Cache Persistence Analysis: Finally Exact
40th IEEE Real-Time Systems Symposium (RTSS)

Gregory Stock  Sebastian Hahn  Jan Reineke
December 3–6, 2019
Saarland University, Germany

This presentation is available as a video:
https://dcloud.cs.uni-saarland.de/s/MkpMgBYoFt9q5Tx
https://youtu.be/PI9SauvflTo
Caches

- Caches contain memory blocks \( b \in \mathcal{B} \), stored in cache lines of same size.
- Caches are organized in sets, size of cache set called associativity \( k \).
- Cache replacement policy: least recently used (LRU).

---

\(^1\)W. l. o. g., we assume a fully-associative cache, i.e., with a single cache set. This is no real restriction as set-associative caches with \( n \) sets can be treated as \( n \) independent fully-associative caches.
Caches Are Important

\[ x = a + b; \]

\[
\begin{align*}
&\text{LOAD } r2, \_a \\
&\text{LOAD } r1, \_b \\
&\text{ADD } r3, r2, r1
\end{align*}
\]

Motorola PowerPC 755

<table>
<thead>
<tr>
<th>Exec. Time (Clock Cycles)</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>300</td>
</tr>
</tbody>
</table>
Control-Flow Graphs (CFG)

A control-flow graph $G = (V, E, i)$ is an abstraction of a program.

```c
int a, b, c;
for (int i = 0; i < 100; i++) {
if (a % 2) {
    b++;
} else {
    c--;
}
}```
A control-flow graph $G = (V, E, i)$ is an abstraction of a program.

