e_i = \langle b, \text{miss} \rangle) \Rightarrow \forall j < i : (e_j \neq \langle b, \text{hit} \rangle \wedge e_j \neq \langle b, \text{miss} \rangle). \quad (2)$$

This condition is stronger, i.e., $\text{FirstAccessIsAMiss}(\tau, b)$ implies $\text{AtMostOneMiss}(\tau, b)$ but not vice versa, as $\text{AtMostOneMiss}(\tau, b)$ is equivalent to $\forall i : (e_i = \langle b, \text{miss} \rangle) \Rightarrow \forall j < i : (e_j \neq \langle b, \text{miss} \rangle)$.

Another stronger notion of persistence is that, after a block has been brought into the cache via a miss, it is not evicted from the cache anymore [12]:

$$\text{NoEviction}(\sigma_0 e_0 \sigma_1 e_1 \dots e_{n-1} \sigma_n, b) := \forall i : (e_i = \langle b, \text{miss} \rangle) \Rightarrow \forall j > i : b \in \sigma_j, \quad (3)$$

where $b \in \sigma_l$ means that b is cached in state σ_l , i.e., it is an abbreviation for $\sigma_l = \langle m_l, c_l \rangle \wedge \text{eff}_{\mathcal{C}}(c_l, b) = \text{hit}$. Clearly, persistence according to (3) also implies persistence according to (1).

The above definitions refer to individual traces. We say that a memory block b is persistent in program P , if it is persistent in all traces of P 's trace collecting semantics:

► **Definition 1** (Persistence in a Program). *Memory block b is persistent in program P , if*

$$\forall \tau \in \text{Col}(P) : \text{AtMostOneMiss}(\tau, b).$$

We choose to use the most liberal notion of persistence in Definition 1, because it corresponds to the property that is being exploited in the later phases of WCET analysis. As we will see, all persistence analyses introduced in Section 5 are in fact based on the stronger *NoEviction* notion expressed by (3). It is conceivable though that future persistence analyses will take advantage of the more liberal *AtMostOneMiss* notion to classify more memory blocks as persistent.

3 Preliminaries: Standard Abstractions and Simplifications

The trace collecting semantics as defined above is not practically computable. In this section, we discuss two very common abstractions that lead to an abstract semantics that is closer to being computable. Based on the resulting sticky-collecting semantics, we then develop further abstractions in Sections 4 and 5 that allow to prove the persistence of memory blocks in practice.

3.1 Control Flow Graph Abstraction

In contrast to data accesses, instruction accesses depend solely on the flow of control through the program and are thus much easier to predict. As the focus of this article is on the analysis of the cache behavior rather than the analysis of the memory accesses generated by a program, we initially limit ourselves to the analysis of instruction caches. Later, in Section 7 we discuss how to lift this restriction. Thus we abstract the state of the memory, in \mathcal{M} , to the program location, in \mathcal{L} , that the program is currently at.

A common abstraction of a program P is its control flow graph $G_P = \langle \mathcal{L}, E, i \rangle$, where

- the nodes in \mathcal{L} represent program locations,
- the edges in $E \subseteq \mathcal{L} \times \mathcal{L}$ represent possible control flow, and
- $i \in \mathcal{L}$ represents the start node, which has no incoming edges.

The set of edges E can be seen as an abstraction of the memory update $update_{\mathcal{M}}$. While $update_{\mathcal{M}}$ is a function, E is a relation, because the successor location may depend on values of registers that have been abstracted away. Similarly, let $eff_{\mathcal{L}} : E \rightarrow \mathcal{B}$ capture the memory block holding the instruction that needs to be fetched when moving from one program location to another. This corresponds to the memory effect $eff_{\mathcal{M}}$. As we limit ourselves to instruction accesses, which are precisely determined by control flow, there is no loss in precision moving from $eff_{\mathcal{M}}$ to $eff_{\mathcal{L}}$. Also, each edge corresponds to exactly one memory access, and so we do not need to consider the trivial case that no memory access is performed upon a transition. With this abstraction, the set of states is $\Sigma_{ins} = \mathcal{L} \times \mathcal{C}$, and the set of initial states is $\mathcal{I}_{ins} = \{i\} \times \mathcal{I}_{\mathcal{C}}$.

Based on these notions, we obtain the following global transition relation \mathcal{T}_{ins} :

$$\mathcal{T}_{ins} = \{ \langle \langle l, c \rangle, \langle b, h \rangle, \langle l', c' \rangle \rangle \mid \langle l, l' \rangle \in E \wedge b = eff_{\mathcal{L}}(l, l') \wedge c' = update_{\mathcal{C}}(c, b) \wedge h = eff_{\mathcal{C}}(c, b) \},$$

which yields the abstraction P_{ins} of P : $P_{ins} = \langle \Sigma_{ins}, \mathcal{I}_{ins}, \mathcal{E}, \mathcal{T}_{ins} \rangle$.

One could formally relate $Col(P_{ins})$ and $Col(P)$ by concretization and abstraction functions and derive correctness conditions on E and $eff_{\mathcal{L}}$, but we omit this here⁴ and assume that the control flow graph is the starting point of the analyses presented below, as is common in the literature [1, 20, 31, 11, 12, 21, 2, 6, 17, 8, 7, 32].

We note, however, that more precise results can be obtained if persistence analysis is carried out on more precise abstractions of the program's memory access behavior, which can be obtained by e.g. trace partitioning [28].

3.2 Abstraction from Locations in Traces

As we can see from Definition 1, to determine whether a block is persistent it suffices to inspect the cache and memory effects of the trace collecting semantics. We thus further abstract the trace collecting semantics to a semantics that only maintains traces of cache and memory effects, forgetting about the intermediate locations. We denote the set of such traces by *CacheTraces*.

⁴ We consider the problem of soundly abstracting a program's memory access behavior to be distinct from the problem of cache persistence analysis based on such an abstraction, which is the topic of this article. See Section 4.2 in [10] for a concretization function relating $Col(P_{ins})$ and $Col(P)$.

This *sticky cache trace collecting semantics*⁵, $StickyCol(P_{ins}) : \mathcal{L} \rightarrow 2^{CacheTraces}$, captures the set of traces of cache states and cache and memory effects that may reach a given program location. It is defined as the least fixpoint of $next_{ins}$, defined below, including $Init$:

$$StickyCol(P_{ins}) := \text{lfp}_{Init}^{\leq} next_{ins},$$

where $Init = \lambda l. (l = i ? \mathcal{I}_C : \emptyset)$, the partial order \leq denotes the point-wise comparison, i.e., $S \leq T := \forall l \in \mathcal{L} : S(l) \subseteq T(l)$, which induces the join $S \vee T := \lambda l \in \mathcal{L}. S(l) \cup T(l)$, and $next_{ins}$ is defined as follows:

$$\begin{aligned} next_{ins}(S) &= \lambda l' \in \mathcal{L}. \bigcup_{(l, l') \in E} \{t.c.e.c' \mid t.c \in S(l) \wedge \langle \langle l, c \rangle, e, \langle l', c' \rangle \rangle \in \mathcal{T}_{ins}\} \\ &\stackrel{\text{Def. of } \mathcal{T}_{ins}}{=} \lambda l' \in \mathcal{L}. \bigcup_{(l, l') \in E} \{t.c \langle b, h \rangle c' \mid t.c \in S(l) \wedge b = \text{eff}_{\mathcal{L}}(l, l') \\ &\quad \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \end{aligned}$$

The function $next_{ins}$ defined above captures how the set of cache traces reaching location l' is recursively determined by the set of cache traces reaching predecessor locations of l' and the memory accesses on the edges from the predecessors to l' .

We can relate the sticky cache trace collecting semantics to the corresponding trace collecting semantics by the concretization function γ_{ins} :

$$\gamma_{ins}(S) = \{\langle l_0, c_0 \rangle e_0 \langle l_1, c_1 \rangle \dots e_{n-1} \langle l_n, c_n \rangle \in Traces \mid \forall i \leq n : c_0 e_0 \dots c_i \in S(l_i)\} \quad (4)$$

It can be shown that $Col(P_{ins}) \subseteq \gamma_{ins}(StickyCol(P_{ins}))$.

4 A Generic Persistence Analysis Framework

The sticky cache trace collecting semantics defined above associates sets of cache traces with each program point. These traces may be arbitrarily long and there may be infinitely many associated with a single program point. Thus, an effective analysis needs to further abstract from this semantics, by representing potentially infinite sets of cache traces in a finite fashion. Before discussing particular abstractions of cache traces in Section 5, we show in this section how to lift any such abstraction to a sound persistence analysis in Sections 4.1 and 4.2, and in Section 4.3 we show how to characterize the relative precision of different persistence analyses.

4.1 Sound Cache Trace Abstractions

Before formally defining cache trace abstractions, let us informally state their components. First, we need a set of abstract traces, which will be used by the analysis in place of sets of concrete cache traces. To enable a proof of correctness, these abstract traces need to be related to sets of concrete traces by a concretization function, which specifies the set of concrete traces represented by each abstract trace. Usually no information is available about the initial state of the cache. Thus, the abstract traces need to contain an initial abstract trace that represents all possible initial cache states. To combine analysis information at control flow joins, a join operator on abstract traces is required. The core of a cache trace abstraction is the abstract update function, which captures the effect of a memory access on abstract cache traces. Finally, a persistence classification function is required to determine whether a memory block is persistent in all concrete cache traces represented by an abstract trace. These components yield the following definition of a cache trace abstraction.

⁵ We call this semantics “sticky” because it sticks sets of traces to each program location.

► **Definition 2** (Cache Trace Abstraction). A cache trace abstraction is a tuple

$$A = \langle C_A^\#, \gamma_A, \widehat{\mathcal{I}}_A, \sqsubseteq_A, \sqcup_A, \text{update}_A^\#, \text{classify}_A^\# \rangle,$$

consisting of the following components:

1. $C_A^\#$, a set of abstract traces,
2. $\gamma_A : C_A^\# \rightarrow 2^{\text{CacheTraces}}$, a concretization function, which specifies the set of concrete cache traces represented by each abstract trace,
3. $\widehat{\mathcal{I}}_A \in C_A^\#$, an abstract initial trace that represents all possible initial cache states,
4. \sqsubseteq_A , a partial order on $C_A^\#$, such that $\langle C_A^\#, \sqsubseteq_A \rangle$ is a complete lattice [9],
5. \sqcup_A , a join operator on abstract traces⁶,
6. $\text{update}_A^\# : C_A^\# \times \mathcal{B} \rightarrow C_A^\#$, an abstract update function,
7. $\text{classify}_A^\# : C_A^\# \times \mathcal{B} \rightarrow \mathbb{B}$, a persistence classification function.

We will introduce requirements on the components of a cache trace abstraction in Theorems 3 and 4 that together imply correct analysis results.

Given a cache trace abstraction A we can define the abstract next operator as follows:

$$\text{next}_{ins,A}^\#(\widehat{S}) = \lambda l' \in \mathcal{L}. \bigsqcup_{\langle l, l' \rangle \in E} \{ \text{update}_A^\#(\widehat{S}(l), b) \mid b = \text{eff}_{\mathcal{L}}(l, l') \}$$

Intuitively, the abstract next operator captures how the analysis state at location l' depends on the analysis state at predecessor locations and the abstract update function of the cache trace abstraction.

Based on the abstract initial trace $\widehat{\mathcal{I}}_A$, we can define the initial analysis state $\widehat{\text{Init}}_A := \lambda l \in \mathcal{L}. (l = i ? \widehat{\mathcal{I}}_A : \perp_A)$ analogously to the definition of Init earlier. The abstract sticky trace collecting semantics $\widehat{\text{StickyCol}}_A$ is then defined as the least fixpoint of $\text{next}_{ins,A}^\#$ greater than $\widehat{\text{Init}}_A$:

$$\widehat{\text{StickyCol}}_A(P_{ins}) = \text{lfp}_{\widehat{\text{Init}}_A}^{\sqsubseteq_A} \text{next}_{ins,A}^\#, \quad (5)$$

where \sqsubseteq_A is lifted to functions as follows: $\widehat{S} \sqsubseteq_A \widehat{T} := \forall l \in \mathcal{L} : \widehat{S}(l) \sqsubseteq_A \widehat{T}(l)$.

In order for the abstract sticky trace collecting semantics to be well-defined, we require the abstract update function to be monotone in the first parameter. This guarantees that the abstract $\text{next}_{ins,A}^\#$ operator is monotone. Then, the Knaster-Tarski fixpoint theorem [9], which is reproduced in Theorem 30 in the appendix, guarantees the existence of a unique least fixpoint. Note that requiring the abstract update function to be monotone is not a restriction: the best abstract update function [4] for a given abstraction is always monotone.

If the partial order \sqsubseteq_A on abstract traces satisfies the ascending chain condition [18], i.e., if there are no infinite ascending chains of abstract traces, then $\widehat{\text{StickyCol}}_A(P_{ins})$ can effectively be computed by fixpoint iteration [3]. In Section 4.2 we recapitulate a variant of the worklist algorithm [24, 29] to more efficiently compute $\widehat{\text{StickyCol}}_A(P_{ins})$.

For the analysis results to be correct, the abstract semantics should soundly approximate its concrete counterpart. This is the case if the cache trace abstraction satisfies these three conditions, which are formalized in the following theorem: 1. The abstract initial trace needs to represent all possible concrete initial cache states. 2. The concretization function needs to be monotone in \sqsubseteq_A . 3. The abstract update function needs to overapproximate the concrete update of cache states.

⁶ Note that in a complete lattice $\langle L, \sqsubseteq \rangle$ the partial order \sqsubseteq uniquely defines the join operator \sqcup . Vice versa, a given join operator uniquely defines a corresponding partial order. Nevertheless, we explicitly provide both partial order and join operator here and in the following.

► **Theorem 3** (Soundness of Persistence Analysis*). *If the cache trace abstraction A satisfies the following conditions:*

$$\mathcal{I}_C \subseteq \gamma_A(\widehat{\mathcal{I}}_A), \quad (6)$$

$$\forall \widehat{S}, \widehat{T} \in C_A^\# : \widehat{S} \sqsubseteq_A \widehat{T} \Rightarrow \gamma_A(\widehat{S}) \subseteq \gamma_A(\widehat{T}), \quad (7)$$

$$\forall \widehat{S} \in C_A^\#, b \in \mathcal{B} : \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\widehat{S}) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_A(\text{update}_A^\#(\widehat{S}, b)). \quad (8)$$

Then, its abstract semantics soundly approximates its concrete counterpart:

$$\text{StickyCol}(P_{ins}) \leq \gamma_A(\widehat{\text{StickyCol}}_A(P_{ins})), \quad (9)$$

where γ_A is lifted to functions as follows: $\gamma_A(\widehat{S}) = \lambda l \in \mathcal{L}. \gamma_A(\widehat{S}(l))$.

The proof to this theorem, and all other proofs that are not provided in the main part of the article, can be found in the appendix. Whenever a lemma or theorem is not immediately followed by its proof, the theorem's name is marked with a \star and serves as a link to the corresponding proof in the appendix. Similarly, theorems reproduced without proof in the appendix link back to their proofs in the main part.

As a consequence, the abstract sticky trace collecting semantics also soundly approximates the trace collecting semantics:

$$\text{Col}(P_{ins}) \subseteq \gamma_{ins}(\text{StickyCol}(P_{ins})) \subseteq \gamma_{ins}(\gamma_A(\widehat{\text{StickyCol}}_A(P_{ins}))).$$

The following theorem gives a condition on the persistence classification function that implies correct persistence classifications of memory blocks:

► **Theorem 4** (Soundness of Persistence Classification*). *If the cache trace abstraction A satisfies conditions (6), (7), (8) from Theorem 3, and $\text{classify}_A^\#$ satisfies*

$$\forall \widehat{S} \in C_A^\#, b \in \mathcal{B} : \text{classify}_A^\#(\widehat{S}, b) \Rightarrow \\ \forall c_0\langle b_0, h_0 \rangle c_1\langle b_1, h_1 \rangle \dots c_n \in \gamma_A(\widehat{S}) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b), \quad (10)$$

then $\text{classify}_A^\#(P_{ins}, b) := \forall l \in \mathcal{L} : \text{classify}_A^\#(\widehat{\text{StickyCol}}_A(P_{ins})(l), b)$ implies the persistence of memory block b in program P_{ins} .

The condition on $\text{classify}_A^\#$ in the theorem above is based on the *NoEviction* persistence notion from (3). It could be replaced by weaker conditions corresponding to the *FirstMiss* or the *AtMostOneMiss* persistence notions from (1) and (2). However, all persistence analyses we are aware of can be shown correct using the *NoEviction* notion.

4.2 Computing the Abstract Sticky Trace Collecting Semantics

Algorithm 1 shows how to compute the abstract sticky trace collecting semantics of a program for a given cache trace abstraction by Kleene iteration. The algorithm computes an increasing sequence of analysis states $\widehat{S}_0 \sqsubseteq_A \widehat{S}_1 \sqsubseteq_A \dots$ starting from the initial analysis state $\widehat{S}_0 = \widehat{\text{Init}}_A$, until a fixpoint is reached. This process is guaranteed to terminate if the complete lattice of abstract traces satisfies the ascending chain condition [18].

Algorithm 2 shows a *worklist algorithm* [24, 29]. The goal of worklist algorithms is to be more efficient than the Kleene iteration by avoiding redundant recomputations of parts of the abstract next operator $\text{next}_{ins, A}^\#$. Specifically, $\text{next}_{ins, A}^\#$ involves the application of the abstract update function to each edge in the control flow graph. However, $\text{update}_A^\#(\widehat{S}(l), b)$ in (5) will only deliver a different value than in the previous iteration if $\widehat{S}(l)$ has changed in the meantime.

Algorithm 1: Kleene Iteration

Input : Control Flow Graph $G_P = \langle \mathcal{L}, E, i \rangle$ and Cache Trace Abstraction A
Output : Abstract Sticky Trace Collecting Semantics $\widehat{StickyCol}_A(P_{ins})$

- 1 $\widehat{S}_0 := \widehat{Init}_A$
- 2 $i := 0$
- 3 **repeat**
- 4 $\widehat{S}_{i+1} := \widehat{Init}_A \sqcup_A next_{ins,A}^\#(\widehat{S}_i)$
- 5 $i := i + 1$
- 6 **until** $\widehat{S}_i = \widehat{S}_{i-1}$
- 7 **return** \widehat{S}_i

Algorithm 2: Worklist Algorithm

Input : Control flow Graph $G_P = \langle \mathcal{L}, E, i \rangle$ and Cache Trace Abstraction A
Output : Abstract Sticky Trace Collecting Semantics $\widehat{StickyCol}_A(P_{ins})$

- 1 $\widehat{S} := \widehat{Init}_A$
- 2 $worklist := \{\langle i, l \rangle \in E\}$
- 3 **while** *exists* $\langle l, l' \rangle \in worklist$ **do**
- 4 remove $\langle l, l' \rangle$ from $worklist$
- 5 $t := update_A^\#(\widehat{S}(l), eff_{\mathcal{L}}(l, l'))$
- 6 **if** $t \not\sqsubseteq_A \widehat{S}(l')$ **then**
- 7 $\widehat{S}(l') := t \sqcup_A \widehat{S}(l')$
- 8 $worklist := worklist \cup \{\langle l', l'' \rangle \in E\}$
- 9 **end**
- 10 **end**
- 11 **return** \widehat{S}

Worklist algorithms maintain a set of edges, stored in the variable `worklist` algorithm, whose source locations have been modified, and which thus have to be (re-)evaluated. Initially, only those edges emanating from the start node i of the control flow graph need to be evaluated. Thus `worklist` is initialized to edges emanating from i in line 2 of the algorithm. While there are edges to (re-)evaluate, the algorithm picks one of these edges, and removes it from `worklist` (lines 3 and 4). If the value $update_A^\#(\widehat{S}(l), eff_{\mathcal{L}}(l, l'))$ computed (line 5) for an edge is not covered (line 6) by the abstract trace $\widehat{S}(l')$ stored for its target location, then $\widehat{S}(l')$ is updated (line 7), and all edges emanating from l' need to be recomputed, and are thus added to `worklist` (line 8).

The performance of worklist algorithms depends on the *iteration strategy*. If the worklist contains multiple edges, the iteration strategy determines which edge to pick next. Nielson et al. [24, Chapter 6.1] discuss various iteration strategies and their performance characteristics.

4.3 On the Relative Precision of Different Cache Trace Abstractions

In Section 5 we introduce various basic approaches to persistence analysis as well as ways of combining basic approaches to obtain more precise combined analyses. In addition to proving these approaches correct, we also characterize their relative precision, based on the following notion of precision:

03:12 The Semantic Foundations and a Landscape of Cache-Persistence Analyses

► **Definition 5** (Precision). *Given two cache trace abstractions A and B , we say that A is at least as precise as B , denoted by $A \succeq B$, if A classifies each block as persistent that B classifies as persistent:*

$$\forall P_{ins}, \forall b : \text{classify}_B^\#(P_{ins}, b) \Rightarrow \text{classify}_A^\#(P_{ins}, b).$$

We say that A is more precise than B , denoted by $A \succ B$, if $A \succeq B$, but $B \not\succeq A$. If neither $A \succeq B$ nor vice versa, we say that A and B are incomparable.

Note that \succeq is a non-strict partial order, i.e., it is reflexive, antisymmetric, and transitive. Its strict counterpart \succ is a strict partial order, i.e., it is irreflexive, asymmetric, and transitive.

One way of showing $A \succeq B$ is to show that B is a sound approximation of A , just like we show that individual domains soundly approximate the concrete trace collecting semantics. This approach yields the following two theorems, which mirror Theorems 3 and 4:

► **Theorem 6** (Approximation of Abstract Semantics*). *Given two cache trace abstractions A and B , and a function $\gamma_{B \rightarrow A} : C_B^\# \rightarrow C_A^\#$ that satisfies the following conditions:*

$$\widehat{\mathcal{I}}_A \subseteq \gamma_{B \rightarrow A}(\widehat{\mathcal{I}}_B), \quad (11)$$

$$\forall \widehat{S}, \widehat{T} \in C_B^\# : \widehat{S} \sqsubseteq_B \widehat{T} \Rightarrow \gamma_{B \rightarrow A}(\widehat{S}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{T}), \quad (12)$$

$$\forall \widehat{S} \in C_B^\#, b \in \mathcal{B} : \text{update}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}), b) \sqsubseteq_A \gamma_{B \rightarrow A}(\text{update}_B^\#(\widehat{S}, b)). \quad (13)$$

Then, B 's abstract semantics soundly approximates its more concrete counterpart:

$$\widehat{\text{StickyCol}}_A(P_{ins}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})), \quad (14)$$

where $\gamma_{B \rightarrow A}$ is lifted to the abstract sticky trace collecting semantics as follows:

$$\gamma_{B \rightarrow A}(\widehat{S}) = \lambda l \in \mathcal{L}. \gamma_{B \rightarrow A}(\widehat{S}(l)).$$

► **Theorem 7** (Precision). *Given cache trace abstractions A, B and a function $\gamma_{B \rightarrow A}$ that satisfies conditions (11), (12), and (13) from Theorem 6, and further*

$$\forall \widehat{S} \in C_B^\#, b \in \mathcal{B} : \text{classify}_B^\#(\widehat{S}, b) \Rightarrow \text{classify}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}), b), \quad (15)$$

$$\forall \widehat{S}, \widehat{T} \in C_A^\#, b \in \mathcal{B} : \widehat{S} \sqsubseteq_A \widehat{T} \Rightarrow \left(\text{classify}_A^\#(\widehat{T}, b) \Rightarrow \text{classify}_A^\#(\widehat{S}, b) \right). \quad (16)$$

Then, A is at least as precise as B , i.e., $A \succeq B$.

Proof. From Theorem 6 we have that $\widehat{\text{StickyCol}}_A(P_{ins}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))$, which by definition is equivalent to $\forall l \in \mathcal{L} : \widehat{\text{StickyCol}}_A(P_{ins})(l) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})(l))$.

Assume $\text{classify}_B^\#(P_{ins}, b)$ for an arbitrary b :

$$\begin{aligned} \text{classify}_B^\#(P_{ins}, b) &\Leftrightarrow \forall l \in \mathcal{L} : \text{classify}_B^\#(\widehat{\text{StickyCol}}_B(P_{ins})(l), b) \\ &\stackrel{(15)}{\Rightarrow} \forall l \in \mathcal{L} : \text{classify}_A^\#(\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})(l)), b) \\ &\stackrel{(*)}{\Rightarrow} \forall l \in \mathcal{L} : \text{classify}_A^\#(\widehat{\text{StickyCol}}_A(P_{ins})(l), b) \\ &\Leftrightarrow \text{classify}_A^\#(P_{ins}, b) \end{aligned}$$

(*) follows from (16) and the fact that $\forall l \in \mathcal{L} : \widehat{\text{StickyCol}}_A(P_{ins})(l) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})(l))$. ◀

Proving that a sound cache trace abstraction A is at least as precise as cache trace abstraction B also proves B 's soundness:

► **Theorem 8** (Soundness of Persistence Classification). *Given two cache trace abstractions A and B . If A is sound, and A is at least as precise as B , then B is also sound.*

Proof. We need to show that $\text{classify}_B^\#(P_{ins}, b) := \forall l \in \mathcal{L} : \text{classify}_B^\#(\widehat{\text{StickyCol}}_B(P_{ins})(l), b)$ implies the persistence of memory block b in program P_{ins} .

Assume $\text{classify}_B^\#(P_{ins}, b)$. Because A is at least as precise as B this implies $\text{classify}_A^\#(P_{ins}, b)$. As A is sound, this implies the persistence of memory block b in program P_{ins} . ◀

While we give independent soundness proofs for all persistence analyses introduced in Section 5, in some cases the relative precision results constitute alternative soundness proofs based on the above theorem.

5 Instantiations of the Analysis Framework: Abstractions of Cache Traces

In this section, we explain and prove correct existing and new abstractions of cache traces for cache-persistence analysis.

Paraphrasing the soundness condition from Theorem 4, a memory block is persistent, if it is guaranteed to remain in the cache *in case* it has been accessed. This suggests that persistence analyses should maintain information about memory blocks *under the condition* that the memory blocks have been accessed. All sound persistence analyses can be seen as maintaining such information as we will see below.

Before describing particular analysis domains let us characterize under which conditions a memory block is guaranteed to be cached under LRU replacement. To this end, we first define the set `LRUCACHETRACES`, which consists of all cache traces that are possible under LRU replacement, assuming an arbitrary initial cache state and an arbitrary sequence of memory access:

$$\begin{aligned} \text{LRUCACHETRACES} := \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid c_0 \in \mathcal{C} \wedge \forall i, 0 \leq i < n : b_i \in \mathcal{B} \wedge \\ c_{i+1} = \text{update}_C^{\text{LRU}}(c_i, b_i) \wedge h_i = \text{eff}_C^{\text{LRU}}(c_i, b_i)\} \end{aligned} \quad (17)$$

The following lemma precisely captures when a memory block is guaranteed to be cached under LRU replacement:

► **Lemma 9** (Persistence under LRU*). *Consider an arbitrary cache trace $c_0\langle b_0, h_0 \rangle c_1\langle b_1, h_1 \rangle \dots c_n \in \text{LRUCACHETRACES}$. Then $c_n(b_0) < k$, if $|\{b_i \mid 0 \leq i < n\}| \leq k$.*

In other words, after a block b is accessed, this block is guaranteed to be cached as long as less than k distinct conflicting blocks have been accessed.

In Section 5.1, we discuss basic abstractions for persistence analysis. These abstractions either bound the *number* of conflicting blocks or overapproximate the *set* of conflicting blocks for each memory block. As we will see, these two approaches are incomparable, i.e., neither of the two dominates the other in terms of precision.

In Section 5.2, we then discuss how to combine these basic abstractions in order to obtain more precise analyses. By exchanging information during analysis time, such combinations go beyond simply running two incomparable analyses in parallel. As a consequence they may classify memory blocks as persistent that none of the basic abstractions would be able to classify as persistent on its own.

5.1 Basic Abstractions

5.1.1 Global-CS: Global May-Conflict Set

If at most k distinct memory blocks may be accessed in a given cache set, then, following Lemma 9, none of these blocks may be evicted after entering the cache. Thus, the *global may-conflict set*

analysis, abbreviated to *Global-CS*, overapproximates the set of memory blocks that may have been accessed, and so its abstract traces are sets of memory blocks:

$$C_{Global-CS}^{\#} := 2^{\mathcal{B}} \quad (18)$$

An abstract trace \widehat{S} represents all concrete cache traces that may be formed by accessing blocks from the set \widehat{S} :⁷

$$\gamma_{Global-CS}(\widehat{S}) := \text{LRUCACHETRACES} \cap \{c_0 \langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \widehat{S}\} \quad (19)$$

At program start, no accesses have yet been performed, and so the initial abstract trace is the empty set, which by $\gamma_{Global-CS}$ represents exactly cache traces of length 0, in other words, all initial cache states:

$$\mathcal{I}_{Global-CS} := \emptyset \quad (20)$$

In order to soundly approximate all memory blocks that *may* have been accessed, abstract traces are joined by taking their union:

$$\widehat{S} \sqsubseteq_{Global-CS} \widehat{T} \Leftrightarrow \widehat{S} \subseteq \widehat{T} \quad \widehat{S} \sqcup_{Global-CS} \widehat{T} := \widehat{S} \cup \widehat{T} \quad (21)$$

Upon a memory access, the accessed block is simply added to the abstract trace:

$$\text{update}_{Global-CS}^{\#}(\widehat{S}, b) := \widehat{S} \cup \{b\} \quad (22)$$

Following Lemma 9, as long as at most k memory blocks have been accessed, *any* block must still be cached *if* it has been accessed:

$$\text{classify}_{Global-CS}^{\#}(\widehat{S}, b) := b \in \widehat{S} \Rightarrow |\widehat{S}| \leq k \quad (23)$$

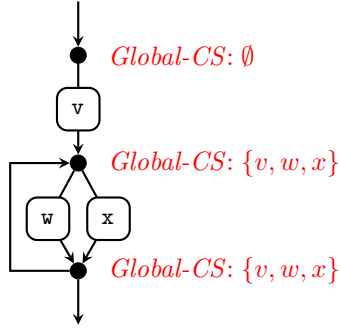
See Figure 1 for a small example of the *Global-CS* analysis. The figure shows the fixpoint of the set of equations determined by the update and join functions of the analysis on the given control flow graph. At any point in the loop, each of the blocks v, w , and x may have been accessed. In a cache of associativity 3 or higher these blocks would all be declared as persistent by *Global-CS*. On the other hand, while w and x are persistent even in a cache of associativity 2, the analysis is unable to detect this.

► **Theorem 10** (Soundness of Global May-Conflict Set*). *Global-CS is a sound persistence analysis.*

If the set of memory blocks \mathcal{B} is finite, then $C_{Global-CS}^{\#} = 2^{\mathcal{B}}$ is also finite, and thus it satisfies the ascending chain condition [18, 9], which guarantees termination of the fixpoint iteration to compute the abstract semantics. Following Davey and Priestley [9], we define the length of an ascending chain as the number of elements of the chain minus one. The length of a chain thus corresponds to the number of steps a fixpoint iteration takes to traverse it. The longest ascending chains in $C_{Global-CS}^{\#}$ are of length $|\mathcal{B}|$, starting from \emptyset and ending in \mathcal{B} .

For readability we limit our exposition to the analysis of fully-associative caches throughout the article. The extension to set-associative caches is straightforward: Either, a separate set of blocks should be maintained for each cache set, or the classification function $\text{classify}_{Global-CS}^{\#}(\widehat{S}, b)$ should count only those blocks mapping to the same cache set as b .

⁷ We have found two alternative approaches to formalize the cache trace abstractions discussed in this article: (1) by constraints on the memory access trace, and (2) by constraints on the resulting final cache states of the traces. The advantage of approach (1) is that, except for the persistence classification function, it can be proved correct *independently* of the employed cache replacement policy. In approach (2) the final cache states are constrained implicitly by considering only cache traces that are compatible with the cache replacement policy. Proving correct the persistence classification function then requires invoking a property of LRU, which is condensed in Lemma 9.



■ **Figure 1** Example illustrating *Global-CS*.

5.1.2 Block-CS: Block-wise May-Conflict Set

As soon as more than k memory blocks are accessed by a program, no block can be classified persistent by *Global-CS*. In such cases, many memory blocks may actually still be persistent: Following Lemma 9, a block is persistent if at most $k - 1$ distinct other blocks are accessed between any two accesses to the block itself.

The *block-wise may-conflict set* analysis, abbreviated to *Block-CS*, maintains a separate conflict set for each memory block, rather than a single global conflict set:

$$C_{Block-CS}^{\#} := \mathcal{B} \rightarrow 2^{\mathcal{B}} \quad (24)$$

Then, an abstract trace \widehat{S} represents all concrete cache traces in which, following the final access to a block, only blocks from its conflict set may have been accessed:

$$\begin{aligned} \gamma_{Block-CS}(\widehat{S}) := \text{LRUCACHETRACES} \cap & \quad (25) \\ \{s = c_0 \langle b_0, h_0 \rangle \dots c_n \mid \forall i, 0 \leq i < n : b_i \in CS_{i+1}(s) \vee CS_i(s) \subseteq \widehat{S}(b_i)\}, & \end{aligned}$$

where $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_n) := \{b_j \mid i \leq j < n\}$.

Similarly to the global conflict-set case, the initial abstract trace assigns the empty conflict set to each block, which by the concretization function above exactly represents all cache traces of length 0. At joins, the union of the conflict sets is taken in order to overapproximate the conflicting blocks:

$$\widehat{\mathcal{I}}_{Block-CS} := \lambda b \in \mathcal{B}. \emptyset \quad (26)$$

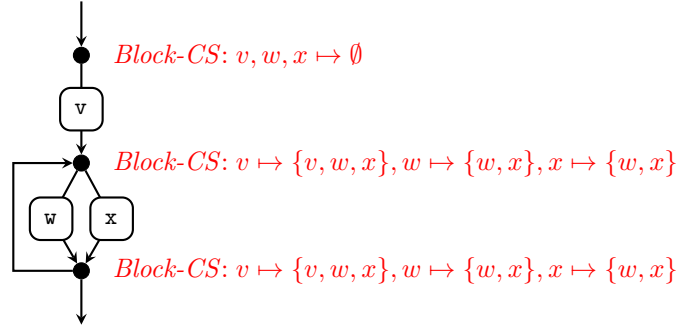
$$\widehat{S} \sqsubseteq_{Block-CS} \widehat{T} := \Leftrightarrow \forall b \in \mathcal{B} : \widehat{S}(b) \subseteq \widehat{T}(b) \quad \widehat{S} \sqcup_{Block-CS} \widehat{T} := \lambda b \in \mathcal{B}. \widehat{S}(b) \cup \widehat{T}(b) \quad (27)$$

Upon a memory access, the accessed block b is added to the conflict sets of all memory blocks that may have been accessed, i.e. blocks for which $\widehat{S}(b') \neq \emptyset$, and crucially b 's conflict set is reset to contain only b . This is where the analysis profits from maintaining separate conflict sets for each block.

$$\text{update}_{Block-CS}^{\#}(\widehat{S}, b) := \lambda b'. \begin{cases} \emptyset & : b' \neq b \wedge \widehat{S}(b') = \emptyset \\ \{b\} & : b' = b \\ \widehat{S}(b') \cup \{b\} & : b' \neq b \wedge \widehat{S}(b') \neq \emptyset \end{cases} \quad (28)$$

Finally, a block is locally classified as persistent, if its conflict set, which includes the block itself if it may have been accessed, is guaranteed to contain at most k blocks:

$$\text{classify}_{Block-CS}^{\#}(\widehat{S}, b) := |\widehat{S}(b)| \leq k \quad (29)$$



■ **Figure 2** Example illustrating *Block-CS*.

Figure 2 shows the result of running *Block-CS* on the same example program as *Global-CS* in the previous section. By tracking each block’s conflict set separately – in contrast to *Global-CS* – the analysis is able to determine that w and x are persistent in a cache of associativity 2 as their conflict sets both only contain w and x .

► **Theorem 11** (Soundness of Block-wise May-Conflict Set^{*}). *Block-CS is a sound persistence analysis.*

If the set of memory blocks \mathcal{B} is finite, then $C_{Block-CS}^\# = \mathcal{B} \rightarrow 2^{\mathcal{B}}$ is also finite, and thus it satisfies the ascending chain condition [18, 9], which guarantees termination of the fixpoint iteration to compute the abstract semantics. More precisely, the longest ascending chains are of length $|\mathcal{B}|^2$: Each of the longest ascending chains begins with the bottom element, i.e., the least element of the complete lattice, $\lambda b \in \mathcal{B}.\emptyset$, mapping each block to an empty conflict set, and ends in the top element of the complete lattice, $\lambda b \in \mathcal{B}.\mathcal{B}$, mapping each block to the greatest possible conflict set, consisting of all $|\mathcal{B}|$ memory blocks. In each step of any strictly ascending chain, the conflict set of at least one of the memory blocks needs to grow by at least one block, while none of the conflict sets may shrink. Thus, any ascending chain may contain at most $|\mathcal{B}|^2$ that are greater than the bottom element.

We note that due to the classification condition in (29), it is not necessary to distinguish conflict sets that have more than k elements. Thus, for efficiency, implementations of *Block-CS* should represent all conflict sets with more than k elements by a single unique representative. This has no effect on analysis correctness or precision but reduces the maximum length of ascending chains to $(k + 1) \cdot |\mathcal{B}|$.

► **Theorem 12** (*Block-CS vs. Global-CS*). *Block-CS is more precise than Global-CS.*

Proof. In Section 5.1.4 we introduce the *conditional may analysis*, abbreviated to *C-May*. In the same section, in Theorems 16 and 17 we show that *Block-CS* is more precise than *C-May*, and that *C-May* is more precise than *Global-CS*. As the more-precise relation is transitive these two statements imply the theorem. ◀

We note that by Theorem 8 the above two theorems also imply the correctness of *Global-CS*.

5.1.3 C-Must: Conditional Must Analysis

The block-wise may-conflict set approach may lose precision at joins, as the union of the conflict sets needs to be taken. Instead of overapproximating the *set* of conflicting blocks, the *conditional*

must analysis, abbreviated to *C-Must*, bounds the *number* of conflicting blocks. Then it is safe to take the maximum rather than the sum of the bounds at joins.

The *conditional must analysis* thus maintains a bound on the size of the conflict sets of each memory block. A bound of 0 is used to encode that a block is guaranteed not to have been accessed so far, in which case 0 correctly bounds the size of its conflict set. Further, for the purpose of classifying a memory block as persistent, it is not useful to track the size of a block's conflict set beyond k . Therefore, all bounds greater than k are collapsed to ∞ :

$$C_{C-Must}^{\#} := \mathcal{B} \rightarrow \{0, 1, \dots, k, \infty\} \quad (30)$$

Its concretization is very similar to that of the *block-wise may-conflict set analysis*. Instead of overapproximating a block's conflict set, the size of its conflict set is bounded:

$$\begin{aligned} \gamma_{C-Must}(\widehat{S}) := & \text{LRUCACHETRACES} \cap \\ & \{s = c_0\langle b_0, h_0 \rangle \dots c_n \mid \forall i, 0 \leq i < n : b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)\}, \end{aligned} \quad (31)$$

where, as before, $CS_i(c_0\langle b_0, h_0 \rangle \dots c_n) := \{b_j \mid i \leq j < n\}$.

In the initial abstract cache trace, each block is assigned a bound of 0. By the concretization function this represents all cache traces of length 0, i.e., traces consisting of an arbitrary initial cache state but no memory accesses.

$$\widehat{\mathcal{I}}_{C-Must} := \lambda b \in \mathcal{B}. 0 \quad (32)$$

The advantage of the *conditional must analysis* over the *block-wise may-conflict set analysis* is that the maximum of the bounds can be taken at joins, rather than their sum:

$$\widehat{S} \sqsubseteq_{C-Must} \widehat{T} \Leftrightarrow \forall b \in \mathcal{B} : \widehat{S}(b) \leq \widehat{T}(b) \quad \widehat{S} \sqcup_{C-Must} \widehat{T} := \lambda b \in \mathcal{B}. \max\{\widehat{S}(b), \widehat{T}(b)\} \quad (33)$$

Upon a memory access, the sizes of the conflict sets of all memory blocks that may have been accessed, i.e. for which $\widehat{S}(b') > 0$ holds, may increase by 1, while the accessed block's conflict set includes only itself, and so its size bound may be reset to 1.

$$\text{update}_{C-Must}^{\#}(\widehat{S}, b) := \lambda b'. \begin{cases} 0 & : b' \neq b \wedge \widehat{S}(b') = 0 \\ 1 & : b' = b \\ \widehat{S}(b') + 1 & : b' \neq b \wedge 0 < \widehat{S}(b') < k \\ \infty & : b' \neq b \wedge k \leq \widehat{S}(b') \end{cases} \quad (34)$$

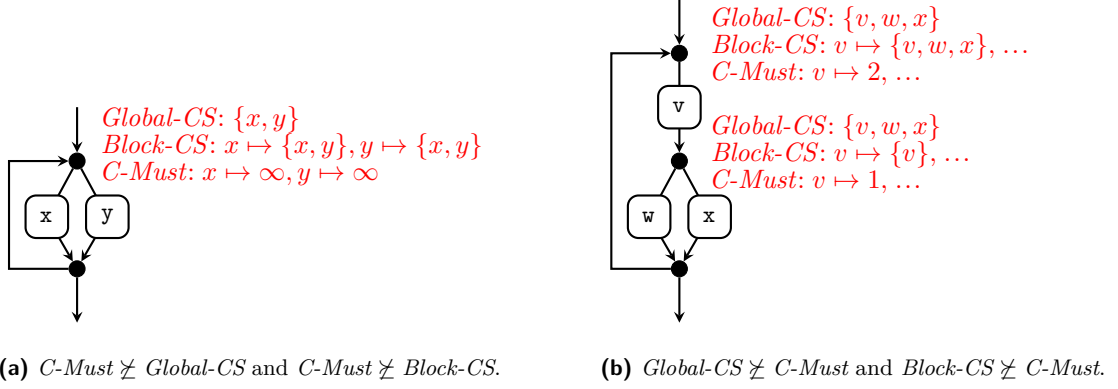
Because the conflict sets are not tracked explicitly, a single memory block may increase the bound of another block multiple times. In such scenarios the *block-wise may-conflict set analysis* may be more precise.

A memory block is locally classified as persistent, if its conflict set is guaranteed to contain less than k blocks:

$$\text{classify}_{C-Must}^{\#}(\widehat{S}, b) := \widehat{S}(b) \leq k \quad (35)$$

► **Theorem 13** (Soundness of Conditional Must*). *C-Must is a sound persistence analysis.*

If the set of memory blocks \mathcal{B} is finite, then $C_{C-Must}^{\#} = \mathcal{B} \rightarrow \{0, 1, \dots, k, \infty\}$ is also finite, and thus termination of the fixpoint iteration to compute the abstract semantics is guaranteed. The longest ascending chains are of length $(k + 1) \cdot |\mathcal{B}|$: Each of the longest ascending chains begins with the bottom element, i.e., the least element of the complete lattice, $\lambda b. 0$. In each step of any strictly ascending chain, the bound for at least one block needs to grow, while none of the bounds may shrink. As the bound of each block may grow at most $k + 1$ times, no strictly ascending chain may be of length greater than $(k + 1) \cdot |\mathcal{B}|$.



■ **Figure 3** Examples illustrating the incomparability of $C\text{-Must}$ with Global-CS and Block-CS .

► **Theorem 14** (*Global-CS vs. Block-CS*). $C\text{-Must}$ is incomparable to Global-CS and Block-CS .

Proof. Consider the examples in Figures 3a and 3b. Assume a cache with associativity 2. In the first example, both x and y are classified as persistent by Global-CS and Block-CS , in contrast to $C\text{-Must}$. This is because $C\text{-Must}$ may account for the same conflicting block multiple times. On the other hand, in the second example, $C\text{-Must}$ classifies v as persistent, while Block-CS and Global-CS do not. Here, unlike Global-CS and Block-CS , $C\text{-Must}$ is able to capture that in any trace either w or x conflicts with v , but never both. ◀

From the description of the $C\text{-Must}$ analysis it may not be obvious why we choose to call it the *conditional must analysis*. The reason is that it strongly resembles the original *must analysis* by Ferdinand and Wilhelm [12]. The bound on the size of the conflict set of each memory block corresponds to a bound on a memory block’s age in the final state of a cache trace *under the condition* that the block has been accessed at least once.

5.1.4 C-May: Conditional May Analysis

Somewhat surprisingly it is also possible to classify memory blocks as persistent with an analysis that determines lower rather than upper bounds on the sizes of memory blocks’ conflict sets. The *conditional may analysis*, abbreviated to $C\text{-May}$, maintains a lower bound on the size of the conflict set of each memory block. These lower bounds need to hold only for blocks that have been accessed at least once during program execution. In this sense the bounds are *conditional*. For the purpose of classifying memory blocks as persistent, it is not useful to track the size of a block’s conflict set beyond k . Therefore, all lower bounds greater than k are collapsed to $k + 1$. In addition, ∞ is used to indicate that a block has never been accessed: in such cases, ∞ is a correct lower bound on the block’s conflict set on the set of traces on which it has been accessed, which is empty.

$$C_{C\text{-May}}^\# := \mathcal{B} \rightarrow \{1, \dots, k, k + 1, \infty\} \quad (36)$$

Its concretization is very similar to that of the *conditional must analysis*. Instead of bounding the size of a block’s conflict set from above, it is bounded from below:

$$\gamma_{C\text{-May}}(\widehat{S}) := \text{LRUCACHETRACES} \cap \{s = c_0\langle b_0, h_0 \rangle \dots c_n \mid \forall i : 0 \leq i < n : b_i \in \text{CS}_{i+1}(s) \vee |\text{CS}_i(s)| \geq \widehat{S}(b_i)\}, \quad (37)$$

where, as before, $\text{CS}_i(c_0\langle b_0, h_0 \rangle \dots c_n) := \{b_j \mid i \leq j < n\}$.

In the initial abstract cache trace, each block is assigned a bound of ∞ . By the concretization function this represents all cache traces of length 0, i.e., traces consisting of an arbitrary initial cache state but no memory accesses.

$$\widehat{\mathcal{I}}_{C\text{-May}} := \lambda b \in \mathcal{B}. \infty \quad (38)$$

At joins the minimum of the lower bounds needs to be taken for each memory block:

$$\widehat{S} \sqsubseteq_{C\text{-May}} \widehat{T} : \Leftrightarrow \forall b \in \mathcal{B} : \widehat{S}(b) \geq \widehat{T}(b) \quad \widehat{S} \sqcup_{C\text{-May}} \widehat{T} := \lambda b \in \mathcal{B}. \min\{\widehat{S}(b), \widehat{T}(b)\} \quad (39)$$

Upon an access, the accessed block's conflict set shrinks to size 1 (case 1 in (40)). Other block's conflict sets may or may not grow (cases 2 and 3 in (40)). It is safe to increase the lower bound for memory block b' , if the previous lower bound for the *accessed block* b was at least as high (case 3 below) as its own lower bound, which can be understood by the following case distinction:

1. Either b was actually contained in b' 's conflict set before the access. Then b 's conflict set is a strict subset of b' 's conflict set, and so $\widehat{S}(b) + 1 \geq \widehat{S}(b') + 1$ is a lower bound on the size of b' 's conflict set.
2. Or b was not contained in b' 's conflict set before the access. Then b' 's conflict set grows by 1 due to the access to b and thus $\widehat{S}(b') + 1$ is a correct lower bound following the access.

Lower bounds beyond $k + 1$ are not distinguished (case 4 in (40)), and finally, blocks that are guaranteed not to have been accessed yet retain a lower bound of ∞ :

$$\text{update}_{C\text{-May}}^{\#}(\widehat{S}, b) := \lambda b'. \begin{cases} 1 & : b' = b \\ \widehat{S}(b') & : b' \neq b \wedge \widehat{S}(b) < \widehat{S}(b') \\ \widehat{S}(b') + 1 & : b' \neq b \wedge \widehat{S}(b) \geq \widehat{S}(b') \wedge \widehat{S}(b') \leq k \\ k + 1 & : b' \neq b \wedge \widehat{S}(b) \geq \widehat{S}(b') \wedge \widehat{S}(b') = k + 1 \\ \infty & : b' \neq b \wedge \widehat{S}(b) = \infty \end{cases} \quad (40)$$

Maybe surprisingly⁸, it is possible to classify memory blocks as persistent using the lower bounds derived by the *conditional may analysis*. The intuition behind the classification function is the following: In any concrete cache trace, the conflict sets of the i most-recently-used memory blocks have sizes 1 to i . So only blocks with a lower bound less than or equal to i may be among these i most-recently-used blocks. If there are at most $i \leq k$ memory blocks with a lower bound less than or equal to i , then at most i blocks compete for the first i locations in the cache. So all of these blocks *must* be cached if they have previously been accessed:

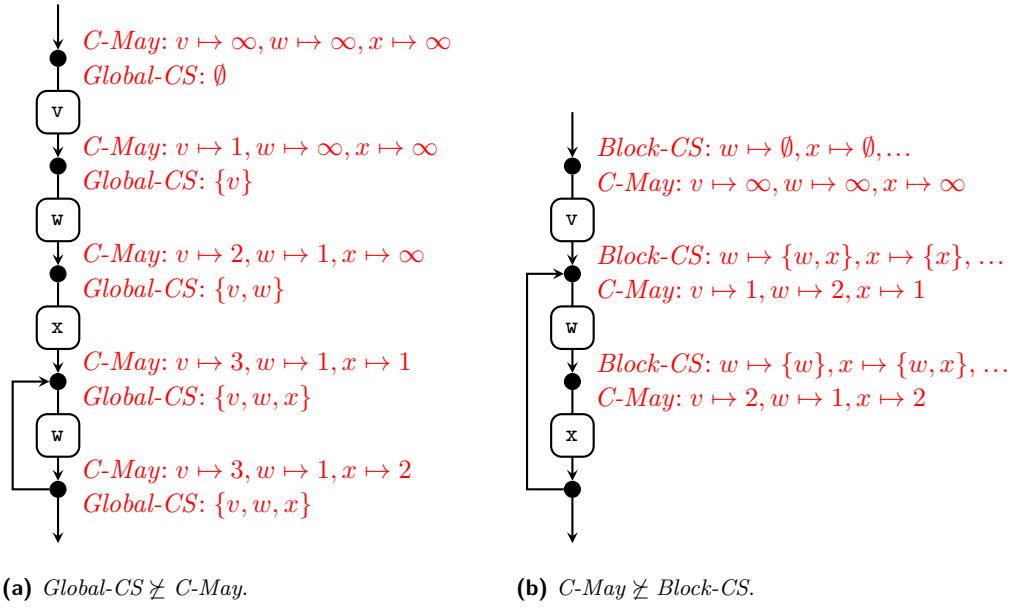
$$\text{classify}_{C\text{-May}}^{\#}(\widehat{S}, b) := (\widehat{S}(b) = \infty) \vee (\exists i \leq k : |C_i(\widehat{S}, b)| < i), \quad (41)$$

where $C_i(\widehat{S}, b) := \{b' \in \mathcal{B} \mid b' \neq b \wedge \widehat{S}(b') \leq i\}$ is the set of memory blocks with a lower bound less than or equal to i other than block b . A detailed proof of correctness is given in the proof of the following theorem:

► **Theorem 15** (Soundness of Conditional May*). *C-May is a sound persistence analysis.*

If the set of memory blocks \mathcal{B} is finite, then $C_{C\text{-May}}^{\#} = \mathcal{B} \rightarrow \{1, \dots, k, k + 1, \infty\}$ is also finite, and thus termination of the fixpoint iteration to compute the abstract semantics is guaranteed. The longest ascending chains are of length $(k + 1) \cdot |\mathcal{B}|$, which can be seen following the same train of thought as in the case of *C-Must*.

⁸ Classifying a memory block as persistent following Lemma 9 requires deriving an *upper* bound on the size of a block's conflict set. Thus it may be surprising that the *lower* bounds on blocks' conflict sets determined by *C-May* can be used for this purpose.



■ **Figure 4** Examples illustrating that $Global-CS \not\subseteq C-May$ and $C-May \not\subseteq Block-CS$.

Let us consider two example programs and their analysis using $C-May$ in Figure 4. On the left, in Figure 4a, $C-May$ is able to classify both w and x as persistent in a cache of associativity 2, while none of the blocks are determined persistent by $Global-CS$. For $C-May$, the figure shows the lower bounds on each block's conflict set at each program point. There are at most two blocks with a lower bound of 2 at any program point and thus these blocks are guaranteed to be cached if they have been accessed. On the right, in Figure 4b, $C-May$ is unable to classify w and x as persistent in a cache of associativity 2, while $Block-CS$ is. This is because all three blocks v , w , and x have a lower bound less than or equal to 2 within the loop.

► **Theorem 16** ($C-May$ vs. $Global-CS^*$). $C-May$ is more precise than $Global-CS$.

► **Theorem 17** ($Block-CS$ vs. $C-May^*$). $Block-CS$ is more precise than $C-May$.

By Theorem 8, Theorems 11 and 17 also imply the correctness of $C-May$. Also, due to the transitivity of the more-precise relation, Theorems 16 and 17 together imply Theorem 12, which states that $Block-CS$ is more precise than $Global-CS$.

5.2 Combinations of Basic Abstractions

We have seen four basic cache persistence abstractions: $Block-CS$, $C-May$, $Global-CS$, and $C-Must$. Among these, $Block-CS$ is more precise than $C-May$, which in turn is more precise than $Global-CS$. On the other hand, $C-Must$ is incomparable to $Block-CS$, $C-May$, and $Global-CS$.

To obtain more precise analysis results, it may be beneficial to combine incomparable cache trace abstractions with each other. In Section 5.2.1 we show how to construct the *direct product* of two arbitrary abstractions and show that the direct product of two incomparable abstractions A and B is more precise than A and B individually.

To further increase analysis precision, Section 5.2.2 introduces two ways to exchange information between two abstractions A and B , which may yield cache trace abstractions that are more precise than the direct product of A and B . In the remainder of the section, we then show how to exploit

these two ways of information exchange to build more precise analyses for various combinations of basic analyses.

5.2.1 Direct Product of Cache Trace Abstractions

The direct product of two persistence analyses corresponds to running the two analyses in parallel and classifying a block as persistent if at least one of the two analyses is able to classify the block persistent. Formally, it is defined as follows:

► **Definition 18** (Direct Product). *The direct product $A \times B$ of two persistence analyses A and B is the tuple $A \times B = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}}_{A \times B}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, \text{update}_{A \times B}^\#, \text{classify}_{A \times B}^\# \rangle$ with*

$$\begin{aligned} C_{A \times B}^\# &:= C_A^\# \times C_B^\#, \\ \gamma_{A \times B}(\widehat{S}_A, \widehat{S}_B) &:= \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B), \\ \widehat{\mathcal{I}}_{A \times B} &:= \langle \widehat{\mathcal{I}}_A, \widehat{\mathcal{I}}_B \rangle, \\ \langle \widehat{S}_A, \widehat{S}_B \rangle \sqsubseteq_{A \times B} \langle \widehat{T}_A, \widehat{T}_B \rangle &:\Leftrightarrow \widehat{S}_A \sqsubseteq_A \widehat{T}_A \wedge \widehat{S}_B \sqsubseteq_B \widehat{T}_B, \\ \langle \widehat{S}_A, \widehat{S}_B \rangle \sqcup_{A \times B} \langle \widehat{T}_A, \widehat{T}_B \rangle &:= \langle \widehat{S}_A \sqcup_A \widehat{T}_A, \widehat{S}_B \sqcup_B \widehat{T}_B \rangle, \\ \text{update}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) &:= \langle \text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b) \rangle, \\ \text{classify}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) &:= \text{classify}_A^\#(\widehat{S}_A, b) \vee \text{classify}_B^\#(\widehat{S}_B, b). \end{aligned}$$

► **Theorem 19** (Soundness of Direct Product*). *The direct product $A \times B$ of two sound persistence analyses A and B that satisfy (6), (7), (8), and (10) is a sound persistence analysis.*

In the proof in the appendix, we show that $A \times B$ satisfies the conditions of Theorems 3 and 4, i.e., (6), (7), (8), and (10), and is thus a sound persistence abstraction.

We note that if both A and B satisfy the ascending chain condition, then so does $A \times B$. Thus, persistence analysis with a direct product of two analyses terminates if both constituent analyses are guaranteed to terminate. Moreover, if the lengths of the ascending chains of A and B are bounded by l_A and l_B , then the length of $A \times B$'s longest ascending chains is bounded by $l_A + l_B$.

► **Theorem 20** (Precision of Direct Product). *The direct product $A \times B$ of two persistence analyses A and B is at least as precise as A and B , i.e., $A \times B \succeq A$ and $A \times B \succeq B$.*

Proof. This follows from the fact that the two constituents of $A \times B$ exactly mirror A and B , respectively, and from the fact that

$$\begin{aligned} \text{classify}_A^\#(\widehat{S}_A, b) &\Rightarrow \text{classify}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) = \text{classify}_A^\#(\widehat{S}_A, b) \vee \text{classify}_B^\#(\widehat{S}_B, b), \\ \text{classify}_B^\#(\widehat{S}_B, b) &\Rightarrow \text{classify}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) = \text{classify}_A^\#(\widehat{S}_A, b) \vee \text{classify}_B^\#(\widehat{S}_B, b). \quad \blacktriangleleft \end{aligned}$$

It is not useful to construct the direct product of two analyses A and B if $A \succ B$, as the result is not going to be more precise than A . If, on the other hand, A and B are incomparable, their direct product will be more precise than both A and B :

► **Corollary 21** (Precision of Direct Product). *The direct product $A \times B$ of two incomparable persistence analyses A and B is more precise than A and B , i.e., $A \times B \succ A$ and $A \times B \succ B$.*

Proof. From Theorem 20 we already know that $A \times B \succeq A$ and $A \times B \succeq B$. Assume for a contradiction that $B \succeq A \times B$. By transitivity of \succeq this would imply $B \succeq A$, which contradicts the assumption that A and B are incomparable. Thus $B \not\succeq A \times B$ and so $A \times B \succ B$. The fact that $A \times B \succ A$ can be shown analogously. \blacktriangleleft

5.2.2 Domain Cooperation

To increase precision, it is sometimes possible for different analyses in a product to exchange information with each other. Here, we distinguish two ways in which such an information exchange can take place between two analyses A and B :

- *State reduction*: the analysis state of A is refined using the analysis state of B .
- *Cooperative update*: The abstract update function for A takes into account not only A 's analysis state but also B 's to compute a more precise successor state.

Below we state correctness conditions for state and update reductions.

► **Definition 22 (State Reduction)**. *Let A and B be persistence analyses. A reduction operator for A in the context of B is a function $red: C_A^\# \times C_B^\# \rightarrow C_A^\#$ that is reductive and that preserves concretizations, i.e., for all $\widehat{S}_A \in C_A^\#, \widehat{S}_B \in C_B^\#$:*

$$red(\widehat{S}_A, \widehat{S}_B) \sqsubseteq_A \widehat{S}_A, \quad (42)$$

$$\gamma_A(red(\widehat{S}_A, \widehat{S}_B)) \cap \gamma_B(\widehat{S}_B) = \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B). \quad (43)$$

A reduction operator can be used as follows to obtain a potentially more precise reduced update for the product of A and B :

► **Theorem 23 (State Reduction*)**. *Let A and B be sound persistence analyses that satisfy (6), (7), (8), and (10), and let red be a reduction operator for A in the context of B . Let the reduced update be defined as follows:*

$$red\text{-}upd(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) := (red(update_A^\#(\widehat{S}_A, b), update_B^\#(\widehat{S}_B, b)), update_B^\#(\widehat{S}_B, b))$$

Then, $A \times B' = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}}_{A \times B}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, red\text{-}upd, classify_{A \times B}^\# \rangle$ is a sound persistence analysis that is at least as precise as $A \times B$, i.e., $A \times B' \succeq A \times B$.

Sometimes, it is not possible to come up with a state reduction to transfer information between two domains A and B , but it is still possible to profit from the information in B during the update of A . We call such an update *cooperative*:

► **Definition 24 (Cooperative Update)**. *Let A and B be two persistence analyses. A cooperative update for A in the context of B is a function $coop\text{-}upd: (C_A^\# \times C_B^\#) \times \mathcal{B} \rightarrow C_A^\#$, such that:*

$$\begin{aligned} \forall \langle \widehat{S}_A, \widehat{S}_B \rangle \in C_A^\# \times C_B^\#, b \in \mathcal{B}: \\ \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_A(coop\text{-}upd(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) \end{aligned} \quad (44)$$

Given a cooperative update, it is straightforward to define the following reduced update for the product of A and B :

► **Theorem 25 (Cooperative Update)**. *Let A and B be sound persistence analyses that satisfy (6), (7), (8), and (10), and let $coop\text{-}upd$ be a cooperative update function for A in the context of B . Let the reduced update be defined as follows:*

$$red\text{-}upd(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) := (coop\text{-}upd(\langle \widehat{S}_A, \widehat{S}_B \rangle, b), update_B^\#(\widehat{S}_B, b))$$

Then, $A \times B' = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}}_{A \times B}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, red\text{-}upd, classify_{A \times B}^\# \rangle$ is a sound persistence analysis.

Proof. We know that $A \times B$ is a sound persistence analysis from Theorem 19. The only condition from Theorem 4 that involves the update function is (8). Thus all conditions but (8) are fulfilled by $A \times B'$ as they are fulfilled by $A \times B$.

For (8), we need to show:

$$\begin{aligned} \forall (\widehat{S}_A, \widehat{S}_B) \in C_{A \times B}^\#, b \in \mathcal{B} : \\ \{t.c \langle b, h \rangle c' \mid t.c \in \gamma_{A \times B}(\widehat{S}_A, \widehat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_{A \times B}(\text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) \\ = \gamma_A(\text{coop-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) \cap \gamma_B(\text{update}_B^\#(\widehat{S}_B, b)) \end{aligned} \quad (45)$$

By the soundness of B and by (44) we have

$$\begin{aligned} \forall \widehat{S}_A \in C_B^\#, b \in \mathcal{B} : \\ \{t.c \langle b, h \rangle c' \mid t.c \in \gamma_B(\widehat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_B(\text{update}_B^\#(\widehat{S}_B, b)) \end{aligned} \quad (46)$$

$$\begin{aligned} \forall (\widehat{S}_A, \widehat{S}_B) \in C_A^\# \times C_B^\#, b \in \mathcal{B} : \\ \{t.c \langle b, h \rangle c' \mid t.c \in \gamma_{A \times B}(\widehat{S}_A, \widehat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_A(\text{coop-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) \end{aligned} \quad (47)$$

Together, (46) and (47) imply (45), and thus (8). \blacktriangleleft

Given the definition of a cooperative update in Definition 24 it is not possible to conclude that the product $A \times B'$ from Theorem 25 is more precise than $A \times B$. However, it is relatively easy to see that this is indeed the case if $\text{coop-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) \sqsubseteq_A \text{update}_A^\#(\widehat{S}_A, b)$ for all $\widehat{S}_A, \widehat{S}_B$, and b .

5.2.3 State Reduction between C-Must and Block-CS

In terms of precision, the *block-wise may-conflict set* and the *conditional must* analyses are incomparable, as the former has more precise updates, while the latter has more precise joins. Here we show how to exchange information between the two analyses, by a state reduction, to achieve higher precision than the direct product of the two analyses would.

How can information be exchanged between the two domains? Clearly, the size of the may-conflict set of a block is also a bound on the number of conflicting blocks. Thus, we introduce the following reduction operation:

$$\text{reduce}_{C\text{-Must} \times \text{Block-CS}} \left(\widehat{S}_{C\text{-Must}}, \widehat{S}_{\text{Block-CS}} \right) := \lambda b \in \mathcal{B}. \min \left\{ \widehat{S}_{C\text{-Must}}(b), |\widehat{S}_{\text{Block-CS}}(b)| \right\} \quad (48)$$

► **Theorem 26** (Soundness of the State Reduction between *C-Must* and *Block-CS*). *The function $\text{reduce}_{C\text{-Must} \times \text{Block-CS}}$ is a reduction operator for *C-Must* in the context of *Block-CS*.*

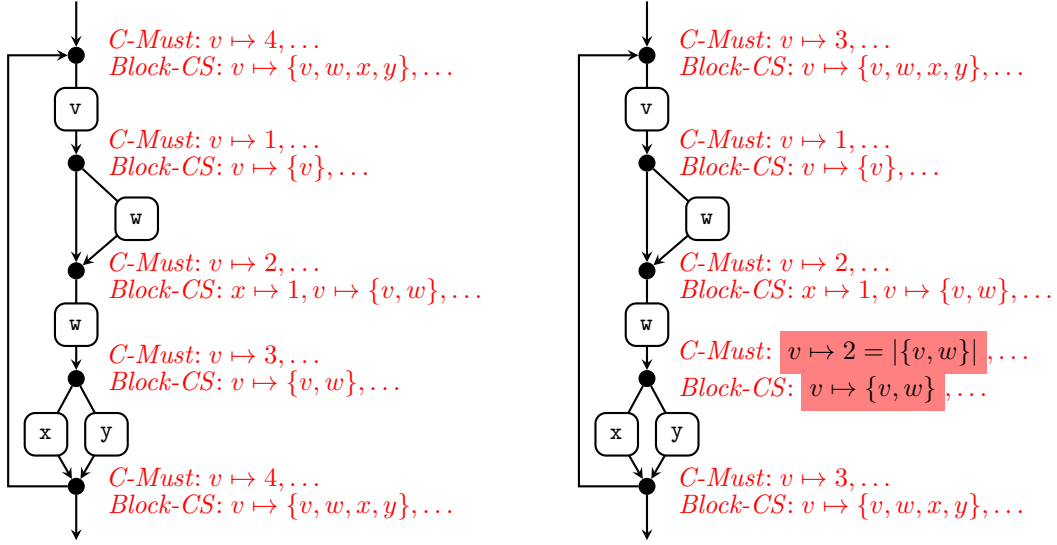
Proof. $\text{reduce}_{C\text{-Must} \times \text{Block-CS}}$ is reductive, as $\min \left\{ \widehat{S}_{C\text{-Must}}(b), |\widehat{S}_{\text{Block-CS}}(b)| \right\} \leq \widehat{S}_{C\text{-Must}}(b)$.

It remains to show that for all $\widehat{S}_1 \in C_{C\text{-Must}}^\#, \widehat{S}_2 \in C_{\text{Block-CS}}^\#$:

$$\gamma_{C\text{-Must} \times \text{Block-CS}} \left(\widehat{S}_1, \widehat{S}_2 \right) = \gamma_{C\text{-Must} \times \text{Block-CS}} \left(\text{reduce}_{C\text{-Must} \times \text{Block-CS}} \left(\widehat{S}_1, \widehat{S}_2 \right), \widehat{S}_2 \right).$$

As $\text{reduce}_{C\text{-Must} \times \text{Block-CS}}$ is reductive and $\gamma_{C\text{-Must} \times \text{Block-CS}}$ is monotone, we have

$$\gamma_{C\text{-Must} \times \text{Block-CS}} \left(\widehat{S}_1, \widehat{S}_2 \right) \supseteq \gamma_{C\text{-Must} \times \text{Block-CS}} \left(\text{reduce}_{C\text{-Must} \times \text{Block-CS}} \left(\widehat{S}_1, \widehat{S}_2 \right), \widehat{S}_2 \right).$$



(a) $C\text{-Must} \times Block\text{-CS}$ without domain cooperation. (b) $C\text{-Must} \times Block\text{-CS}$ with state reduction.

■ **Figure 5** Example illustrating $C\text{-Must} \times Block\text{-CS}$.

To show that $\gamma_{C\text{-Must} \times Block\text{-CS}}(\widehat{S}_1, \widehat{S}_2) \subseteq \gamma_{C\text{-Must} \times Block\text{-CS}}(\text{reduce}_{C\text{-Must} \times Block\text{-CS}}(\widehat{S}_1, \widehat{S}_2), \widehat{S}_2)$, assume for a contradiction that there is a trace $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ in $\gamma_{C\text{-Must} \times Block\text{-CS}}(\widehat{S}_1, \widehat{S}_2)$ that is not in $\gamma_{C\text{-Must} \times Block\text{-CS}}(\text{reduce}_{C\text{-Must} \times Block\text{-CS}}(\widehat{S}_1, \widehat{S}_2), \widehat{S}_2)$. Then, there must be an $i, 0 \leq i < n$, such that $|CS_i| \leq \widehat{S}_1(b_i)$ and $CS_i \subseteq \widehat{S}_2(b_i)$, but $|CS_i| \not\leq \min\{\widehat{S}_1(b_i), |\widehat{S}_2(b_i)|\}$. However, observe that $CS_i \subseteq \widehat{S}_2(b_i)$ implies that $|CS_i| \leq |\widehat{S}_2(b_i)|$, and so $|CS_i| \leq \min\{\widehat{S}_1(b_i), |\widehat{S}_2(b_i)|\}$, which yields a contradiction. ◀

We have seen previously that $C\text{-Must}$ and $Block\text{-CS}$ are incomparable in terms of precision. See Figure 5 for an example where the state reduction described above yields a more precise analysis result than is possible with any of the two analyses in isolation. In the example, v is persistent in a cache of associativity 3, which neither $C\text{-Must}$ nor $Block\text{-CS}$ are able to prove on their own. The example consists of a loop whose body contains two phases, each of which can only be handled precisely by one of the two domains. In the first phase of the loop body, containing the two accesses to w , $Block\text{-CS}$ is more precise, as it does not double count these two accesses in v 's conflict set. In the second phase of the loop body, $C\text{-Must}$ is more precise as it accounts for a single conflict due to the potential accesses to x and y , whereas $Block\text{-CS}$ accounts for two conflicts. State reduction enables $C\text{-Must}$ to profit from the more precise $Block\text{-CS}$ analysis in the phase of the loop body, reducing the bound for v to 2, as highlighted in the figure on the right. Due to this reduction, $C\text{-Must}$ is then able to show that v is indeed persistent in a cache of associativity 3.

5.2.4 State Reduction between $C\text{-Must}$ and $C\text{-May}$

Similarly to the information exchange between $C\text{-Must}$ and $Block\text{-CS}$, information can also be exchanged between $C\text{-Must}$ and $C\text{-May}$.

The idea of the reduction is the following: *C-May* and *C-Must* provide lower and upper bounds on the ages of memory blocks that have been accessed. A memory block b 's conflict set may only include another block c , if c 's lower bound is less than b 's upper bound. Thus b 's conflict set must be a subset of $\{b\} \cup \{c \in \mathcal{B} \mid c \neq b \wedge \widehat{S}_{C-May}(c) < \widehat{S}_{C-Must}(b)\}$ and so its size is bounded by $|\{c \in \mathcal{B} \mid c \neq b \wedge \widehat{S}_{C-May}(c) < \widehat{S}_{C-Must}(b)\}| + 1$.

Based on this insight, we introduce the following reduction operation:

$$\begin{aligned} \text{reduce}_{C-Must \times C-May} \left(\widehat{S}_{C-Must}, \widehat{S}_{C-May} \right) := \\ \lambda b \in \mathcal{B}. \min \left\{ \widehat{S}_{C-Must}(b), |\{c \in \mathcal{B} \mid c \neq b \wedge \widehat{S}_{C-May}(c) < \widehat{S}_{C-Must}(b)\}| + 1 \right\} \end{aligned} \quad (49)$$

► **Theorem 27** (Soundness of the State Reduction between *C-Must* and *C-May**). *The operator $\text{reduce}_{C-Must \times C-May}$ is a reduction operator for *C-Must* in the context of *C-May*.*

On the example program given in Figure 5 the state reduction between *C-Must* and *C-May* yields the same analysis result as the state reduction between *C-Must* and *Block-CS* given in the previous section.

5.2.5 Must Analysis

Recall the update of the *C-Must* analysis in (34). The bound on the size of a block's conflict set is increased by 1 upon *any* access to a different block (case 3 in (34)). At first sight it may seem that the update could be improved. It is tempting to take into account the size of the conflict set of the accessed block, as we did in case of *C-May*. Unfortunately, any such attempt would be incorrect: This is because the size bounds determined by *C-Must* are *conditional*, i.e., they only hold given that a block has been accessed at all. Given this definition of *C-Must*, it is always possible that an access is the very first access to the given block, which would contribute to the conflict sets of all other blocks, which have been accessed before, necessitating the update as it is.

In order to improve the update of *C-Must*, *unconditional* bounds on the sizes of blocks' conflict sets are required. Such unconditional bounds can be determined using the original LRU must analysis by Ferdinand and Wilhelm [12]. In this section, we recapitulate this must analysis, which we simply call *Must*. In order to use it in a cooperative update of *C-Must* in the context of *Must* in Section 5.2.6, we formalize the *Must* analysis as a persistence analysis. In particular, we provide a concretization function that expresses the set of cache traces represented by a *Must* analysis state. As a stand-alone persistence analysis, *Must* is not useful at all: no memory block can be classified as persistent by a stand-alone *Must* analysis. Its utility in persistence analysis stems from the fact that it may be used to improve the precision of *C-Must* via a cooperative update, which is provided in the following section.

Must analysis maintains an upper bound on the ages of memory blocks, where age bounds greater than k are collapsed to ∞ :

$$C_{Must}^{\#} := \mathcal{B} \rightarrow \{1, \dots, k, \infty\} \quad (50)$$

The original formalization of *Must* in [12] is not based on a *trace* collecting semantics. There, an abstract state corresponds to a set of concrete cache states. To fit into our persistence analysis framework, we here give an alternative formalization that captures the set of cache traces represented by an abstract cache trace⁹. The resulting concretization is quite similar to the one

⁹ We use the term *abstract trace* rather than *abstract state* as we interpret it to represent a set of cache traces.

for $C\text{-Must}$. The difference is that $C_{Must}^\#(b)$ must be ∞ for blocks that have not been accessed:

$$\begin{aligned} \gamma_{Must}(\widehat{S}) &:= \text{LRUCACHETRACES} \cap \\ &\{s = c_0\langle b_0, h_0 \rangle \dots c_n \mid (\forall b \in \mathcal{B} : (\forall i, 0 \leq i < n : b_i \neq b) \Rightarrow \widehat{S}(b) = \infty) \\ &\quad \wedge \forall i, 0 \leq i < n : b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)\}, \end{aligned} \quad (51)$$

where, as before, $CS_i(c_0\langle b_0, h_0 \rangle \dots c_n) := \{b_j \mid i \leq j < n\}$.

The concretization function given in the original formalization by Ferdinand and Wilhelm [12] is an abstraction of the one given above. It captures the final cache states of the cache traces determined by $\gamma_{Must}(\widehat{S})$, which is sufficient to classify memory accesses as guaranteed hits.

In the initial abstract cache trace, each block is assigned a bound of ∞ . By the concretization function this represents all possible cache traces, in particular those of length 0, i.e., traces consisting of an arbitrary initial cache state:

$$\widehat{T}_{Must} := \lambda b \in \mathcal{B}. \infty \quad (52)$$

As in $C\text{-Must}$ the maximum of the bounds is taken at joins:

$$\widehat{S} \sqsubseteq_{Must} \widehat{T} :\Leftrightarrow \forall b \in \mathcal{B} : \widehat{S}(b) \leq \widehat{T}(b) \quad \widehat{S} \sqcup_{Must} \widehat{T} := \lambda b \in \mathcal{B}. \max\{\widehat{S}(b), \widehat{T}(b)\} \quad (53)$$

Upon a memory access, the accessed block's bound is reduced to 1, as its conflict set will only contain the block itself (case 1, below). Other blocks' bounds are increased only if the accessed block's bound is greater than their bound (cases 2 and 3). This is sound because the bounds are unconditional in the must analysis.

$$\text{update}_{Must}^\#(\widehat{S}, b) := \lambda b'. \begin{cases} 1 & : b' = b \\ \widehat{S}(b') & : b' \neq b \wedge \widehat{S}(b) \leq \widehat{S}(b') \\ \widehat{S}(b') + 1 & : b' \neq b \wedge \widehat{S}(b) > \widehat{S}(b') \wedge \widehat{S}(b') < k \\ \infty & : b' \neq b \wedge \widehat{S}(b) > \widehat{S}(b') \wedge \widehat{S}(b') = k \end{cases} \quad (54)$$

For completeness, we provide the following classification function. A memory block is locally classified as persistent, if its bound is less than or equal to k , which implies that the block *must* be cached:

$$\text{classify}_{Must}^\#(\widehat{S}, b) := \widehat{S}(b) \leq k \quad (55)$$

As the bounds are initialized to ∞ , prior to the first access to a block, no block can be classified as persistent. This is why *Must* is not useful as a stand-alone persistence analysis.

► **Theorem 28** (Soundness of Must Analysis*). *Must is a sound persistence analysis.*

5.2.6 Cooperative Update for C-Must in the Context of Must

The unconditional bounds computed by *Must* can be used to improve the update of $C\text{-Must}$. A cooperative update for $C\text{-Must}$ in the context of *Must* is given below:

$$\text{coop-upd}_{C\text{-Must} \times \text{Must}}(\widehat{S}, \widehat{S}_{Must}, b) := \lambda b'. \begin{cases} 0 & : b' \neq b \wedge \widehat{S}(b') = 0 \\ 1 & : b' = b \\ \widehat{S}(b') & : b' \neq b \wedge \widehat{S}_{Must}(b) \leq \widehat{S}(b') \\ \widehat{S}(b') + 1 & : b' \neq b \wedge \widehat{S}_{Must}(b) > \widehat{S}(b') \wedge 0 < \widehat{S}(b') < k \\ \infty & : b' \neq b \wedge \widehat{S}_{Must}(b) > \widehat{S}(b') \wedge k \leq \widehat{S}(b') \end{cases} \quad (56)$$

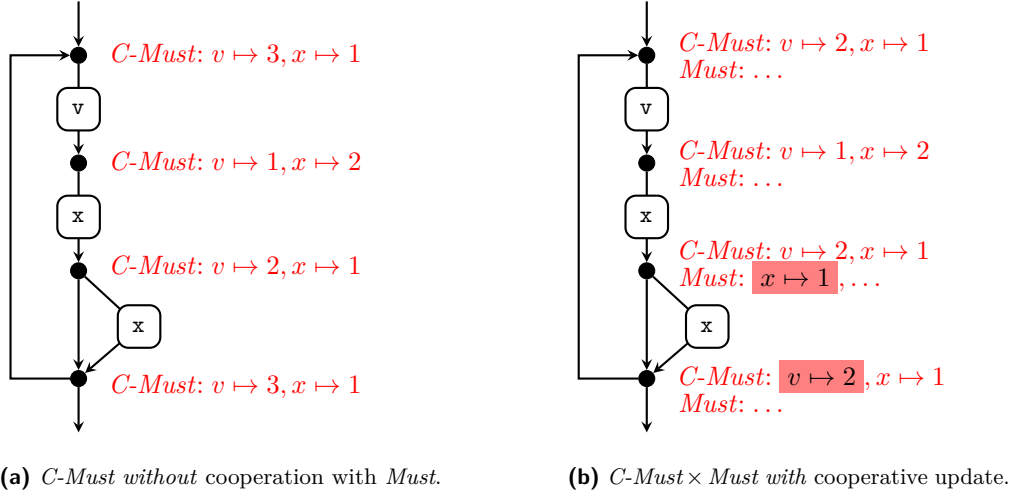


Figure 6 Example illustrating $C\text{-Must} \times Must$.

The update differs from the stand-alone update for $C\text{-Must}$ in the third case, where the bound for b' is not increased even though another block is accessed. The correctness of all other cases follows from the correctness of the original update for $C\text{-Must}$. Let's consider the third case more carefully. It occurs under the condition that $b' \neq b \wedge \hat{S}_{Must}(b) \leq \hat{S}(b')$. If $\hat{S}(b') = \infty$ then the update is trivially correct. So assume $\hat{S}(b') \leq k$ and thus also $\hat{S}_{Must}(b) \leq k$. In this case, the correctness of the update can be understood by the following case distinction:

1. Either b was actually contained in b' 's conflict set before the access. Then b' 's conflict set does not grow due to the access to b and keeping the previous bound on its size is correct.
2. Or b was not contained in b' 's conflict set before the access. Crucially, b must have been accessed before, as $\hat{S}_{Must}(b) \leq k$, and so b 's conflict set must contain b' 's conflict set. Further, b 's conflict set additionally contains b itself. As a consequence, b 's conflict set before the access contains b' 's conflict set after the access, and thus its bound, $\hat{S}_{Must}(b) \leq \hat{S}(b')$ is a correct bound for b' 's conflict set after the access.

A more formal and detailed correctness argument is given in the proof of the following theorem:

► **Theorem 29** (Soundness of Cooperative Update^{*}). *The function $coop\text{-}upd_{C\text{-Must} \times Must}$ is a cooperative update for $C\text{-Must}$ in the context of $Must$.*

Let's consider a small example illustrating the benefit of the cooperative update for $C\text{-Must}$ in the context of $Must$. Figures 6a and 6b show the analysis results of $C\text{-Must}$ with and without cooperation with $Must$ on a loop containing a conditional. Without cooperation, $C\text{-Must}$ is unable to prove that v is persistent in a cache of associativity 2. This is because, the two accesses to x both cause the bound on the size of v 's conflict set to increase by 1, even though only the first access to x may actually increase its size. In contrast, $C\text{-Must} \times Must$ with a cooperative update is able to prove that v is persistent in a cache of associativity 2, as illustrated in Figure 6b. Here, we only show the *relevant* information that $Must$ provides for the cooperative update: Right before the potential second access to x in the loop, $Must$ determines an *unconditional* bound on the size of x 's conflict set of 1. This information is then exploited in the cooperative update to determine that this second access may not increase the size of v 's conflict set, and so 2 is a correct bound on v 's conflict set at the end of the loop body.

5.3 Summary: The Landscape of Persistence Abstractions

In this section, we summarize the results obtained in Sections 5.1 and 5.2. In Section 5.1 we have seen abstractions following these two general approaches to persistence analysis:

1. Overapproximating the *set* of conflicting blocks
2. Overapproximating the *number* of conflicting blocks

Global-CS, *C-May*, and *Block-CS* all follow the first of these two approaches. Further, these three abstractions are totally ordered in terms of precision, with *Block-CS* strictly dominating *C-May*, and *C-May* strictly dominating *Global-CS*.

The only basic abstraction following the second approach is *C-Must*, which is incomparable to *Global-CS*, *C-May*, and *Block-CS*, i.e., there are cases where *C-Must* is more precise than *Block-CS*, but there are also cases where even *Global-CS* is more precise than *C-Must*.

In Section 5.2 we have then seen how to combine two incomparable basic abstractions into their so-called *direct product*, which is more precise than its constituents. To further increase analysis precision, we have also introduced two general mechanisms to exchange information between two abstractions:

- *state reduction* – where the analysis state of one abstraction is refined based on the analysis state of another abstraction, and
- *update reduction* – where the update of one abstraction takes into account information provided by another abstraction.

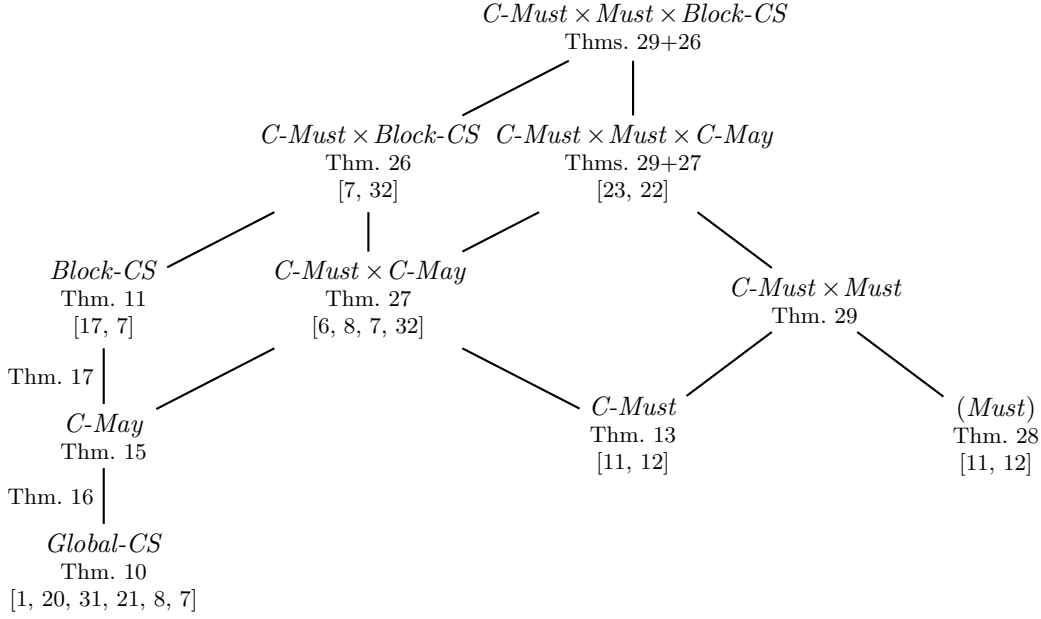
Then, we have seen three concrete instantiations of these mechanisms: state reductions between *C-Must* and *Block-CS* and between *C-Must* and *C-May*, as well as an update reduction between *C-Must* and a regular must cache analysis, which we simply call *Must* here. To facilitate the update reduction with *C-Must*, we have formalized *Must* as a persistence analysis, i.e., as an abstraction of sets of cache traces rather than sets of cache states.

We note that a state reduction between *C-Must* and *Global-CS* could be defined easily, similarly to the state reduction between *C-Must* and *Block-CS*. We do not provide this reduction here, because we believe that it would have little practical value: We have shown *C-May* to strictly dominate *Global-CS*, while it is hardly more expensive in terms of analysis time and memory consumption than *Global-CS*.

Figure 7 illustrates the landscape of persistence abstractions formalized in this article in the form of a Hasse diagram. Two comparable abstractions are connected by an edge, with the more precise abstraction drawn higher up in the diagram. Transitive relations are omitted for readability. Each abstraction is annotated with the corresponding theorem that shows its soundness. Similarly, relations between abstractions that are not the consequence of a product constructions are annotated with the corresponding dominance theorem. We also annotate abstractions with papers from the literature, which are based on the given abstraction. We discuss the related work in more detail in the following section.

6 Related Work and How It Maps Into the Landscape of Persistence Abstractions

In 1994, Mueller et al. [20, 1] introduced the “first miss” persistence notion and a corresponding persistence analysis for direct-mapped instruction caches. Later they extended their analysis to set-associative data [31] and instruction caches [21] with LRU replacement. The basic idea behind their analysis for set-associative caches is to collect all conflicting blocks in a given cache set within a loop. If all conflicting blocks fit into the cache together, then these blocks are classified as “first miss”. This corresponds to *Global-CS* applied separately to each loop in the program.



■ **Figure 7** Hasse diagram illustrating the relative precision of different persistence abstractions. The *Must* domain is in parentheses because it is not suitable to prove persistence of memory blocks on its own, but it may be useful in conjunction with other domains.

Ferdinand and Wilhelm [11, 12] introduced the “no eviction” persistence notion and a persistence analysis for set-associative caches. They characterized their analysis [12] as computing “the *maximal* position (relative age) for all memory blocks that *may* be in the cache.” Intuitively, their analysis thus corresponds to the *C-Must* analysis defined in this article. However, while their analysis employs the same join function as in the *C-Must* analysis, its update function differs; it is identical to the update function of the *Must* analysis: upon an access to memory block b only the ages of younger blocks are incremented. Unfortunately, this is unsound. To our knowledge, Hugues Cassé was the first to point this out. Given the concretization function of *C-Must* defined in this article, it is apparent why the update function is incorrect: *C-Must* bounds the age (the size of its conflict set) of a block only in case the block has previously been accessed. If the accessed block has not been accessed previously, it increases the ages of all other previously accessed blocks (the sizes of their conflict sets, respectively). Thus, without any additional information, upon an access to block b , a sound update function has to increase the bounds of all blocks, other than b , that have potentially been accessed before. The original *Must* analysis may provide such additional information, as it provides unconditional bounds on the maximum ages of blocks. The cooperative update for *C-Must* in the context of *Must* defined in Section 5.2.6 shows how to exploit this information for a more precise update of *C-Must*.

Aiming to solve the soundness issue of Ferdinand’s analysis, Cullmann [6] proposed an analysis combining Ferdinand’s persistence analysis with a slightly modified version of the *may* analysis from [12]. This combination corresponds to the product of *C-Must* and *C-May*, however with a less precise reduction than the one given in (49).

In a different approach to fix the soundness issues of the original persistence analysis, Huynh et al. [17] proposed a scope-aware persistence analysis for set-associative data caches. Their analysis tracks a *younger set* for each memory block, which corresponds to the *Block-CS* analysis in this article. To increase precision in the analysis of data caches, *temporal scopes* are used to distinguish different loop iterations in which array accesses in a loop touch different memory blocks.

■ **Listing 2** Input- and loop-iteration-dependent data accesses.

```
for (int i=0; i<N; i++) {
    k = read_sensor();
    sum[k] = sum[k] + arr[i];
}
```

Later, Cullmann [8] proposed “set-wise conflict counting”, which corresponds to *Global-CS* in this article and is similar to Mueller et al.’s approach. In his dissertation [7], Cullmann discusses two further analyses:

1. *Element-wise conflict counting*, which corresponds to the younger-set analysis by Huynh et al. [17] and *Block-CS* in this article.
2. *Age-tracking conflict counting*, which corresponds to the direct product of *C-Must* and *Block-CS* in this article, however, without the state reduction given in (48).

Nagar and Srikant [23, 22] show how to improve the precision of must, may, and persistence analysis by exchanging information between the analyses via what we call reductions in this article. Along the way they also identify and correct the soundness issue of Ferdinand’s persistence analysis. In case of persistence analysis, their approach corresponds to $C\text{-Must} \times \text{Must} \times C\text{-May}$ with the update and state reductions given in Theorems 29 and 27.

Similarly to Nagar and Srikant, Zhang and Koutsoukos [32] show how to combine *C-Must* and *C-May* using an update reduction. Their update improves upon Cullmann’s update in the combination of *C-Must* and *C-May*, which only uses the information from *C-May* to exclude the eviction of memory blocks from *C-Must*. Zhang and Koutsoukos also note that *Block-CS* is incomparable to $C\text{-Must} \times C\text{-May}$. They propose to combine $C\text{-Must} \times C\text{-May}$ and *Block-CS* in a single analysis to achieve higher precision than any of its constituents. As *Block-CS* dominates *C-May*, the resulting analysis is equivalent to $C\text{-Must} \times \text{Block-CS}$ in our framework (with the appropriate state reduction), and one might wonder why it could be useful to run the two analyses together with *C-May*. However, Zhang and Koutsoukos show how to derive “younger sets” from $C\text{-Must} \times C\text{-May}$. Whenever the derived younger set is equal to the one maintained by *Block-CS* it is not necessary to explicitly represent the younger set in *Block-CS*. In this way, the memory consumption of the analysis can be significantly reduced.

Ballabriga and Cassé [2] note that different blocks can be persistent within different scopes. For example, while an inner loop may entirely fit into the cache, its enclosing loop might not. In such a case, a sound persistence analysis would not be able to declare any of the loop’s memory blocks as persistent. Still, during any execution of the inner loop, each of its memory blocks may miss the cache at most once. Ballabriga and Cassé thus propose “multi-level” persistence analysis, which determines for each loop nesting level, whether blocks are persistent within the execution of the loop at that nesting level. This idea was later also applied to “temporal scopes” by Huynh et al. [17] as discussed earlier.

7 Extension to Data Caches

In the preceding sections we have focused on persistence analysis for instruction caches. Persistence analysis for data caches faces the additional challenge that an individual load or store instruction may result in different data memory accesses depending on the program’s inputs or the loop iteration the instruction is executed in. Consider the example in Listing 2. Within the loop, the access to the array `arr` depends on the loop iteration, and the accesses to the array `sum` depend on external sensor inputs.

It is possible to employ a control flow abstraction similar to the one described in Section 3.1 to such programs. However, due to input- and loop-iteration-dependent data accesses, some transitions in the control flow graph will have to be annotated with a set of possible memory blocks rather than an single one. The abstract trace update function then needs to be lifted to sets, which can be done in a generic manner as follows:

$$\text{update}_A^\#(\widehat{S}, B) := \bigsqcup_{b \in B} \text{update}_A^\#(\widehat{S}, b), \quad (57)$$

where B is a set of memory blocks.

In this way, all the persistence analyses discussed in this article can also be applied to the analysis of data caches. However, this generic approach comes with two drawbacks:

1. *Reduced efficiency*: Implementing (57) literally, the update and join functions need to be applied $|B|$ and $|B| - 1$ times, respectively. So if the set of potentially-accessed blocks B is large the analysis may become quite costly. However, in most cases, it is fairly straightforward to derive an expression for $\text{update}_A^\#(\widehat{S}, B)$ that does not involve applying the original update and join functions that often. For example, $\text{update}_{Global-CS}^\#(\widehat{S}, B) = \bigsqcup_{b \in B} \text{update}_{Global-CS}^\#(\widehat{S}, b)$ can be simplified to $\text{update}_{Global-CS}^\#(\widehat{S}, B) = \widehat{S} \cup B$. Nagar and Srikant [23, 22] describe such simplifications for their persistence analysis.
2. *Limited precision*: Due to uncertainty about the accessed memory blocks the analysis may be imprecise. In case of loop-iteration-dependent data accesses, more precise persistence classifications could be derived by performing the analysis on a more fine-grained abstraction of the program than its control flow abstraction. For instance, to increase analysis precision, Huynh et al. [17] introduce *temporal scopes* to distinguish different loop iterations in which array accesses in a loop touch different memory blocks.

8 Conclusions and Future Work

Our main goal has been to put persistence analysis on a more solid semantic foundation. We have argued that persistence is a property of cache traces rather than cache states. Accordingly, we introduced a trace-based semantics to formally capture varying persistence notions and to enable rigorous correctness proofs of persistence analyses.

Section 5 demonstrates that persistence analyses can be defined and proved correct as abstractions of a trace collecting semantics; we believe rather elegantly. Such formalizations also contribute to a better understanding of how and why an analysis works, simply by requiring its designer to precisely capture the meaning of the abstraction that the analysis is based upon. To our own surprise, it is possible to explain the essence of all prior persistence abstractions as combinations of just a few rather basic abstractions.

We note that our focus has been on the underlying abstractions and not on their efficient implementation. Different implementations of the same abstraction will deliver the same persistence classifications, but may exhibit different performance characteristics, in particular in terms of space consumption. This is, for example, demonstrated by Zhang and Koutsoukous [32], who show how to implement *Block-CS* more efficiently than a straightforward implementation that directly matches its logical definition.

In this article, we have only considered private single-level caches with LRU replacement. Future work should consider replacement policies other than LRU, which have received some attention in classifying cache analysis [26, 27, 25, 13, 14] and in the broader scope of quantitative cache analysis [16, 15], but which have so far received very little attention in persistence analysis. It may also be interesting to study persistence analysis for shared caches in multi cores. Such

shared caches are usually second- or third-level caches and thus any step in this direction would also require the analysis of multi-level caches. The lattice of abstractions in Figure 7 may be a good starting point for a rigorous experimental evaluation of the various persistence analyses that have been proposed to date. All abstractions studied in this article are sound but incomplete. It is conceivable to design a sound and complete persistence analysis along the lines of the recent work of Touzeau et al. [30].

Acknowledgments. I would like to sincerely thank the anonymous reviewers for their help in improving this paper.

References

- 1 Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS '94), San Juan, Puerto Rico, December 7-9, 1994*, pages 172–181. IEEE Computer Society, 1994. doi:10.1109/REAL.1994.342718.
- 2 Clément Ballabriga and Hugues Cassé. Improving the first-miss computation in set-associative instruction caches. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 341–350. IEEE Computer Society, 2008. doi:10.1109/ECRTS.2008.34.
- 3 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.
- 4 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.
- 5 Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In René Jacquart, editor, *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, volume 156 of *IFIP*, pages 359–366. Kluwer/Springer, 2004. doi:10.1007/978-1-4020-8157-6_27.
- 6 Christoph Cullmann. Cache persistence analysis: a novel approach theory and practice. In Jan Vitek and Bjorn De Sutter, editors, *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011*, pages 121–130. ACM, 2011. doi:10.1145/1967677.1967695.
- 7 Christoph Cullmann. *Cache persistence analysis for embedded real-time systems*. PhD thesis, Saarland University, Saarbrücken, Germany, 2013. URL: <http://d-nb.info/1052779867>.
- 8 Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embedded Comput. Syst.*, 12(1s):40:1–40:25, 2013. doi:10.1145/2435227.2435236.
- 9 B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.
- 10 Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Trans. Inf. Syst. Secur.*, 18(1):4:1–4:32, 2015. doi:10.1145/2756550.
- 11 Christian Ferdinand. *Cache behavior prediction for real-time systems*. PhD thesis, Saarland University, Saarbrücken, Germany, 1997. URL: <http://d-nb.info/953983706>.
- 12 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999. doi:10.1023/A:1008186323068.
- 13 Daniel Grund and Jan Reineke. Precise and efficient fifo-replacement analysis based on static phase detection. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*, pages 155–164. IEEE Computer Society, 2010. doi:10.1109/ECRTS.2010.8.
- 14 Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASICS*, pages 23–35. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. doi:10.4230/OASICS.WCET.2010.23.
- 15 Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU cache: Challenging LRU for predictability. *ACM Trans. Embedded Comput. Syst.*, 13(4s):123:1–123:26, 2014. doi:10.1145/2584655.
- 16 Nan Guan, Xinpeng Yang, Mingsong Lv, and Wang Yi. FIFO cache analysis for WCET estimation: a quantitative approach. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 296–301. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi:10.7873/DATE.2013.073.
- 17 Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*,

- RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011, pages 203–212. IEEE Computer Society, 2011. doi:10.1109/RTAS.2011.27.
- 18 Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973. doi:10.1145/512927.512945.
 - 19 Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. A survey on static cache analysis for real-time systems. *LITES*, 3(1):05:1–05:48, 2016. doi:10.4230/LITES-v003-i001-a005.
 - 20 Frank Mueller. *Static cache simulation and its applications*. PhD thesis, Florida State University, Tallahassee, United States, 1994. URL: http://www.cs.fsu.edu/~whalley/papers/mueller_diss94.pdf.
 - 21 Frank Müller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):217–247, 2000. doi:10.1023/A:1008145215849.
 - 22 Kartik Nagar. Cache analysis for multi-level data caches. Master’s thesis, Indian Institute of Science, Bangalore, India, 2012.
 - 23 Kartik Nagar and Y. N. Srikant. Interdependent cache analyses for better precision and safety. In *Tenth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEM-CODE 2012, Arlington, VA, USA, July 16-17, 2012*, pages 99–108. IEEE, 2012. doi:10.1109/MEMCOD.2012.6292306.
 - 24 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
 - 25 Jens Palsberg and Zhendong Su, editors. *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*. Springer, 2009. doi:10.1007/978-3-642-03237-0.
 - 26 Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008. URL: <http://rw4.cs.uni-saarland.de/~reineke/publications/DissertationCachesInWCETAnalysis.pdf>.
 - 27 Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. In Krisztián Flautner and John Regehr, editors, *Proceedings of the 2008 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’08), Tucson, AZ, USA, June 12-13, 2008*, pages 51–60. ACM, 2008. doi:10.1145/1375657.1375665.
 - 28 Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26, 2007. doi:10.1145/1275497.1275501.
 - 29 Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Compiler Design - Analysis and Transformation*. Springer, 2012. doi:10.1007/978-3-642-17548-0.
 - 30 Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Ascertaining uncertainty for efficient exact cache analysis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2017. doi:10.1007/978-3-319-63390-9_2.
 - 31 Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *3rd IEEE Real-Time Technology and Applications Symposium, RTAS ’97, Montreal, Canada, June 9-11, 1997*, pages 192–202. IEEE Computer Society, 1997. doi:10.1109/RTTAS.1997.601358.
 - 32 Zhenkai Zhang and Xenofon D. Koutsoukos. Improving the precision of abstract interpretation based cache persistence analysis. In Sam H. Noh, Sebastian Fischmeister, and Jason Xue, editors, *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES 2015, CD-ROM, Portland, OR, USA, June 18 - 19, 2015*, pages 10:1–10:10. ACM, 2015. doi:10.1145/2670529.2754967.

A Proofs

- **Definition 1** (Persistence in a Program). *Memory block b is persistent in program P , if*

$$\forall \tau \in \text{Col}(P) : \text{AtMostOneMiss}(\tau, b).$$

- **Definition 2** (Cache Trace Abstraction). *A cache trace abstraction is a tuple*

$$A = \left\langle C_A^\#, \gamma_A, \widehat{\mathcal{I}}_A, \sqsubseteq_A, \sqcup_A, \text{update}_A^\#, \text{classify}_A^\# \right\rangle,$$

consisting of the following components:

1. $C_A^\#$, a set of abstract traces,
2. $\gamma_A : C_A^\# \rightarrow 2^{\text{CacheTraces}}$, a concretization function, which specifies the set of concrete cache traces represented by each abstract trace,

3. $\widehat{\mathcal{I}}_A \in C_A^\#$, an abstract initial trace that represents all possible initial cache states,
4. \sqsubseteq_A , a partial order on $C_A^\#$, such that $\langle C_A^\#, \sqsubseteq_A \rangle$ is a complete lattice [9],
5. \sqcup_A , a join operator on abstract traces¹⁰,
6. $update_A^\# : C_A^\# \times \mathcal{B} \rightarrow C_A^\#$, an abstract update function,
7. $classify_A^\# : C_A^\# \times \mathcal{B} \rightarrow \mathbb{B}$, a persistence classification function.

In the proof of the following theorem, we will make use of Knaster-Tarski's fixpoint theorem. Many variants of Knaster-Tarski's fixpoint theorem can be found in the literature. Below, we reproduce one such variant and its proof from [9], adapted to the terminology used in this article:

► **Theorem 30 (Knaster-Tarski Fixpoint Theorem).** *Let (L, \leq) be a complete lattice and $F : L \rightarrow L$ a monotone function. Let $\bigwedge A$ denote the greatest lower bound of $A \subseteq L$. Then,*

$$\alpha := \bigwedge \{x \in L \mid F(x) \leq x\}$$

is a fixpoint of F . Further, α is the least fixpoint of F .

Proof. Let $H = \{x \in L \mid F(x) \leq x\}$. For all $x \in H$, we have $\alpha \leq x$, so $F(\alpha) \leq F(x) \leq x$. Thus $F(\alpha)$ is a lower bound of H , and so $F(\alpha) \leq \alpha$, as α is the greatest lower bound of H .

Since F is monotone, $F(F(\alpha)) \leq F(\alpha)$, and so $F(\alpha) \in H$, and thus $\alpha \leq F(\alpha)$. Thus we have established that α is a fixpoint of F .

If β is any fixpoint of F , then $\beta \in H$, and so $\alpha \leq \beta$. Thus α is the least fixpoint of F . ◀

► **Theorem 3 (Soundness of Persistence Analysis).** *If the cache trace abstraction A satisfies the following conditions:*

$$\mathcal{I}_C \subseteq \gamma_A(\widehat{\mathcal{I}}_A), \tag{6}$$

$$\forall \widehat{S}, \widehat{T} \in C_A^\# : \widehat{S} \sqsubseteq_A \widehat{T} \Rightarrow \gamma_A(\widehat{S}) \subseteq \gamma_A(\widehat{T}), \tag{7}$$

$$\begin{aligned} \forall \widehat{S} \in C_A^\#, b \in \mathcal{B} : \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\widehat{S}) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_A(\text{update}_A^\#(\widehat{S}, b)). \end{aligned} \tag{8}$$

Then, its abstract semantics soundly approximates its concrete counterpart:

$$\text{StickyCol}(P_{ins}) \leq \gamma_A(\widehat{\text{StickyCol}}_A(P_{ins})), \tag{9}$$

where γ_A is lifted to functions as follows: $\gamma_A(\widehat{S}) = \lambda l \in \mathcal{L}. \gamma_A(\widehat{S}(l))$.

Proof. We first show that (8) implies the “local consistency” of $next_{ins, A}^\#$ relative to $next_{ins}$, i.e., $next_{ins}(\gamma_A(\widehat{S})) \leq \gamma_A(next_{ins, A}^\#(\widehat{S}))$.

Choose an arbitrary $l' \in \mathcal{L}$. Then:

$$\begin{aligned} next_{ins}(\gamma_A(\widehat{S}))(l') &\stackrel{\text{Def.}}{=} \bigcup_{(l, l') \in E} \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\widehat{S})(l) \\ &\quad \wedge b = \text{eff}_C(l, l') \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ &\stackrel{(8)}{\subseteq} \bigcup_{(l, l') \in E} \gamma_A(\text{update}_A^\#(\widehat{S}(l), \text{eff}_C(l, l'))) \\ &\stackrel{(7)}{\subseteq} \gamma_A(\bigsqcup_{(l, l') \in E} \text{update}_A^\#(\widehat{S}(l), \text{eff}_C(l, l'))) \\ &\stackrel{\text{Def.}}{=} \gamma_A(next_{ins, A}^\#(\widehat{S})(l')) \end{aligned}$$

¹⁰Note that in a complete lattice (L, \sqsubseteq) the partial order \sqsubseteq uniquely defines the join operator \sqcup . Vice versa, a given join operator uniquely defines a corresponding partial order. Nevertheless, we explicitly provide both partial order and join operator here and in the following.

From this it follows that $\gamma_A(\widehat{StickyCol}_A(P_{ins}))$ is a post fixpoint of $next_{ins}$:

$$\begin{aligned}
\gamma_A(\widehat{StickyCol}_A(P_{ins})) & \stackrel{\text{fixpoint}}{=} \gamma_A(\widehat{Init}_A \sqcup_A next_{ins,A}^\#(\widehat{StickyCol}_A(P_{ins}))) \\
& \stackrel{(7)}{\geq} \gamma_A(\widehat{Init}) \vee \gamma_A(next_{ins,A}^\#(\widehat{StickyCol}_A(P_{ins}))) \\
& \stackrel{\text{"local consistency"}}{\geq} \gamma_A(\widehat{Init}) \vee next_{ins}(\gamma_A(\widehat{StickyCol}_A(P_{ins}))) \\
& \stackrel{(6)}{\geq} Init \vee next_{ins}(\gamma_A(\widehat{StickyCol}_A(P_{ins})))
\end{aligned}$$

In order to apply Knaster-Tarski's fixpoint theorem, we need to show that the domain of the sticky cache trace collecting semantics $(\mathcal{L} \rightarrow 2^{CacheTraces}, \leq)$ is a complete lattice, and that $next_{ins}$ is a monotone function. The power set 2^A of any set A is a complete lattice with respect to the subset relation \subseteq . Thus $(2^{CacheTraces}, \subseteq)$ is a complete lattice. Also, the total function space $A \rightarrow B$ between a set A and a complete lattice (B, \leq) is a complete lattice w.r.t. to the pointwise ordering $f \leq g \Leftrightarrow \forall a \in A : f(a) \leq g(a)$. Thus $\mathcal{L} \rightarrow 2^{CacheTraces}$ is a complete lattice w.r.t. to \leq , as it is defined in Section 3.2.

To see that $next_{ins}$ is monotone w.r.t. \leq , first observe that the lifting $F(X) := \{f(x) \mid x \in X\}$ of any function f to sets is a monotone function w.r.t. \subseteq , i.e., if $X \subseteq Y$, then also $F(X) \subseteq F(Y)$. Thus $F(l, l') := \{t.c(b, h)c' \mid t.c \in X \wedge b = eff_{\mathcal{L}}(l, l') \wedge h = eff_C(c, b) \wedge c' = update_C(c, b)\}$ is monotone w.r.t. \subseteq for any l, l' . Also, the union of multiple monotone functions w.r.t. \subseteq is monotone w.r.t. \subseteq . Finally, the pointwise application of monotone functions w.r.t. \leq is monotone w.r.t. its pointwise extension \leq , as defined in Section 3.2, and so $next_{ins}$ is monotone. Further, any constant function is monotone w.r.t. to any order. Thus, the pointwise union of the constant function $Init$ and $next_{ins}$ is monotone as well.

Applying Knaster-Tarski's fixpoint theorem to the complete lattice $(\mathcal{L} \rightarrow 2^{CacheTraces}, \leq)$ and the monotone function $Init \vee next_{ins}$, we get that its post fixpoint $\gamma_A(\widehat{StickyCol}_A(P_{ins}))$ is greater than or equal to its least fixpoint

$$\begin{aligned}
StickyCol(P_{ins}) & \stackrel{\text{Def.}}{=} lfp_{Init}^{\leq} next_{ins} \\
& = lfp^{\leq}(Init \vee next_{ins}) \\
& \stackrel{\text{Knaster-Tarski}}{=} \bigwedge \{x \mid x \geq Init \vee next_{ins}(x)\}. \quad \blacktriangleleft
\end{aligned}$$

► **Theorem 4 (Soundness of Persistence Classification).** *If the cache trace abstraction A satisfies conditions (6), (7), (8) from Theorem 3, and $classify_A^\#$ satisfies*

$$\begin{aligned}
\forall \widehat{S} \in C_A^\#, b \in \mathcal{B} : classify_A^\#(\widehat{S}, b) \Rightarrow \\
\forall c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \gamma_A(\widehat{S}) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b), \quad (10)
\end{aligned}$$

then $classify_A^\#(P_{ins}, b) := \forall l \in \mathcal{L} : classify_A^\#(\widehat{StickyCol}_A(P_{ins})(l), b)$ implies the persistence of memory block b in program P_{ins} .

Proof. Proof by contradiction. Assume that $\forall l \in \mathcal{L} : classify_A^\#(\widehat{StickyCol}_A(P_{ins})(l), b)$ holds for some memory block b , but b is not persistent in P_{ins} according to Definition 1. Then, there must be a trace $\tau = \langle l_0, c_0 \rangle e_0 \langle l_1, c_1 \rangle e_1 \dots e_{n-1} \langle l_n, c_n \rangle \in Col(P_{ins})$, such that $AtMostOneMiss(\tau, b)$ does not hold. Let i and j be such that $e_i = e_j = \langle b, miss \rangle$ and $i < j$.

By conditions (6), (7), and (8), Theorem 3 holds, and so

$$Col(P_{ins}) \subseteq \gamma_{ins}(StickyCol(P_{ins})) \subseteq \gamma_{ins}(\gamma_A(\widehat{StickyCol}_A(P_{ins}))).$$

So $\tau \in \gamma_{ins}(\gamma_A(\widehat{StickyCol}_A(P_{ins})))$. By (4), this implies that

$$c_0 e_0 \dots c_j \in \gamma_A(\widehat{StickyCol}_A(P_{ins}))(l_j) = \gamma_A(\widehat{StickyCol}_A(P_{ins}))(l_j).$$

By assumption $\text{classify}_A^\#(\widehat{\text{StickyCol}}_A(P_{ins})(l_j), b)$ holds, and so due to (10), we have

$$b \in c_j \vee (\forall i, 0 \leq i < j : b_i \neq b).$$

As $e_i = \langle b, \text{miss} \rangle$, we can exclude the second part of the disjunction. However, $b \in c_j$ contradicts $e_j = \langle b, \text{miss} \rangle$, which concludes the proof. \blacktriangleleft

► **Definition 5** (Precision). *Given two cache trace abstractions A and B , we say that A is at least as precise as B , denoted by $A \succeq B$, if A classifies each block as persistent that B classifies as persistent:*

$$\forall P_{ins}, \forall b : \text{classify}_B^\#(P_{ins}, b) \Rightarrow \text{classify}_A^\#(P_{ins}, b).$$

We say that A is more precise than B , denoted by $A \succ B$, if $A \succeq B$, but $B \not\succeq A$. If neither $A \succeq B$ nor vice versa, we say that A and B are incomparable.

► **Theorem 6** (Approximation of Abstract Semantics). *Given two cache trace abstractions A and B , and a function $\gamma_{B \rightarrow A} : C_B^\# \rightarrow C_A^\#$ that satisfies the following conditions:*

$$\widehat{\mathcal{I}}_A \subseteq \gamma_{B \rightarrow A}(\widehat{\mathcal{I}}_B), \quad (11)$$

$$\forall \widehat{S}, \widehat{T} \in C_B^\# : \widehat{S} \sqsubseteq_B \widehat{T} \Rightarrow \gamma_{B \rightarrow A}(\widehat{S}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{T}), \quad (12)$$

$$\forall \widehat{S} \in C_B^\#, b \in \mathcal{B} : \text{update}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}), b) \sqsubseteq_A \gamma_{B \rightarrow A}(\text{update}_B^\#(\widehat{S}, b)). \quad (13)$$

Then, B 's abstract semantics soundly approximates its more concrete counterpart:

$$\widehat{\text{StickyCol}}_A(P_{ins}) \sqsubseteq_A \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})), \quad (14)$$

where $\gamma_{B \rightarrow A}$ is lifted to the abstract sticky trace collecting semantics as follows:

$$\gamma_{B \rightarrow A}(\widehat{S}) = \lambda l \in \mathcal{L}. \gamma_{B \rightarrow A}(\widehat{S}(l)).$$

Proof. We first show that (13) implies the ‘‘local consistency’’ of $\text{next}_{ins,B}^\#$ relative $\text{next}_{ins,A}^\#$, i.e., $\text{next}_{ins,A}^\#(\gamma_{B \rightarrow A}(\widehat{S})) \sqsubseteq_A \gamma_{B \rightarrow A}(\text{next}_{ins,B}^\#(\widehat{S}))$.

Choose an arbitrary $l' \in \mathcal{L}$. Then:

$$\begin{aligned} \text{next}_{ins,A}^\#(\gamma_{B \rightarrow A}(\widehat{S}))(l') &\stackrel{\text{Def.}}{=} \bigsqcup_{(l,l') \in E} \text{update}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}(l)), \text{eff}_{\mathcal{L}}(l, l')) \\ &\stackrel{(13)}{\sqsubseteq_A} \bigsqcup_{(l,l') \in E} \gamma_{B \rightarrow A}(\text{update}_B^\#(\widehat{S}(l), \text{eff}_{\mathcal{L}}(l, l'))) \\ &\stackrel{(12)}{\sqsubseteq_A} \gamma_{B \rightarrow A}(\bigsqcup_{(l,l') \in E} \text{update}_B^\#(\widehat{S}(l), \text{eff}_{\mathcal{L}}(l, l'))) \\ &\stackrel{\text{Def.}}{=} \gamma_{B \rightarrow A}(\text{next}_{ins,B}^\#(\widehat{S})(l')) \end{aligned}$$

From this it follows that $\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))$ is a post fixpoint of $\text{next}_{ins,A}^\#$:

$$\begin{aligned} \gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins})) &\stackrel{\text{fixpoint}}{=} \gamma_{B \rightarrow A}(\widehat{\text{Init}}_B \sqcup_B \text{next}_{ins,B}^\#(\widehat{\text{StickyCol}}_B(P_{ins}))) \\ &\stackrel{(12)}{\sqsupseteq_A} \gamma_{B \rightarrow A}(\widehat{\text{Init}}_B) \sqcup_A \gamma_{B \rightarrow A}(\text{next}_{ins,B}^\#(\widehat{\text{StickyCol}}_B(P_{ins}))) \\ &\stackrel{\text{‘‘local consistency’’}}{\sqsupseteq_A} \gamma_{B \rightarrow A}(\widehat{\text{Init}}_B) \sqcup_A \text{next}_{ins,A}^\#(\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))) \\ &\stackrel{(11)}{\sqsupseteq_A} \widehat{\text{Init}}_A \sqcup_A \text{next}_{ins}(\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))) \end{aligned}$$

In order to apply Knaster-Tarski's fixpoint theorem, we need to show that the domain of abstraction A , $(\mathcal{L} \rightarrow C_A^\#, \sqsubseteq_A)$ is a complete lattice, and that $\text{next}_{ins,A}^\#$ is a monotone function. The total function space $A \rightarrow B$ between a set A and a complete lattice (B, \leq) is a complete lattice w.r.t.

to the pointwise ordering $f \leq g := \forall a \in A : f(a) \leq g(a)$. Thus $\mathcal{L} \rightarrow C_A^\#$ is a complete lattice w.r.t. to \sqsubseteq_A , as it is defined in Section 4.1.

To see that $\text{next}_{ins,A}^\#$ is monotone w.r.t. \sqsubseteq_A , observe that by assumption $\text{update}_A^\#$ is monotone in its first parameter. Thus, $F_{l,l'}(X) := \text{update}_A^\#(\widehat{S}(l), \text{eff}_{\mathcal{L}}(l, l'))$ is monotone in \widehat{S} for any l, l' . Also, the least upper bound of multiple monotone functions is a monotone function, and so $F(l') := \bigsqcup_{(l,l') \in E} \{\text{update}_A^\#(\widehat{S}(l), b) \mid b = \text{eff}_{\mathcal{L}}(l, l')\}$ is monotone in \widehat{S} . Finally, the pointwise application of monotone functions w.r.t. \leq is monotone w.r.t. its pointwise extension \leq , and so $\text{next}_{ins,A}^\# = \lambda l' \in \mathcal{L}.F(l')$ is monotone. Further, any constant function is monotone w.r.t. to any order. Thus, the pointwise union of the constant function $\widehat{\text{Init}}_A$ and $\text{next}_{ins,A}^\#$ is monotone as well.

Applying Knaster-Tarski's fixpoint theorem to the complete lattice $(\mathcal{L} \rightarrow C_A^\#, \sqsubseteq_A)$ and the monotone function $\widehat{\text{Init}}_A \sqcup_A \text{next}_{ins,A}^\#$, we get that its post fixpoint $\gamma_{B \rightarrow A}(\widehat{\text{StickyCol}}_B(P_{ins}))$ is greater than or equal to its least fixpoint:

$$\begin{aligned} \widehat{\text{StickyCol}}_A(P_{ins}) &\stackrel{\text{Def.}}{=} \text{lfp}_{\widehat{\text{Init}}_A}^{\sqsubseteq_A} \text{next}_{ins,A}^\# \\ &= \text{lfp}^{\sqsubseteq_A} \widehat{\text{Init}}_A \sqcup_A \text{next}_{ins,A}^\# \\ &\stackrel{\text{Knaster-Tarski}}{=} \bigsqcap_A \{x \mid x \sqsubseteq_A \widehat{\text{Init}}_A \sqcup_A \text{next}_{ins,A}^\#(x)\}. \quad \blacktriangleleft \end{aligned}$$

Whenever the proof of a theorem is in the main part of the article, the name of the theorem is marked with a \star and serves as a link to the corresponding proof. The first example of such a case is the following theorem:

► **Theorem 7 (Precision \star)**. *Given cache trace abstractions A, B and a function $\gamma_{B \rightarrow A}$ that satisfies conditions (11), (12), and (13) from Theorem 6, and further*

$$\forall \widehat{S} \in C_B^\#, b \in \mathcal{B} : \text{classify}_B^\#(\widehat{S}, b) \Rightarrow \text{classify}_A^\#(\gamma_{B \rightarrow A}(\widehat{S}), b), \quad (15)$$

$$\forall \widehat{S}, \widehat{T} \in C_A^\#, b \in \mathcal{B} : \widehat{S} \sqsubseteq_A \widehat{T} \Rightarrow (\text{classify}_A^\#(\widehat{T}, b) \Rightarrow \text{classify}_A^\#(\widehat{S}, b)). \quad (16)$$

Then, A is at least as precise as B , i.e., $A \succeq B$.

► **Theorem 8 (Soundness of Persistence Classification \star)**. *Given two cache trace abstractions A and B . If A is sound, and A is at least as precise as B , then B is also sound.*

► **Lemma 31 (Monotonicity of LRU)**. *Consider an arbitrary cache trace $c_0(b_0, h_0)c_1(b_1, h_1) \dots c_n \in \text{LRUCACHETRACES}$. Assume that $c_0(b) > c_0(b')$ and $b \notin \{b_0, b_1, \dots, b_{n-1}\}$. Then:*

$$\forall i, 0 \leq i \leq n : c_i(b) > c_i(b').$$

Proof. Proof by induction over i :

■ Base case ($i = 0$):

$c_0(b) > c_0(b')$ holds by assumption.

■ Inductive step:

We must show that $c_{i+1}(b) > c_{i+1}(b')$.

By the inductive hypothesis (I.H.) we have $c_i(b) > c_i(b')$.

We distinguish two cases:

1. $c_i(b) > c_i(b') + 1$:

By the definition of $\text{update}_{\mathcal{C}}^{LRU}$ we have $c_i(b') + 1 \geq c_{i+1}(b')$.

As by assumption $b \neq b_i$, it also follows from the definition of $\text{update}_{\mathcal{C}}^{LRU}$ that $c_{i+1}(b) \geq c_i(b)$.

Thus, $c_{i+1}(b) \geq c_i(b) > c_i(b') + 1 \geq c_{i+1}(b')$.

2. $c_i(b) = c_i(b') + 1$:

We distinguish four cases based on the value of $c_i(b_i)$:

– $c_i(b_i) < c_i(b')$:

By the definition of $update_C^{LRU}$ (third case), $c_{i+1}(b') = c_i(b')$ and $c_{i+1}(b) = c_i(b)$.

And so $c_{i+1}(b) = c_i(b) \stackrel{\text{I.H.}}{>} c_i(b') = c_{i+1}(b')$.

– $c_i(b_i) = c_i(b')$:

This implies that $b_i = b'$. Thus, by the definition of $update_C^{LRU}$, $c_{i+1}(b) \stackrel{\text{first case}}{=} 0 < c_i(b') + 1 = c_{i+1}(b) \stackrel{\text{third case}}{=} c_i(b)$.

– $c_i(b_i) = c_i(b)$:

This implies that $b_i = b$, which contradicts our assumption that $b \notin \{b_0, b_1, \dots, b_{n-1}\}$.

– $c_i(b_i) > c_i(b')$:

By the definition of $update_C^{LRU}$, $c_{i+1}(b') \stackrel{\text{second case}}{=} c_i(b') + 1$ and $c_{i+1}(b) \stackrel{\text{second case}}{=} c_i(b) + 1$.

And so $c_{i+1}(b) = c_i(b) + 1 \stackrel{\text{I.H.}}{>} c_i(b') + 1 = c_{i+1}(b')$. ◀

► **Lemma 9** (Persistence under LRU). *Consider an arbitrary cache trace $c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \text{LRUCACHETRACES}$. Then $c_n(b_0) < k$, if $|\{b_i \mid 0 \leq i < n\}| \leq k$.*

Proof. Let $s = c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \text{LRUCACHETRACES}$ be an arbitrary cache trace and assume that $B = \{b_i \mid 0 \leq i < n\}$ with $|B| \leq k$. We need to show that $c_n(b_0) < k$.

Let j be the index of the last occurrence of b_0 in s , i.e., $b_j = b_0$ and $\forall l > j : b_l \neq b_0$. Observe that $c_{j+1}(b_0) = 0$, because of the preceding access to $b_0 = b_j$. Let $B_j = \{b_i \mid j < i < n\}$. By construction, $b_0 \notin B_j$. As $b_0 \in B$ and $B_j \subseteq B$, we have $|B_j| < |B| \leq k$ and thus $|B_j| < k$.

Each memory block in B_j occurs one or more times in the suffix $s_j = c_{j+1} \langle b_{j+1}, h_{j+1} \rangle \dots c_n$. Let I be the set of indices of the first occurrences of the blocks in B_j in s_j , i.e.,

$$I = \{i \mid j < i < n \wedge \forall l, j < l < i : b_l \neq b_i\}.$$

Let I^C be the complement of I , i.e., $I^C = \{j + 1, \dots, n - 1\} \setminus I$. We claim that

1. $c_{i+1}(b_0) \leq c_i(b_0) + 1$ for all $i \in I$, and
2. $c_{t+1}(b_0) = c_t(b_0)$ for all $t \in I^C$.

These two claims imply that $c_n(b_0) \leq c_{j+1}(b_0) + |I| = |I|$. As $|I| = |B_j| < k$, $c_n(b_0) = |I| < k$, and it only remains to show the two claims:

1. The fact that $c_{i+1}(b_0) \leq c_i(b_0) + 1$ follows immediately from the definition of $update_C^{LRU}$.
2. Let t be an arbitrary index in I^C and let v be the greatest index smaller than t such that $b_v = b_t$. As $t \in I^C$ there must be such a $v > j$ due to the definitions of I and I^C .

Observe that $c_{v+1}(b_t) = c_{v+1}(b_v) = 0$ and $c_{v+1}(b_0) > 0$. Applying Lemma 31 to the the subsequence $c_{v+1} \langle b_{v+1}, h_{v+1} \rangle \dots c_t$ with $b = b_0$ and $b' = b_t$ yields that $c_t(b_0) > c_t(b_t)$.

As $c_t(b_0) > c_t(b_t)$, the third case in $update_C^{LRU}$ applies and we get $c_{t+1}(b_0) = c_t(b_0)$. ◀

► **Theorem 10** (Soundness of Global May-Conflict Set). *Global-CS is a sound persistence analysis.*

Proof. We show that *Global-CS* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

- Equation (6) is trivially satisfied, because sequences consisting only of the initial state c_0 are unconstrained in the definition of $\gamma_{\text{Global-CS}}$.
- Let $\widehat{S} \sqsubseteq_{\text{Global-CS}} \widehat{T}$. Let $s = c_0 \langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{\text{Global-CS}}(\widehat{S})$. Then $\{b_i \mid 0 \leq i < n\} \subseteq \widehat{S} \subseteq \widehat{T}$ and so by definition of $\gamma_{\text{Global-CS}}$, $s \in \gamma_{\text{Global-CS}}(\widehat{T})$, which shows that (7) is satisfied.

- Let $\widehat{S} \in C_{Global-CS}^\#$, $b \in \mathcal{B}$, and $t.c \in \gamma_{Global-CS}(\widehat{S})$ be arbitrary.
 To show that (8) is satisfied, we have to show that $t.c\langle b, h \rangle c'$ with $h = \text{eff}_C^{LRU}(c, b)$ and $c' = \text{update}_C^{LRU}(c, b)$ is an element of $\gamma_{Global-CS}(\text{update}_{Global-CS}^\#(\widehat{S}, b))$.
 By definition of $\gamma_{Global-CS}$ and $\text{update}_{Global-CS}^\#$ we have

$$\begin{aligned} & \gamma_{Global-CS}(\text{update}_{Global-CS}^\#(\widehat{S}, b)) \\ & \stackrel{\text{Def. } \text{update}_{Global-CS}^\#}{=} \gamma_{Global-CS}(\widehat{S} \cup \{b\}) \\ & \stackrel{\text{Def. } \gamma_{Global-CS}}{=} \text{LRUCACHETRACES} \cap \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \widehat{S} \cup \{b\}\} \end{aligned}$$
 Because $t.c \in \gamma_{Global-CS}(\widehat{S})$, we have that $t.c \in \text{LRUCACHETRACES}$. From $h = \text{eff}_C^{LRU}(c, b)$ and $c' = \text{update}_C^{LRU}(c, b)$, it follows that $t.c\langle b, h \rangle c' \in \text{LRUCACHETRACES}$.
 It remains to show that $t.c\langle b, h \rangle c' \in \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \widehat{S} \cup \{b\}\}$.
 As $t.c \in \gamma_{Global-CS}(\widehat{S})$, we have that $t.c \in \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \widehat{S}\}$. So $t.c\langle b, h \rangle c' \in \{c_0\langle b_0, h_0 \rangle c_1 \dots c_n \mid \{b_i \mid 0 \leq i < n\} \subseteq \widehat{S} \cup \{b\}\}$.
- Let $\widehat{S} \in C_{Global-CS}^\#$ and $b \in \mathcal{B}$ be arbitrary.
 To show that (10) is satisfied, we consider two cases: 1. $b \notin \widehat{S}$ and 2. $b \in \widehat{S}$.
Case 1: If $c_0\langle b_0, h_0 \rangle c_1 \dots c_n \in \gamma_{Global-CS}(\widehat{S})$, then $\{b_i \mid 0 \leq i < n\} \subseteq \widehat{S}$ by the definition of $\gamma_{Global-CS}$. As $b \notin \widehat{S}$, the second disjunct in (10) holds: $\forall i, 0 \leq i < n : b_i \neq b$.
Case 2: Let $c_0\langle b_0, h_0 \rangle c_1 \dots c_n \in \gamma_{Global-CS}(\widehat{S})$. Assume $b_i = b$ for some i . Otherwise the second disjunct of (10) holds. As $c_0\langle b_0, h_0 \rangle c_1 \dots c_n \in \gamma_{Global-CS}(\widehat{S})$, in particular $\{b_j \mid i \leq j < n\} \subseteq \widehat{S}$. As $|\widehat{S}| \leq k$ and $c_0\langle b_0, h_0 \rangle c_1 \dots c_n \in \text{LRUCACHETRACES}$, we can apply Lemma 9 to the trace $c_i\langle b_i, h_i \rangle c_{i+1} \dots c_n$ to conclude that $b \in c_n$. ◀

► **Theorem 11** (Soundness of Block-wise May-Conflict Set). *Block-CS is a sound persistence analysis.*

Proof. We show that *Block-CS* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

- Equation (6) is trivially satisfied, because sequences consisting only of the initial state c_0 are unconstrained in the definition of $\gamma_{Block-CS}$.
- Let $\widehat{S} \sqsubseteq_{Block-CS} \widehat{T}$. Let $s = c_0\langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{Block-CS}(\widehat{S})$. Because $\widehat{S}(b_i) \subseteq \widehat{T}(b_i)$ for all i , and \subseteq is transitive, s is also an element of $\gamma_{Block-CS}(\widehat{T})$, which shows that (7) is satisfied.
- Let $\widehat{S} \in C_{Block-CS}^\#$, $b \in \mathcal{B}$, and $s = c_0\langle b_0, h_0 \rangle \dots c_n \in \gamma_{Block-CS}(\widehat{S})$ be arbitrary.
 To show that (8) is satisfied, we have to show that $t = c_0\langle b_0, h_0 \rangle \dots c_n\langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{Block-CS}(\widehat{T})$, with $\widehat{T} = \text{update}_{Block-CS}^\#(\widehat{S}, b_n)$.
 Because $s \in \gamma_{Block-CS}(\widehat{S})$, we have that $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s.\langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .
 It remains to show that the second constraint in (25) holds¹¹, i.e.,

$$\forall i, 0 \leq i < n + 1 : b_i \in CS_{i+1}(t) \vee CS_i(t) \subseteq \widehat{T}(b_i), \quad (58)$$

where $CS_i(c_0\langle b_0, h_0 \rangle \dots c_{n+1}) := \{b_j \mid i \leq j < n + 1\}$.

In order to show that (58) holds, we distinguish two cases based on the value of i :

¹¹The constraint below accounts for the fact that t contains $n + 1$ accesses, where n is the number of accesses in s .

1. $i = n$:

Observe that $CS_n(t) = \{b_n\}$.

Due to the second case in the definition of $update_{Block-CS}^\#$, $\widehat{T}(b_n) = \{b_n\}$, and so

$$CS_n(t) = \{b_n\} \subseteq \{b_n\} = \widehat{T}(b_n).$$

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee CS_i(s) \subseteq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{Block-CS}(\widehat{S})$.

We distinguish two cases:

a. $b_i \in CS_{i+1}(s)$:

As $CS_{i+1}(s) = CS_{i+1}(t) \cup \{b_n\}$, the fact that $b_i \in CS_{i+1}(s)$ implies $b_i \in CS_{i+1}(t)$.

b. $b_i \notin CS_{i+1}(s)$ and thus $CS_i(s) \subseteq \widehat{S}(b_i)$:

We distinguish two cases:

i. $b_i \neq b_n$:

Then $CS_i(t) = CS_i(s) \cup \{b_n\} \subseteq \widehat{S}(b_i) \cup \{b_n\} = \widehat{T}(b_i)$ as the third case in $update_{Block-CS}^\#$ applies:

$CS_i(s) \neq \emptyset$ and thus $\widehat{S}(b_i) \neq \emptyset$ and $b_i \neq b_n$.

ii. $b_i = b_n$:

Then $b_i \in CS_{i+1}(t) = CS_{i+1}(s) \cup \{b_i\}$.

■ Let $\widehat{S} \in C_{Block-CS}^\#$ and $b \in \mathcal{B}$ be arbitrary.

To show that (10) is satisfied, assume $classify_{Block-CS}^\#(\widehat{S}, b)$ holds and thus $|\widehat{S}(b)| \leq k$. Let $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ be an arbitrary cache trace in $\gamma_{Block-CS}(\widehat{S})$. Let b_i be the last occurrence of b in the trace. If b does not occur in the trace, then (10) holds by the second disjunct. Otherwise, $b_i \notin CS_{i+1}(s)$ and so $CS_i(s) \subseteq \widehat{S}(b)$. As $|\widehat{S}(b)| \leq k$ and $s \in \text{LRUCACHETRACES}$, we can apply Lemma 9 to the suffix $c_i \langle b_i, h_i \rangle \dots c_n$ to prove that $b \in c_n$. ◀

► **Theorem 12** (*Block-CS vs. Global-CS**). *Block-CS is more precise than Global-CS.*

► **Theorem 13** (*Soundness of Conditional Must*). *C-Must is a sound persistence analysis.*

Proof. We show that *C-Must* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

■ Equation (6) is trivially satisfied, because sequences consisting only of the initial state c_0 only are unconstrained in the definition of γ_{C-Must} .

■ Let $\widehat{S} \sqsubseteq_{C-Must} \widehat{T}$. Let $s = c_0 \langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{C-Must}(\widehat{S})$. Because $\widehat{S}(b_i) \leq \widehat{T}(b_i)$ for all i , and \leq is transitive, s is also an element of $\gamma_{C-Must}(\widehat{T})$, which shows that (7) is satisfied.

■ Let $\widehat{S} \in C_{C-Must}^\#$, $b \in \mathcal{B}$, and $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-Must}(\widehat{S})$ be arbitrary.

To show that (8) is satisfied, we have to show that $t = c_0 \langle b_0, h_0 \rangle \dots c_n \langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{C-Must}(\widehat{T})$, with $\widehat{T} = \text{update}_{C-Must}^\#(\widehat{S}, b_n)$.

Because $s \in \gamma_{C-Must}(\widehat{S})$, we have that $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s \langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .

It remains to show that the second constraint in (31) holds, i.e.,

$$\forall i, 0 \leq i < n + 1 : b_i \in CS_{i+1}(t) \vee |CS_i(t)| \leq \widehat{T}(b_i), \quad (59)$$

where $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_{n+1}) := \{b_j \mid i \leq j < n + 1\}$.

In order to show that (59) holds, we distinguish two cases based on the value of i :

1. $i = n$:

Observe that $CS_n(t) = \{b_n\}$.

Due to the second case in the definition of $update_{C-Must}^\#$, $\widehat{T}(b_n) = 1$, and so

$$|CS_n(t)| = |\{b_n\}| = 1 \leq 1 = \widehat{T}(b_n).$$

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-Must}(\widehat{S})$.

We distinguish two cases:

a. $b_i \in CS_{i+1}(s)$:

As $CS_{i+1}(s) = CS_{i+1}(t) \cup \{b_n\}$, the fact that $b_i \in CS_{i+1}(s)$ implies $b_i \in CS_{i+1}(t)$.

b. $b_i \notin CS_{i+1}(s)$ and thus $|CS_i(s)| \leq \widehat{S}(b_i)$:

We distinguish two cases:

i. $b_i \neq b_n$:

Then $CS_i(t) = CS_i(s) \cup \{b_n\}$.

Because $1 \leq |CS_i(s)| \leq \widehat{S}(b_i)$ and $b_i \neq b_n$, the third or fourth case in $update_{C-Must}^\#$ applies. Thus $|CS_i(t)| = |CS_i(s) \cup \{b_n\}| \leq \widehat{S}(b_i) + 1 \leq \widehat{T}(b_i)$.

ii. $b_i = b_n$:

Then $b_i \in CS_{i+1}(t) = CS_{i+1}(s) \cup \{b_i\}$.

■ Let $\widehat{S} \in C_{C-Must}^\#$ and $b \in \mathcal{B}$ be arbitrary.

To show that (10) is satisfied, assume $classify_{C-Must}^\#(\widehat{S}, b)$ holds and thus $\widehat{S}(b) < k$. Let $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ be an arbitrary trace in $\gamma_{C-Must}(\widehat{S})$. Let b_i be the last occurrence of b in the trace. If b does not occur in the trace, then (10) holds by the second disjunct. Otherwise, $b_i \notin CS_{i+1}(s)$ and so $|CS_i(s)| \leq \widehat{S}(b)$. As $\widehat{S}(b) \leq k$ and $s \in \text{LRUCACHETRACES}$, we can apply Lemma 9 to the suffix $c_i \langle b_i, h_i \rangle \dots c_n$ to prove that $b \in c_n$. ◀

► **Theorem 14** (*Global-CS vs. Block-CS**). *C-Must is incomparable to Global-CS and Block-CS.*

► **Theorem 15** (*Soundness of Conditional May*). *C-May is a sound persistence analysis.*

Proof. We show that *C-May* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

■ Equation (6) is trivially satisfied, because sequences consisting only of the initial state c_0 only are unconstrained in the definition of γ_{C-May} .

■ Let $\widehat{S} \sqsubseteq_{C-May} \widehat{T}$. Let $s = c_0 \langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{C-May}(\widehat{S})$. Because $\widehat{S}(b_i) \geq \widehat{T}(b_i)$ for all i , and \geq is transitive, s is also an element of $\gamma_{C-May}(\widehat{T})$, which shows that (7) is satisfied.

■ Let $\widehat{S} \in C_{C-May}^\#$, $b \in \mathcal{B}$, and $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-May}(\widehat{S})$ be arbitrary.

To show that (8) is satisfied, we have to show that $t = c_0 \langle b_0, h_0 \rangle \dots c_n \langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{C-May}(\widehat{T})$, with $\widehat{T} = \text{update}_{C-May}^\#(\widehat{S}, b_n)$.

Because $s \in \gamma_{C-May}(\widehat{S})$, we have that $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s \langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .

It remains to show that the second constraint in (37) holds, i.e.,

$$\forall i : 0 \leq i < n + 1 : b_i \in CS_{i+1}(t) \vee |CS_i(t)| \geq \widehat{T}(b_i), \quad (60)$$

where $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_{n+1}) := \{b_j \mid i \leq j < n + 1\}$.

In order to show that (60) holds, we distinguish two cases based on the value of i :

1. $i = n$:

$CS_n(t) = \{b_n\}$. Due to the first case in the definition of $update_{C-May}^\#$, $\widehat{T}(b_n) = 1$, and so $|CS_n(t)| \geq \widehat{T}(b_n)$.

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee |CS_i(s)| \geq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C-May}(\widehat{S})$.

We distinguish two cases:

a. $b_i \in CS_{i+1}(s)$:

As $CS_{i+1}(s) = CS_{i+1}(t) \cup \{b_n\}$, the fact that $b_i \in CS_{i+1}(s)$ implies $b_i \in CS_{i+1}(t)$.

b. $b_i \notin CS_{i+1}(s)$ and thus $|CS_i(s)| \geq \widehat{S}(b_i)$:

We distinguish two cases:

i. $b_i \neq b_n$:

We distinguish two cases:

A. $\widehat{S}(b_n) < \widehat{S}(b_i)$:

Then $|CS_i(t)| \geq |CS_i(s)| \geq \widehat{S}(b_i) = \widehat{T}(b_i)$, as the second case of $update_{C-May}^\#$ applies.

B. $\widehat{S}(b_n) \geq \widehat{S}(b_i)$:

Then, by the definition of $update_{C-May}^\#$, $\widehat{T}(b_i) \leq \widehat{S}(b_i) + 1$.

Let j be the index of the last occurrence of b_n in s . We distinguish two cases:

- $i > j$:

Then $b_j \notin CS_i(s)$. Thus

$$|CS_i(t)| = |CS_i(s) \cup \{b_n\}| = |CS_i(s)| + 1 \geq \widehat{S}(b_i) + 1 \geq \widehat{T}(b_i).$$

- $i < j$:

Then $b_i, b_j \in CS_i(s)$ and $b_i \notin CS_j(s)$ and $CS_i(s) \supseteq CS_j(s)$. Thus

$$|CS_i(t)| = |CS_i(s) \cup \{b_j\}| \geq |CS_j(s)| + 1 \geq \widehat{S}(b_j) + 1 = \widehat{S}(b_n) + 1 \geq \widehat{T}(b_i).$$

ii. $b_i = b_n$:

Then $b_i \in CS_i(t) = CS_i(s) \cup \{b_i\}$.

■ Let $\widehat{S} \in C_{C-May}^\#$ and $b \in \mathcal{B}$ be arbitrary.

To show that (10) is satisfied, assume $classify_{C-May}^\#(\widehat{S}, b)$ holds and thus $\widehat{S}(b) = \infty$ or $|C_i(\widehat{S}, b)| < i$ for some $i \leq k$. Let $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ be an arbitrary trace in $\gamma_{C-May}(\widehat{S})$. Let i' be the index of the last occurrence of b in the trace. If b does not occur in the trace, then (10) holds by the second disjunct. If $\widehat{S}(b) = \infty$ holds, then b is guaranteed not to occur in the trace s .

Otherwise, $|C_i(\widehat{S}, b)| = |\{b' \in \mathcal{B} \mid b' \neq b \wedge \widehat{S}(b') \leq i\}| < i$ and $b_{i'} \notin CS_{i'+1}(s)$.

We will show that $|CS_{i'}(s)| \leq i$.

Assume for a contradiction that $|CS_{i'}(s)| > i$. Let $j > i'$ be an index such that $|CS_j(s)| = i$, which must then exist as $|CS_{i'}(s)| > i$ and $CS_i(s)$ is monotonically decreasing in i and eventually reaches $|CS_n(s)| = 1$.

For each element b' of $CS_j(s)$, we must have $\widehat{S}(b') \leq i$ as $s \in \gamma_{C-May}(\widehat{S})$. So $CS_j(s) \subseteq C_i(\widehat{S}, b)$. Thus $i = |CS_j(s)| \leq |C_i(\widehat{S}, b)|$, which contradicts the fact that $|C_i(\widehat{S}, b)| < i$.

Thus $|CS_{i'}(s)| \leq i$. As $i \leq k$ and $s \in \text{LRUCACHE TRACES}$, we can apply Lemma 9 to the suffix $c_{i'} \langle b_{i'}, h_{i'} \rangle \dots c_n$ to prove that $b \in c_n$. ◀

► **Theorem 16** (*C-May vs. Global-CS*). *C-May is more precise than Global-CS.*

Proof. We will show this by making use of Theorem 7. To this end, we need to define a function $\gamma_{CS \rightarrow May} : C_{Global-CS}^{\#} \rightarrow C_{C-May}^{\#}$ that satisfies conditions (11), (12), (13), (15), and (16).

We define $\gamma_{CS \rightarrow May}$ as follows:

$$\gamma_{CS \rightarrow May}(\widehat{S}) := \lambda b. \begin{cases} \infty & : b \notin \widehat{S} \\ 1 & : b \in \widehat{S} \end{cases} \quad (61)$$

The rationale is that if $b \notin \widehat{S}$ then it has not yet been accessed and thus ∞ is a sound lower bound on the size of b 's conflict set. On the other hand, if $b \in \widehat{S}$, and thus may have been accessed, then 1 is the best sound lower bound on the size of b 's conflict set that can be given, as the access to b may have been the final one in the cache trace.

- Proof of satisfaction of (11): $\gamma_{CS \rightarrow May}(\widehat{\mathcal{I}_{Global-CS}}) = \gamma_{CS \rightarrow May}(\emptyset) = \lambda b. \infty = \widehat{\mathcal{I}_{C-May}}$.
- Proof of satisfaction of (12): Let \widehat{S}, \widehat{T} be arbitrary abstract traces from $C_{Global-CS}^{\#}$. Assume $\widehat{S} \sqsubseteq_{Global-CS} \widehat{T}$, i.e., $\widehat{S} \subseteq \widehat{T}$.

Then $\forall b \in \widehat{S} : \gamma_{CS \rightarrow May}(\widehat{S})(b) = 1 = \gamma_{CS \rightarrow May}(\widehat{T})(b)$ and

$$\forall b \notin \widehat{S} : \gamma_{CS \rightarrow May}(\widehat{S})(b) = \infty \geq \gamma_{CS \rightarrow May}(\widehat{T})(b),$$

which implies

$$\forall b : \gamma_{CS \rightarrow May}(\widehat{S})(b) \geq \gamma_{CS \rightarrow May}(\widehat{T})(b), \text{ i.e., } \gamma_{CS \rightarrow May}(\widehat{S}) \sqsubseteq_{C-May} \gamma_{CS \rightarrow May}(\widehat{T}),$$

which shows (12).

- Proof of satisfaction of (13): We need to show that

$$\begin{aligned} \forall \widehat{S} \in C_{Global-CS}^{\#}, b \in \mathcal{B} : \\ \text{update}_{C-May}^{\#}(\gamma_{CS \rightarrow May}(\widehat{S}), b) \sqsubseteq_{C-May} \gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b)). \end{aligned}$$

Let $\widehat{S} \in C_{Global-CS}^{\#}$ and $b \in \mathcal{B}$ be arbitrary.

Due to the definition of \sqsubseteq_{C-May} , we need to show

$$\forall b' \in \mathcal{B} : \text{update}_{C-May}^{\#}(\gamma_{CS \rightarrow May}(\widehat{S}), b)(b') \geq \gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b))(b').$$

To prove this, let $b' \in \mathcal{B}$ be arbitrary.

We distinguish two cases:

1. $b' \in \text{update}_{Global-CS}^{\#}(\widehat{S}, b)$:
Then $\gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b))(b') = 1$, which is the smallest value that a block may be assigned to in $C_{C-May}^{\#}$, and so

$$\text{update}_{C-May}^{\#}(\gamma_{CS \rightarrow May}(\widehat{S}), b)(b') \geq 1 = \gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b))(b').$$

2. $b' \notin \text{update}_{Global-CS}^{\#}(\widehat{S}, b)$:
Then, $b' \neq b$ and $b' \notin \widehat{S}$. Thus, $\gamma_{CS \rightarrow May}(\widehat{S})(b') = \infty$ and the fifth case in the definition of $\text{update}_{C-May}^{\#}$ applies, so

$$\text{update}_{Global-CS}^{\#}(\gamma_{CS \rightarrow May}(\widehat{S}), b)(b') = \infty \geq \infty = \gamma_{CS \rightarrow May}(\text{update}_{Global-CS}^{\#}(\widehat{S}, b))(b').$$

- Proof of (15): Let $\widehat{S} \in C_{Global-CS}^{\#}$ and $b \in \mathcal{B}$ be arbitrary. Assume $\text{classify}_{Global-CS}^{\#}(\widehat{S}, b)$ holds. Then, either $b \notin \widehat{S}$ or $|\widehat{S}| \leq k$:

- If $b \notin \widehat{S}$, then $\gamma_{CS \rightarrow May}(\widehat{S})(b) = \infty$ and by definition, $classify_{C-May}^\#(\gamma_{CS \rightarrow May}(\widehat{S}), b)$ holds as well.
- If $b \in \widehat{S}$ and thus $|\widehat{S}| \leq k$ we have exactly $|\widehat{S}|$ blocks b' for which $\gamma_{CS \rightarrow May}(\widehat{S})(b) \leq k$. Thus, the second disjunct of $classify_{C-May}^\#$ applies and $classify_{C-May}^\#(\gamma_{CS \rightarrow May}(\widehat{S}), b)$ holds.
- Proof of satisfaction of (16): Let $\widehat{S}, \widehat{T} \in C_{C-May}^\#$ with $\widehat{S} \sqsubseteq_{C-May} \widehat{T}$ and $b \in \mathcal{B}$ be arbitrary. Assume $classify_{C-May}^\#(\widehat{T}, b)$ holds.
 - If $\widehat{T}(b) = \infty$, then $\widehat{S}(b) = \infty$ as $\widehat{S}(b) \geq \widehat{T}(b)$. Then, $classify_{C-May}^\#(\widehat{S}, b)$ holds as well.
 - If $\widehat{T}(b) \leq k + 1$, then there is an $i \leq k$, such that $|C_i(\widehat{T}, b)| < i$. As $\widehat{S} \sqsubseteq_{C-May} \widehat{T}$ we have $\widehat{S}(b') \geq \widehat{T}(b')$ for all $b' \in \mathcal{B}$. So $C_i(\widehat{S}, b) = \{b' \in \mathcal{B} \mid b' \neq b \wedge \widehat{T}(b') \leq \widehat{S}(b') \leq i\} \subseteq C_i(\widehat{T}, b)$, and thus $|C_i(\widehat{S}, b)| \leq |C_i(\widehat{T}, b)| < i$, which implies $classify_{C-May}^\#(\widehat{S}, b)$.

To see that *C-May* is more precise than *Global-CS*, consider the example in Figure 4a. Here, x is classified as persistent by *C-May*, but not by *Global-CS*. ◀

► **Theorem 17** (*Block-CS vs. C-May*). *Block-CS* is more precise than *C-May*.

Proof. We will show this by making use of Theorem 7. To this end, we need to define a function $\gamma_{May \rightarrow CS} : C_{C-May}^\# \rightarrow C_{Block-CS}^\#$ that satisfies conditions (11), (12), (13), (15), and (16).

We define $\gamma_{May \rightarrow CS}$ as follows:

$$\gamma_{May \rightarrow CS}(\widehat{S}) := \lambda b. \begin{cases} \emptyset & : \widehat{S}(b) = \infty \\ \{b\} \cup C_n(\widehat{S}, b) & : \widehat{S}(b) \neq \infty \wedge n = \min\{i \in \mathbb{N} \mid |C_i(\widehat{S}, b)| < i\} \end{cases} \quad (62)$$

where $C_i(\widehat{S}, b) := \{b' \in \mathcal{B} \mid b' \neq b \wedge \widehat{S}(b') \leq i\}$.

- Proof of satisfaction of (11): $\gamma_{May \rightarrow CS}(\widehat{\mathcal{I}}_{C-May}) = \gamma_{May \rightarrow CS}(\lambda b. \infty) = \lambda b. \emptyset = \widehat{\mathcal{I}}_{Block-CS}$.
- Proof of satisfaction of (12): Let \widehat{S}, \widehat{T} be arbitrary abstract traces from $C_{C-May}^\#$. Assume $\widehat{S} \sqsubseteq_{C-May} \widehat{T}$, i.e., $\forall b : \widehat{S}(b) \geq \widehat{T}(b)$.

We need to show that

$$\gamma_{May \rightarrow CS}(\widehat{S}) \sqsubseteq_{Block-CS} \gamma_{May \rightarrow CS}(\widehat{T}) \Leftrightarrow \forall b : \gamma_{May \rightarrow CS}(\widehat{S})(b) \subseteq \gamma_{May \rightarrow CS}(\widehat{T})(b).$$

Let b be arbitrary. We will show $\gamma_{May \rightarrow CS}(\widehat{S})(b) \subseteq \gamma_{May \rightarrow CS}(\widehat{T})(b)$ by the following case distinction:

- $\widehat{S}(b) = \infty$: Then $\gamma_{May \rightarrow CS}(\widehat{S})(b) = \emptyset$, which is a subset of any set, in particular $\gamma_{May \rightarrow CS}(\widehat{T})(b)$.
- $\widehat{S}(b) \leq k + 1$: Let $n = \min\{i \in \mathbb{N} \mid |C_i(\widehat{T}, b)| < i\}$ and thus $\gamma_{May \rightarrow CS}(\widehat{T})(b) = \{b\} \cup C_n(\widehat{T}, b)$. As $\forall b' : \widehat{S}(b') \geq \widehat{T}(b')$, $C_i(\widehat{S}, b) \subseteq C_i(\widehat{T}, b)$ and so $|C_i(\widehat{S}, b)| \leq |C_i(\widehat{T}, b)| < i$. Thus, $\gamma_{May \rightarrow CS}(\widehat{S})(b) = \{b\} \cup C_{n'}(\widehat{S}, b)$, with $n' = \min\{i \in \mathbb{N} \mid |C_i(\widehat{S}, b)| < i\} \leq n$. As CS_i is monotone in i and $CS_i(\widehat{S}, b) \subseteq CS_i(\widehat{T}, b)$, we have $\gamma_{May \rightarrow CS}(\widehat{S})(b) \subseteq \gamma_{May \rightarrow CS}(\widehat{T})(b)$.

- Proof of satisfaction of (13): We need to show that

$$\forall \widehat{S} \in C_{C-May}^\#, b \in \mathcal{B} : update_{Block-CS}^\#(\gamma_{May \rightarrow CS}(\widehat{S}), b) \sqsubseteq_{Block-CS} \gamma_{May \rightarrow CS}(update_{C-May}^\#(\widehat{S}, b)).$$

Let \widehat{S} and b be arbitrary, and let $\widehat{T} = update_{C-May}^\#(\widehat{S}, b)$. Then, we need to show for all $b' \in \mathcal{B}$:

$$update_{Block-CS}^\#(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') \subseteq \gamma_{May \rightarrow CS}(update_{C-May}^\#(\widehat{S}, b))(b') = \gamma_{May \rightarrow CS}(\widehat{T})(b').$$

To prove this we distinguish two cases:

1. $b' = b$:

Then $update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \{b'\}$ and as $update_{C-May}^{\#}(\widehat{S}, b)(b') \neq \infty$, we have $\gamma_{May \rightarrow CS}(update_{C-May}^{\#}(\widehat{S}, b))(b') \supseteq \{b'\}$, and so

$$update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \{b'\} \subseteq \gamma_{May \rightarrow CS}(update_{C-May}^{\#}(\widehat{S}, b))(b').$$

2. $b' \neq b$:

We further distinguish two cases:

a. $\widehat{S}(b') = \infty$:

Then, it is easy to see that

$$update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \emptyset = \gamma_{May \rightarrow CS}(update_{C-May}^{\#}(\widehat{S}, b))(b').$$

b. $\widehat{S}(b') \leq k + 1$:

Let $n = \min\{i \in \mathbb{N} \mid |C_i(\widehat{S}, b')| < i\}$ and $n' = \min\{i \in \mathbb{N} \mid |C_i(\widehat{T}, b')| < i\}$.

We further distinguish three cases:

i. $\widehat{S}(b) < n$:

Then $b \in \gamma_{May \rightarrow CS}(\widehat{S})(b')$ and so

$$update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \gamma_{May \rightarrow CS}(\widehat{S})(b').$$

Observe that

- $C_1(\widehat{T}, b') = \{b\}$,
- $C_j(\widehat{T}, b') = C_{j-1}(\widehat{S}, b') \dot{\cup} \{b\}$ for $j \in \{2, \dots, \widehat{S}(b)\}$, and
- $C_j(\widehat{T}, b') = C_j(\widehat{S}, b')$ for $j \in \{\widehat{S}(b) + 1, \dots, n\}$.

Due to the definition of n , we have that $|C_i(\widehat{S}, b')| \geq i$ for all $i < n$, and so:

- $|C_1(\widehat{T}, b')| \geq 1$,
- $|C_j(\widehat{T}, b')| \geq |C_{j-1}(\widehat{S}, b')| + 1 \geq j - 1 + 1 = j$ for $j \in \{2, \dots, \widehat{S}(b)\}$, and
- $|C_j(\widehat{T}, b')| = |C_j(\widehat{S}, b')| \geq j$ for $j \in \{\widehat{S}(b) + 1, \dots, n\}$.

As a consequence $n' \geq n$ and thus

$$\gamma_{May \rightarrow CS}(\widehat{T})(b') \supseteq \gamma_{May \rightarrow CS}(\widehat{S})(b') = update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b').$$

ii. $\widehat{S}(b) = n$:

This case is impossible:

As $n = \min\{i \in \mathbb{N} \mid |C_i(\widehat{S}, b')| < i\}$, we have $|C_{n-1}(\widehat{S}, b)| \geq n - 1$.

However, $C_n(\widehat{S}, b') \supseteq C_{n-1}(\widehat{S}, b) \dot{\cup} \{b\}$, if $\widehat{S}(b) = n$, which implies $|C_n(\widehat{S}, b')| \geq n$, which contradicts of our definition of n , which implies $|C_n(\widehat{S}, b')| < n$.

iii. $\widehat{S}(b) > n$:

Then $b \notin \gamma_{May \rightarrow CS}(\widehat{S})(b')$ and so

$$update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b') = \gamma_{May \rightarrow CS}(\widehat{S})(b') \dot{\cup} \{b\}.$$

Observe that

- $C_1(\widehat{T}, b') = \{b\}$, and
- $C_j(\widehat{T}, b') = C_{j-1}(\widehat{S}, b') \dot{\cup} \{b\}$ for $j \in \{2, \dots, n\}$.

Due to the definition of n , we have that $|C_i(\widehat{S}, b')| \geq i$ for all $i < n$, and so:

- $|C_1(\widehat{T}, b')| \geq 1$, and
- $|C_j(\widehat{T}, b')| \geq |C_{j-1}(\widehat{S}, b')| + 1 \geq j - 1 + 1 = j$ for $j \in \{2, \dots, n\}$.

Thus $n' \geq n + 1$. Also, $C_{n+1}(\widehat{T}, b') \supseteq C_n(\widehat{S}, b') \dot{\cup} \{b\}$ and thus

$$\begin{aligned} \gamma_{May \rightarrow CS}(\widehat{T})(b') &\supseteq C_n(\widehat{S}, b') \dot{\cup} \{b\} \\ &= \gamma_{May \rightarrow CS}(\widehat{S})(b') \cup \{b\} = update_{Block-CS}^{\#}(\gamma_{May \rightarrow CS}(\widehat{S}), b)(b'). \end{aligned}$$

- Proof of satisfaction of (15): Let $\widehat{S} \in C_{C\text{-May}}^\#$ and $b \in \mathcal{B}$ be arbitrary. Assume $\text{classify}_{C\text{-May}}^\#(\widehat{S}, b)$ holds.

Then either $\widehat{S}(b) = \infty$ or $\exists i \leq k : |C_i(\widehat{S}, b)| < i$.

In the first case, $\gamma_{\text{May} \rightarrow \text{CS}}(\widehat{S})(b) = \emptyset$ and so $\text{classify}_{\text{Block-CS}}^\#(\gamma_{\text{May} \rightarrow \text{CS}}(\widehat{S}), b)$ holds.

In the second case, $\gamma_{\text{May} \rightarrow \text{CS}}(\widehat{S})(b) = \{b\} \cup C_n(\widehat{S}, b)$ with $|C_n(\widehat{S}, b)| < k$ and thus

$|\gamma_{\text{May} \rightarrow \text{CS}}(\widehat{S})(b)| \leq k$, which implies that $\text{classify}_{\text{Block-CS}}^\#(\gamma_{\text{May} \rightarrow \text{CS}}(\widehat{S}), b)$ holds as well.

- Proof of satisfaction of (16): Let $\widehat{S}, \widehat{T} \in C_{\text{Block-CS}}^\#$ with $\widehat{S} \sqsubseteq_{\text{Block-CS}} \widehat{T}$ and $b \in \mathcal{B}$ be arbitrary. Assume $\text{classify}_{\text{Block-CS}}^\#(\widehat{T}, b)$ holds. Then, $|\widehat{T}(b)| \leq k$. As $\widehat{S}(b) \subseteq \widehat{T}(b)$ this implies $|\widehat{S}(b)| \leq k$ and so $\text{classify}_{\text{Block-CS}}^\#(\widehat{S}, b)$ holds as well.

To see that *Block-CS* is more precise than *C-May*, consider the example in Figure 4b. Here, w and x are classified as persistent by *Block-CS*, but not by *C-May*. ◀

► **Definition 18** (Direct Product). *The direct product $A \times B$ of two persistence analyses A and B is the tuple $A \times B = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}}_{A \times B}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, \text{update}_{A \times B}^\#, \text{classify}_{A \times B}^\# \rangle$ with*

$$\begin{aligned} C_{A \times B}^\# &:= C_A^\# \times C_B^\#, \\ \gamma_{A \times B}(\widehat{S}_A, \widehat{S}_B) &:= \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B), \\ \widehat{\mathcal{I}}_{A \times B} &:= \langle \widehat{\mathcal{I}}_A, \widehat{\mathcal{I}}_B \rangle, \\ \langle \widehat{S}_A, \widehat{S}_B \rangle \sqsubseteq_{A \times B} \langle \widehat{T}_A, \widehat{T}_B \rangle &:\Leftrightarrow \widehat{S}_A \sqsubseteq_A \widehat{T}_A \wedge \widehat{S}_B \sqsubseteq_B \widehat{T}_B, \\ \langle \widehat{S}_A, \widehat{S}_B \rangle \sqcup_{A \times B} \langle \widehat{T}_A, \widehat{T}_B \rangle &:= \langle \widehat{S}_A \sqcup_A \widehat{T}_A, \widehat{S}_B \sqcup_B \widehat{T}_B \rangle, \\ \text{update}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) &:= \langle \text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b) \rangle, \\ \text{classify}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) &:= \text{classify}_A^\#(\widehat{S}_A, b) \vee \text{classify}_B^\#(\widehat{S}_B, b). \end{aligned}$$

► **Theorem 19** (Soundness of Direct Product). *The direct product $A \times B$ of two sound persistence analyses A and B that satisfy (6), (7), (8), and (10) is a sound persistence analysis.*

Proof. We show that $A \times B$ satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

- As A and B satisfy (6), we have $\mathcal{I}_C \subseteq \gamma_A(\widehat{\mathcal{I}}_A)$ and $\mathcal{I}_C \subseteq \gamma_B(\widehat{\mathcal{I}}_B)$. Thus,

$$\mathcal{I}_C \subseteq \gamma_A(\widehat{\mathcal{I}}_A) \cap \gamma_B(\widehat{\mathcal{I}}_B) \stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_{A \times B}(\widehat{\mathcal{I}}_A, \widehat{\mathcal{I}}_B) \stackrel{\text{Def. } \widehat{\mathcal{I}}_{A \times B}}{=} \gamma_{A \times B}(\widehat{\mathcal{I}}_{A \times B}),$$

and so $A \times B$ satisfies (6).

- For (7), we have to show that

$$\forall \widehat{S}, \widehat{T} \in C_{A \times B}^\# : \widehat{S} \sqsubseteq_{A \times B} \widehat{T} \Rightarrow \gamma_{A \times B}(\widehat{S}) \subseteq \gamma_{A \times B}(\widehat{T}).$$

Let $\widehat{S} = \langle \widehat{S}_A, \widehat{S}_B \rangle$ and $\widehat{T} = \langle \widehat{T}_A, \widehat{T}_B \rangle$ be arbitrary. Assume that $\widehat{S} \sqsubseteq_{A \times B} \widehat{T}$, otherwise the implication trivially holds. Then, we have $\widehat{S}_A \sqsubseteq_A \widehat{T}_A$ and $\widehat{S}_B \sqsubseteq_B \widehat{T}_B$ by definition of $\sqsubseteq_{A \times B}$. As A and B satisfy (7) this implies $\gamma_A(\widehat{S}_A) \subseteq \gamma_A(\widehat{T}_A)$ and $\gamma_B(\widehat{S}_B) \subseteq \gamma_B(\widehat{T}_B)$, and so we have

$$\begin{aligned} \gamma_{A \times B}(\widehat{S}) &\stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B) \\ &\subseteq \gamma_A(\widehat{T}_A) \cap \gamma_B(\widehat{T}_B) \\ &\stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_{A \times B}(\widehat{T}). \end{aligned}$$

- For (8), we have to show that

$$\begin{aligned} \forall \widehat{S} \in C_{A \times B}^\#, b \in \mathcal{B} : \{t.c(b, h)c' \mid t.c \in \gamma_{A \times B}(\widehat{S}) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\ \subseteq \gamma_{A \times B}(\text{update}_{A \times B}^\#(\widehat{S}, b)). \end{aligned}$$

Let $\widehat{S} = \langle \widehat{S}_A, \widehat{S}_B \rangle \in C_{A \times B}^\#$ and $b \in \mathcal{B}$ be arbitrary. Further, let $t.c \in \gamma_{A \times B}(\widehat{S})$ be arbitrary. We will show that $t.c \langle b, h \rangle c' \in \gamma_{A \times B}(\text{update}_{A \times B}^\#(\widehat{S}, b))$, with $h = \text{eff}_c(c, b)$ and $c' = \text{update}_c(c, b)$. By definition of $\gamma_{A \times B}$, $t.c \in \gamma_A(\widehat{S}_A)$ and $t.c \in \gamma_B(\widehat{S}_B)$. Because A and B satisfy (8), we have both $t.c \langle b, h \rangle c' \in \gamma_A(\text{update}_A^\#(\widehat{S}_A, b))$ and $t.c \langle b, h \rangle c' \in \gamma_B(\text{update}_B^\#(\widehat{S}_B, b))$, and thus:

$$\begin{aligned} t.c \langle b, h \rangle c' &\in \gamma_A(\text{update}_A^\#(\widehat{S}_A, b)) \cap \gamma_B(\text{update}_B^\#(\widehat{S}_B, b)) \\ &\stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_{A \times B}(\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b)) \\ &\stackrel{\text{Def. } \text{update}_{A \times B}^\#}{=} \gamma_{A \times B}(\text{update}_{A \times B}^\#(\widehat{S}, b)). \end{aligned}$$

■ For (10), we have to show that

$$\begin{aligned} \forall \widehat{S} \in C_{A \times B}^\#, b \in \mathcal{B} : \text{classify}_{A \times B}^\#(\widehat{S}, b) \Rightarrow \\ \forall c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \gamma_{A \times B}(\widehat{S}) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b), \end{aligned} \quad (63)$$

Let $\widehat{S} = \langle \widehat{S}_A, \widehat{S}_B \rangle \in C_{A \times B}^\#$ and $b \in \mathcal{B}$ be arbitrary. Assume $\text{classify}_{A \times B}^\#(\widehat{S}, b)$ holds, otherwise the implication holds trivially.

By the definition of $\text{classify}_{A \times B}^\#$, $\text{classify}_A^\#(\widehat{S}_A, b)$ holds or $\text{classify}_B^\#(\widehat{S}_B, b)$ holds. Assume $\text{classify}_A^\#(\widehat{S}_A, b)$ holds. The case that $\text{classify}_B^\#(\widehat{S}_B, b)$ is analogous.

As A satisfies (10), we have $\forall c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \gamma_A(\widehat{S}_A) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b)$. As by definition, $\gamma_{A \times B}(\widehat{S}) \subseteq \gamma_A(\widehat{S}_A)$, we also have: $\forall c_0 \langle b_0, h_0 \rangle c_1 \langle b_1, h_1 \rangle \dots c_n \in \gamma_{A \times B}(\widehat{S}) : b \in c_n \vee (\forall i, 0 \leq i < n : b_i \neq b)$. ◀

► **Theorem 20** (Precision of Direct Product*). *The direct product $A \times B$ of two persistence analyses A and B is at least as precise as A and B , i.e., $A \times B \succeq A$ and $A \times B \succeq B$.*

► **Corollary 21** (Precision of Direct Product*). *The direct product $A \times B$ of two incomparable persistence analyses A and B is more precise than A and B , i.e., $A \times B \succ A$ and $A \times B \succ B$.*

► **Definition 22** (State Reduction). *Let A and B be persistence analyses. A reduction operator for A in the context of B is a function $\text{red} : C_A^\# \times C_B^\# \rightarrow C_A^\#$ that is reductive and that preserves concretizations, i.e., for all $\widehat{S}_A \in C_A^\#, \widehat{S}_B \in C_B^\#$:*

$$\text{red}(\widehat{S}_A, \widehat{S}_B) \sqsubseteq_A \widehat{S}_A, \quad (42)$$

$$\gamma_A(\text{red}(\widehat{S}_A, \widehat{S}_B)) \cap \gamma_B(\widehat{S}_B) = \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B). \quad (43)$$

► **Theorem 23** (State Reduction). *Let A and B be sound persistence analyses that satisfy (6), (7), (8), and (10), and let red be a reduction operator for A in the context of B . Let the reduced update be defined as follows:*

$$\text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) := (\text{red}(\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b)), \text{update}_B^\#(\widehat{S}_B, b))$$

Then, $A \times B' = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}}_{A \times B}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, \text{red-upd}, \text{classify}_{A \times B}^\# \rangle$ is a sound persistence analysis that is at least as precise as $A \times B$, i.e., $A \times B' \succeq A \times B$.

Proof. We know that $A \times B$ is a sound persistence analysis from Theorem 19 that satisfies (6), (7), (8), and (10). The only condition from Theorem 4 that involves the update function is (8). Thus all conditions but (8) are fulfilled by $A \times B'$ as they are fulfilled by $A \times B$.

To show that (8) is satisfied, we argue that $\gamma_{A \times B}(\text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) =$

$\gamma_{A \times B}(\text{update}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b))$ for all $\langle \widehat{S}_A, \widehat{S}_B \rangle \in C_{A \times B}^\#$ and $b \in \mathcal{B}$:

$$\begin{aligned}
& \gamma_{A \times B}(\text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) \\
& \stackrel{\text{Def. red-upd}}{=} \gamma_{A \times B}(\text{red}(\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b)), \text{update}_B^\#(\widehat{S}_B, b)) \\
& \stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_A(\text{red}(\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b))) \cap \gamma_B(\text{update}_B^\#(\widehat{S}_B, b)) \\
& \stackrel{(43)}{=} \gamma_A(\text{update}_A^\#(\widehat{S}_A, b)) \cap \gamma_B(\text{update}_B^\#(\widehat{S}_B, b)) \\
& \stackrel{\text{Def. } \gamma_{A \times B}}{=} \gamma_{A \times B}(\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b)) \\
& \stackrel{\text{Def. } \text{update}_{A \times B}^\#}{=} \gamma_{A \times B}(\text{update}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b))
\end{aligned}$$

We can easily show that $A \times B'$ is at least as precise as $A \times B$ by making use of Theorem 7. To this end, we need to define a function $\gamma_{A \times B' \rightarrow A \times B}$ that satisfies conditions (11), (12), (13), (15), and (16). We define $\gamma_{A \times B' \rightarrow A \times B}$ to be the identity function. Conditions (11), (12), (15), and (16) trivially hold as the left and right hand sides of these inequalities are the same. Finally (13) reduces to $\forall \langle \widehat{S}_A, \widehat{S}_B \rangle \in C_{A \times B}^\#, b \in \mathcal{B} : \text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) \sqsubseteq_{A \times B} \text{update}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)$, which follows from (42):

$$\begin{aligned}
\text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) &= (\text{red}(\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b)), \text{update}_B^\#(\widehat{S}_B, b)) \\
&\stackrel{(42)}{\sqsubseteq_{A \times B}} (\text{update}_A^\#(\widehat{S}_A, b), \text{update}_B^\#(\widehat{S}_B, b)) \\
&= (\text{update}_{A \times B}^\#(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) \quad \blacktriangleleft
\end{aligned}$$

► **Definition 24** (Cooperative Update). *Let A and B be two persistence analyses. A cooperative update for A in the context of B is a function $\text{coop-upd} : (C_A^\# \times C_B^\#) \times \mathcal{B} \rightarrow C_A^\#$, such that:*

$$\begin{aligned}
& \forall \langle \widehat{S}_A, \widehat{S}_B \rangle \in C_A^\# \times C_B^\#, b \in \mathcal{B} : \\
& \{t.c\langle b, h \rangle c' \mid t.c \in \gamma_A(\widehat{S}_A) \cap \gamma_B(\widehat{S}_B) \wedge h = \text{eff}_C(c, b) \wedge c' = \text{update}_C(c, b)\} \\
& \qquad \qquad \qquad \subseteq \gamma_A(\text{coop-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b)) \quad (44)
\end{aligned}$$

► **Theorem 25** (Cooperative Update*). *Let A and B be sound persistence analyses that satisfy (6), (7), (8), and (10), and let coop-upd be a cooperative update function for A in the context of B . Let the reduced update be defined as follows:*

$$\text{red-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b) := (\text{coop-upd}(\langle \widehat{S}_A, \widehat{S}_B \rangle, b), \text{update}_B^\#(\widehat{S}_B, b))$$

Then, $A \times B' = \langle C_{A \times B}^\#, \gamma_{A \times B}, \widehat{\mathcal{I}}_{A \times B}, \sqsubseteq_{A \times B}, \sqcup_{A \times B}, \text{red-upd}, \text{classify}_{A \times B}^\# \rangle$ is a sound persistence analysis.

► **Theorem 26** (Soundness of the State Reduction between C -Must and $Block$ -CS*). *The function $\text{reduce}_{C\text{-Must} \times Block\text{-CS}}$ is a reduction operator for C -Must in the context of $Block$ -CS.*

► **Theorem 27** (Soundness of the State Reduction between C -Must and C -May). *The operator $\text{reduce}_{C\text{-Must} \times C\text{-May}}$ is a reduction operator for C -Must in the context of C -May.*

Proof. $\text{reduce}_{C\text{-Must} \times Block\text{-CS}}$ is reductive, as $\min \{ \widehat{S}_{C\text{-Must}}(b), \dots \} \leq \widehat{S}_{C\text{-Must}}(b)$.

It remains to show that for all $\widehat{S}_1 \in C_{C\text{-Must}}^\#, \widehat{S}_2 \in C_{C\text{-May}}^\#$:

$$\gamma_{C\text{-Must} + C\text{-May}}(\widehat{S}_1, \widehat{S}_2) = \gamma_{C\text{-Must} + C\text{-May}}(\text{reduce}_{C\text{-Must} \times C\text{-May}}(\widehat{S}_1, \widehat{S}_2), \widehat{S}_2). \quad (64)$$

If we can show

$$\gamma_{C\text{-Must}}(\widehat{S}_1) \cap \gamma_{C\text{-May}}(\widehat{S}_2) = \gamma_{C\text{-Must}}\left(\text{reduce}_{C\text{-Must}\times C\text{-May}}(\widehat{S}_1, \widehat{S}_2)\right) \cap \gamma_{C\text{-May}}(\widehat{S}_2), \quad (65)$$

then (64) can be shown as follows:

$$\begin{aligned} \gamma_{C\text{-Must}+C\text{-May}}(\widehat{S}_1, \widehat{S}_2) &\stackrel{\text{Def. } \gamma_{C\text{-Must}+C\text{-May}}}{=} \gamma_{C\text{-Must}}(\widehat{S}_1) \cap \gamma_{C\text{-May}}(\widehat{S}_2) \\ &\stackrel{(65)}{=} \gamma_{C\text{-Must}}\left(\text{reduce}_{C\text{-Must}\times C\text{-May}}(\widehat{S}_1, \widehat{S}_2)\right) \cap \gamma_{C\text{-May}}(\widehat{S}_2) \\ &\stackrel{\text{Def. } \gamma_{C\text{-Must}+C\text{-May}}}{=} \gamma_{C\text{-Must}+C\text{-May}}\left(\text{reduce}_{C\text{-Must}\times C\text{-May}}(\widehat{S}_1, \widehat{S}_2), \widehat{S}_2\right) \end{aligned}$$

Let us now show that (65) holds:

As $\text{reduce}_{C\text{-Must}\times\text{Block-}CS}$ is reductive and $\gamma_{C\text{-Must}}$ is monotone, we know that

$$\gamma_{C\text{-Must}}(\widehat{S}_1) \cap \gamma_{C\text{-May}}(\widehat{S}_2) \supseteq \gamma_{C\text{-Must}}\left(\text{reduce}_{C\text{-Must}\times C\text{-May}}(\widehat{S}_1, \widehat{S}_2)\right) \cap \gamma_{C\text{-May}}(\widehat{S}_2).$$

To prove that

$$\gamma_{C\text{-Must}}(\widehat{S}_1) \cap \gamma_{C\text{-May}}(\widehat{S}_2) \subseteq \gamma_{C\text{-Must}}\left(\text{reduce}_{C\text{-Must}\times C\text{-May}}(\widehat{S}_1, \widehat{S}_2)\right) \cap \gamma_{C\text{-May}}(\widehat{S}_2),$$

assume for a contradiction that there is a trace $s = c_0\langle b_0, h_0 \rangle \dots c_n$ in $\gamma_{C\text{-Must}}(\widehat{S}_1) \cap \gamma_{C\text{-May}}(\widehat{S}_2)$ that is not in $\gamma_{C\text{-Must}+C\text{-May}}\left(\text{reduce}_{C\text{-Must}\times C\text{-May}}(\widehat{S}_1, \widehat{S}_2), \widehat{S}_2\right)$.

Then, there must be an i , such that $b_i \notin CS_{i+1}$ and $|CS_i| \leq \widehat{S}_1(b_i)$, but not

$$\begin{aligned} |CS_i| &\leq \text{reduce}_{C\text{-Must}\times C\text{-May}}(\widehat{S}_1, \widehat{S}_2)(b_i) \\ &= \min\{\widehat{S}_1(b_i) \mid \{b' \in \mathcal{B} \mid b' \neq b_i \wedge \widehat{S}_2(b') < \widehat{S}_1(b_i)\} + 1\} \\ &\leq |\{b' \in \mathcal{B} \mid b' \neq b_i \wedge \widehat{S}_2(b') < \widehat{S}_1(b_i)\}| + 1. \end{aligned}$$

To reach a contradiction, we will show that $CS_i \setminus \{b_i\} \subseteq \{b' \in \mathcal{B} \mid b' \neq b_i \wedge \widehat{S}_2(b') \leq \widehat{S}_1(b_i)\}$.

Let c be an arbitrary element of $CS_i \setminus \{b_i\}$ and let j denote the index of the last occurrence of c in the trace s . As $c \in CS_i \setminus \{b_i\}$, j must be greater than i . Thus $j \geq i + 1$, and so $CS_j \subseteq CS_{i+1}$. As $b_i \notin CS_{i+1}$, we also have $b_i \notin CS_j$. So $|CS_i| \leq \widehat{S}_1(b_i)$ implies $|CS_j| < \widehat{S}_1(b_i)$.

As the trace s is in $\gamma_{C\text{-May}}(\widehat{S}_2)$, we have $|CS_j| \geq \widehat{S}_2(c)$.

Thus $\widehat{S}_2(c) \leq |CS_j| < \widehat{S}_1(b_i)$, which shows that $c \in \{b' \in \mathcal{B} \mid b' \neq b_i \wedge \widehat{S}_2(b') < \widehat{S}_1(b_i)\}$. ◀

► **Theorem 28** (Soundness of Must Analysis). *Must is a sound persistence analysis.*

Proof. We show that *Must* satisfies the conditions of Theorems 3 and 4 and is thus a sound persistence abstraction. We need to show that the abstraction satisfies (6), (7), (8), and (10).

- Equation (6) is satisfied, because $\gamma_{\text{Must}}(\widehat{\mathcal{I}}_{\text{Must}})$ represents all possible sequences.
- Let $\widehat{S} \sqsubseteq_{\text{Must}} \widehat{T}$. Let $s = c_0\langle b_0, h_0 \rangle c_1 \dots c_n$ be an arbitrary trace such that $s \in \gamma_{\text{Must}}(\widehat{S})$. Because $\widehat{S}(b_i) \leq \widehat{T}(b_i)$ for all i , and \leq is transitive, s is also an element of $\gamma_{\text{Must}}(\widehat{T})$, which shows that (7) is satisfied.
- Let $\widehat{S} \in C_{\text{Must}}^\#$, $b \in \mathcal{B}$, and $s = c_0\langle b_0, h_0 \rangle \dots c_n \in \gamma_{\text{Must}}(\widehat{S})$ be arbitrary. To show that (8) is satisfied, we have to show that $t = c_0\langle b_0, h_0 \rangle \dots c_n\langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{\text{LRU}}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{\text{LRU}}(c_n, b_n)$ is an element of $\gamma_{\text{Must}}(\widehat{T})$, with $\widehat{T} = \text{update}_{\text{Must}}^\#(\widehat{S}, b_n)$.

Because $s \in \gamma_{Must}(\widehat{S})$, we have that $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s.\langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .

It remains to show that the second and third constraint in (51) hold¹², i.e.,

$$(\forall b \in \mathcal{B} : (\forall i, 0 \leq i < n + 1 : b_i \neq b) \Rightarrow \widehat{T}(b) = \infty) \quad (66)$$

$$\wedge \forall i, 0 \leq i < n + 1 : b_i \in CS_{i+1}(t) \vee |CS_i(t)| \leq \widehat{T}(b_i) \quad (67)$$

where $CS_i(c_0 \langle b_0, h_0 \rangle \dots c_{n+1}) := \{b_j \mid i \leq j < n + 1\}$.

- Let us consider the constraint (66) first.
Let $b \in \mathcal{B}$ be arbitrary. We distinguish two cases:
 1. $\widehat{S}(b) \neq \infty$:
Then, as $s \in \gamma_{Must}(\widehat{S})$, there must be an $i, 0 \leq i < n$, such that $b_i = b$. As t is an extension of s , b_i also is part of t and thus $(\forall i, 0 \leq i < n + 1 : b_i \neq b)$ is false for t as well.
 2. $\widehat{S}(b) = \infty$:
Then, we further distinguish two cases:
 - a. $b_n = b$:
Then, the constraints holds for t , because $(\forall i : b_i \neq b)$ is false.
 - b. $b_n \neq b$:
Then, $\widehat{T}(b) = \infty$ due to the definition of $\text{update}_{Must}^\#$, where the second case applies.
With $\widehat{T}(b) = \infty$ the constraint holds trivially.
- Let us now consider the constraint (67).
Let i be arbitrary. We distinguish two cases based on i 's value:
 1. $i = n$:
 $CS_n(t) = \{b_n\}$. Due to the first case in the definition of $\text{update}_{Must}^\#$, $\widehat{T}(b_n) = 1$, and so $|CS_n(t)| \leq \widehat{T}(b_n)$.
 2. $0 \leq i < n$:
We have that $b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{Must}(\widehat{S})$.
Observe that $CS_i(t) = CS_i(s) \cup \{b_n\}$ for all $i, 0 \leq i < n$.
We distinguish two cases:
 - a. $b_i \in CS_{i+1}(s)$:
As $CS_i(t) \supseteq CS_i(s)$, we have $b_i \in CS_{i+1}(t)$.
 - b. $b_i \notin CS_{i+1}(s)$ and thus $|CS_i(s)| \leq \widehat{S}(b_i)$:
We distinguish two further cases:
 - i. $b_i = b_n$:
Then $b_i \in CS_{i+1}(t) = CS_{i+1}(s) \cup \{b_n\} = CS_{i+1}(s) \cup \{b_i\}$.
 - ii. $b_i \neq b_n$:
We distinguish three cases based on which case in $\text{update}_{Must}^\#$ applies:
 - A. First case in $\text{update}_{Must}^\#$ applies:
This is impossible as $b_i \neq b_n$.
 - B. Second case in $\text{update}_{Must}^\#$ applies:
Thus, $\widehat{S}(b_n) \leq \widehat{S}(b_i)$ and $\widehat{T}(b_i) = \widehat{S}(b_i)$.
We distinguish two further cases:
 - * $\widehat{S}(b_i) = \infty$:
Then, $\widehat{T}(b_i) = \infty$ and trivially $|CS_i(t)| \leq \widehat{T}(b_i)$.

¹²The constraints below account for the fact that t contains $n + 1$ accesses, where n is the number of accesses in s .

* $\widehat{S}(b_i) < \infty$:

As $\widehat{S}(b_n) \leq \widehat{S}(b_i)$, we also have $\widehat{S}(b_n) < \infty$.

As a consequence, due to the second constraint in (51), b_n must occur in s .

Let j be the index of the last occurrence of b_n in s .

If $i < j$ then $CS_i(t) = CS_i(s)$ and so $|CS_i(t)| = |CS_i(s)| \leq \widehat{S}(b_i) = \widehat{T}(b_i)$.

Otherwise, if $j < i$ then $CS_i(t) = CS_i(s) \cup \{b_n\} \subseteq CS_j(s)$.

As $|CS_j(s)| \leq \widehat{S}(b_n)$ and $\widehat{S}(b_n) \leq \widehat{S}(b_i) = \widehat{T}(b_i)$, we have $|CS_i(t)| \leq \widehat{T}(b_i)$.

C. Third or fourth case in $update_{Must}^\#$ applies:

Then $|CS_i(t)| \leq |CS_i(s)| + 1 \leq \widehat{S}(b_i) + 1 \leq \widehat{T}(b_i)$

■ Let $\widehat{S} \in C_{Must}^\#$ and $b \in \mathcal{B}$ be arbitrary.

Assume $classify_{Must}^\#(\widehat{S}, b)$ holds and thus $\widehat{S}(b) < k$. Let $s = c_0 \langle b_0, h_0 \rangle \dots c_n$ be an arbitrary trace in $\gamma_{Must}(\widehat{S})$. Let b_i be the last occurrence of b in the trace. If b does not occur in the trace, then (10) is satisfied by the second disjunct. Otherwise, $b_i \notin CS_{i+1}(s)$ and so $|CS_i(s)| \leq \widehat{S}(b)$. As $\widehat{S}(b) \leq k$ and $s \in \text{LRUCACHETRACES}$, we can apply Lemma 9 to the suffix $c_i \langle b_i, h_i \rangle \dots c_n$ to prove that $b \in c_n$. ◀

► **Theorem 29** (Soundness of Cooperative Update). *The function $coop\text{-}upd_{C\text{-}Must \times Must}$ is a cooperative update for $C\text{-}Must$ in the context of $Must$.*

Proof. We need to show that $coop\text{-}upd_{C\text{-}Must \times Must}$ satisfies (44), i.e.

$$\begin{aligned} \forall (\widehat{S}, \widehat{S}_{Must}) \in C_{C\text{-}Must}^\# \times C_{Must}^\#, b \in \mathcal{B} : \\ \{t.c \langle b, h \rangle c' \mid t.c \in \gamma_{C\text{-}Must}(\widehat{S}) \cap \gamma_{Must}(\widehat{S}_{Must}) \wedge h = \text{eff}_C^{LRU}(c, b) \wedge c' = \text{update}_C^{LRU}(c, b)\} \\ \subseteq \gamma_{C\text{-}Must}(coop\text{-}upd_{C\text{-}Must \times Must}(\widehat{S}, \widehat{S}_{Must}, b)) \quad (68) \end{aligned}$$

Let $(\widehat{S}, \widehat{S}_{Must}) \in C_{C\text{-}Must}^\# \times C_{Must}^\#$ and $b \in \mathcal{B}$ be arbitrary.

Pick an arbitrary $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C\text{-}Must}(\widehat{S}) \cap \gamma_{Must}(\widehat{S}_{Must})$. We have to show that $t = s.c \langle b_n, h_n \rangle c_{n+1}$ with $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$ is an element of $\gamma_{C\text{-}Must}(\widehat{T})$, with $\widehat{T} = \text{update}_{C\text{-}Must}^\#(coop\text{-}upd_{C\text{-}Must \times Must}(\widehat{S}, \widehat{S}_{Must}, b_n))$ for all $b_n \in \mathcal{B}$.

Because $s \in \gamma_{C\text{-}Must}(\widehat{S})$, $s \in \text{LRUCACHETRACES}$. From $h_n = \text{eff}_C^{LRU}(c_n, b_n)$ and $c_{n+1} = \text{update}_C^{LRU}(c_n, b_n)$, it follows that $t = s.c \langle b_n, h_n \rangle c_{n+1}$ is also in LRUCACHETRACES .

It remains to show that the second constraint in (31) holds, i.e.:

$$\forall i, 0 \leq i < n + 1 : b_i \in CS_{i+1}(t) \vee |CS_i(t)| \leq \widehat{S}(b_i).$$

Let i be arbitrary. We distinguish two cases based on its value:

1. $i = n$:

Observe that $CS_n(t) = \{b_n\}$.

The second case in the definition of $coop\text{-}upd_{C\text{-}Must \times Must}$ applies, and so $\widehat{T}(b_n) = 1$.

Thus, $|CS_n(t)| \leq \widehat{T}(b_n)$.

2. $0 \leq i < n$:

We have that $b_i \in CS_{i+1}(s) \vee |CS_i(s)| \leq \widehat{S}(b_i)$, because $s = c_0 \langle b_0, h_0 \rangle \dots c_n \in \gamma_{C\text{-}Must}(\widehat{S})$.

Clearly, $b_i \in CS_{i+1}(s)$ implies $b_i \in CS_{i+1}(t)$.

So the case where $b_i \notin CS_{i+1}(s)$ and thus $|CS_i(s)| \leq \widehat{S}(b_i)$ remains.

Then, the first case in the update may not apply, because $|CS_i(s)| > 0$ and thus $\widehat{S}(b_i) > 0$.

We distinguish two cases:

a. $b_i = b_n$:

Then $b_i \in CS_{i+1}(t) = CS_{i+1}(s) \cup \{b_n\} = CS_{i+1}(s) \cup \{b_i\}$.

b. $b_i \neq b_n$:

Because $b_i \neq b_n$, the second case in the update may not apply. So only the three final cases in $\text{coop-upd}_{C\text{-}Must \times Must}$ are possible.

Observe that $CS_i(t) = CS_i(s) \cup \{b_n\}$.

We distinguish two cases:

i. $b_i \in CS_i(s)$:

Then $|CS_i(t)| = |CS_i(s)| \leq \widehat{S}(b_i) \leq \widehat{T}(b_i)$ regardless of which of the three possible final cases in $\text{coop-upd}_{C\text{-}Must \times Must}$ applies to b_i .

ii. $b_n \notin CS_i(s)$:

Then $|CS_i(t)| = |CS_i(s)| + 1$.

We apply a case distinction based on the three possible final cases in $\text{coop-upd}_{C\text{-}Must \times Must}$:

A. If the fourth case in $\text{coop-upd}_{C\text{-}Must \times Must}$ applies to b_i , then

$$|CS_i(t)| = |CS_i(s)| + 1 \leq \widehat{S}(b_i) + 1 = \widehat{T}(b_i).$$

B. If the fifth case in the update applies, then $|CS_i(t)| \leq \infty = \widehat{T}(b_i)$.

C. It remains to show that $|CS_i(t)| \leq \widehat{T}(b_i)$ even if the third case in the update applies, which is where the update profits from the information provided by the must analysis.

If b_n does not occur in s , then by the definition of γ_{Must} , $\widehat{S}_{Must}(b_n) = \infty$, and so $\widehat{T}(b_n) = \widehat{S}(b_n) = \infty > |CS_i(t)|$.

Otherwise, let j be the index of the last occurrence of b_n in s .

As $b_n \notin CS_i(s)$, $j < i$, and $CS_j(s) \supseteq CS_i(s) \cup \{b_j\} = CS_i(s) \cup \{b_n\} = CS_i(t)$.

By the definition of γ_{Must} , $|CS_j(s)| \leq \widehat{S}_{Must}(b_n)$.

Under the assumption that the third case in the update applies, $\widehat{S}_{Must}(b_n) \leq \widehat{S}(b_i)$ and thus $|CS_i(t)| \leq |CS_j(s)| \leq \widehat{S}_{Must}(b_n) \leq \widehat{S}(b_i) = \widehat{T}(b_i)$. ◀