```c
int a, b, c;
for (int i = 0; i < 100; i++) {
    if (a % 2) {
        b++;
    } else {
        c--;
    }
}
```
Control-Flow Graphs (CFG)

A control-flow graph $G = (V, E, i)$ is an abstraction of a program.

1 int a, b, c;
2
3 for (int i = 0; i < 100; i++) {
4     if (a % 2) {
5         b++;
6     } else {
7         c--;
8     }
9 }

The control-flow graph visualization shows the flow of control in the program:

- **Nodes** represent statements or branches.
- **Edges** indicate the flow of control from one statement to another.
- **Decision nodes** (e.g., if statements) have branches leading to other nodes.
- **Loop** structure indicated by the connection back to the start node.
Two Approaches

- classify accesses as “hits” or “misses”
- bound the total number of misses
Timing Analysis

Two Approaches

- classify accesses as “hits” or “misses”
- bound the total number of misses

Example

- cache of associativity $k = 4$
- no classification possible
  (no block either always misses or always hits the cache)
- at most three cache misses; max. one per block
Persistence Analysis
History of Cache Persistence Analysis

- 1994: Mueller • Arnold et al.
- 1997: White et al. • Ferdinand et al.
- 2000: Mueller
- 1999: Ferdinand et al.
- 2008: Ballabriga et al.
- 2011: Cullmann • Huynh et al.
- 2012: Nagar • Nagar et al.
- 2013: Cullmann • Cullmann

Dimensions:
- Instruction Cache
- Data Cache
- Direct-Mapped Cache
- Set-Associative Cache
- Analysis Precision
History of Cache Persistence Analysis

1994
Mueller • Arnold et al.

1997
White et al. • Ferdinand et al.

1999
Ferdinand et al.

2000
Mueller

2008
Ballabriga et al.

2011
Cullmann • Huynh et al.

2012
Nagar • Nagar et al.

2013
Cullmann

2019
Stock et al.

First Exact Analysis

Dimensions
- Instruction Cache
- Data Cache
- Direct-Mapped Cache
- Set-Associative Cache
- Analysis Precision
Persistence on a Trace

\[ \tau = f\ c\ e\ b\ a\ a\ b\ d\ c\ e\ f\ b\ e\ c\ a\ b\ f \]

**Definitions**

- **persistence**: all accesses to memory block result in *at most one* cache miss
Persistence on a Trace

\[ \tau = f \ c \ e \ b \ a \ a \ b \ d \ c \ e \ f \ b \ e \ c \ a \ b \ f \]

Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
Persistence on a Trace

\( \tau = f c e b a a b d c e f b e c a b f \)

Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
Persistence on a Trace

\[ \tau = f c e b a a b d c e f b e c a b f \]

Definitions

- **Persistence**: all accesses to memory block result in *at most one* cache miss
- LRU cache of **associativity** \( k = 4 \)
Persistence on a Trace

\[ \tau = f \ c \ e \ b \ a \ a \ b \ d \ c \ e \ f \ b \ e \ c \ a \ b \ f \]

\( \times \)

**Definitions**

- **persistence**: all accesses to memory block result in *at most one* cache miss
- **LRU cache of associativity** \( k = 4 \)
- **b’s conflict set**: set of all blocks accessed from last access to block \( b \) onward
Persistence on a Trace

\[ \tau = f^* c^* e^* b\ a\ \ a\ b\ d\ c\ e\ f\ b\ e\ c\ a\ b\ f \]

Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
- **LRU cache of associativity** \( k = 4 \)
- **b’s conflict set**: set of all blocks accessed from last access to block \( b \) onward
Persistence on a Trace

\[ \tau = \begin{array}{ccc} f & c & e \\ \uparrow & \uparrow & \uparrow \\ \emptyset & \{a, b\} \end{array} \]

Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
- LRU cache of **associativity** \( k = 4 \)
- **b’s conflict set**: set of all blocks accessed from last access to block \( b \) onward
Persistence on a Trace

\[ \tau = \begin{array}{ccc}
f & c & e \\
\uparrow & \uparrow & \checkmark \\
\emptyset & \{a, b\} \\
\end{array} \]

Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
- **LRU cache of associativity** \( k = 4 \)
- **b’s conflict set**: set of all blocks accessed from last access to block \( b \) onward
Persistence on a Trace

\[ \tau = f \ c \ e \]

\[ b \ a \ a \ b \ d \ c \ e \ f \ b \ e \ c \ a \ b \ f \]

Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
- **LRU cache of associativity** \( k = 4 \)
- **b’s conflict set**: set of all blocks accessed from last access to block \( b \) onward
Persistence on a Trace

\[ \tau = f \; c \; e \]

\[ \emptyset \rightarrow \{a, b\} \]

\[ b \; a \; a \; b \; d \; c \; e \; f \; b \; e \; c \; a \; b \; f \]

Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
- **LRU cache of associativity** \( k = 4 \)
- **b’s conflict set**: set of all blocks accessed from last access to block \( b \) onward
Persistence on a Trace

\[ \tau = f \ x \ c \ x \ e \ x \ b \ x \ a \ x \ a \ x \ b \ x \ d \ x \ c \ x \ e \ x \ f \ x \ b \ x \ e \ x \ c \ x \ a \ x \ b \ x \ f \]

**Definitions**

- **persistence**: all accesses to memory block result in *at most one* cache miss
- **LRU cache of associativity** $k = 4$
- **b’s conflict set**: set of all blocks accessed from last access to block b onward
Persistence on a Trace

\[ \tau = f \cdot c \cdot e \cdot b \cdot a \cdot a \cdot b \cdot d \cdot c \cdot e \cdot f \]\n
Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
- LRU cache of **associativity** \( k = 4 \)
- **b’s conflict set**: set of all blocks accessed from last access to block \( b \) onward
Persistence on a Trace

\[ \tau = f \ c \ e \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ b \ \ a \ \ a \ \ \ \ \ \ \ \ \ \ \ b \ d \ c \ e \ f \ \ \ \ \ \ \ b \ e \ c \ a \ \ \ \ \ \ \ b \ \ f \]

Definitions

- **persistence**: all accesses to memory block result in *at most one* cache miss
- LRU cache of **associativity** \( k = 4 \)
- **b's conflict set**: set of all blocks accessed from last access to block \( b \) onward
Persistence in a Program

Definition: Persistence in a Program
A block $b$ is persistent in a program if:

Example Traces

$\tau_1 = a \, c$
Persistence in a Program

Example Traces

\[ \tau_1 = a \ c \]

\[ b \leftrightarrow \emptyset \]
Persistence in a Program

Example Traces

- $\tau_1 = a \ c$  
  $b \mapsto \emptyset$
- $\tau_2 = a \ b \ d$  
  $b \mapsto \{b, d\}$

Definition: Persistence in a Program

A block $b$ is persistent in a program if:

$\forall \tau \in Traces: CS(\tau, b)$

associativity
Persistence in a Program

Example Traces

\[ \tau_1 = a \ c \quad b \mapsto \emptyset \]
\[ \tau_2 = a \ b \ d \quad b \mapsto \{b, d\} \]
\[ \tau_3 = a \ b \ b \ b \ b \ b \quad b \mapsto \{b\} \]
Definition: Persistence in a Program

A block $b$ is persistent in a program if:

- For all traces $\tau_2$ and for all $j$ such that $CS(\tau, b) = j$, the associativity holds.

### Example Traces

- $\tau_1 = a \ c$  
  $b \mapsto \emptyset$

- $\tau_2 = a \ b \ d$  
  $b \mapsto \{b, d\}$

- $\tau_3 = a \ b \ b \ b \ b \ b$  
  $b \mapsto \{b\}$

- $\tau_4 = a \ b \ d \ a \ c \ c \ b \ d$  
  $b \mapsto \{b, d\}$
**Persistence in a Program**

Example Traces

- $\tau_1 = a\, c$
  - $b \mapsto \emptyset$
- $\tau_2 = a\, b\, d$
  - $b \mapsto \{b, d\}$
- $\tau_3 = a\, b\, b\, b\, b\, b$
  - $b \mapsto \{b\}$
- $\tau_4 = a\, b\, d\, a\, c\, c\, b\, d$
  - $b \mapsto \{b, d\}$

Definition: Persistence in a Program

A block $b$ is persistent in a program if:

- $\forall \tau \in Traces:\ CS(\tau, b) \text{ is associative}$
Persistence in a Program

Example Traces

- \( \tau_1 = a \; c \)
- \( \tau_2 = a \; b \; d \)
- \( \tau_3 = a \; b \; b \; b \; b \)
- \( \tau_4 = a \; b \; d \; a \; c \; c \; b \; d \)

\( b \mapsto \emptyset \)
\( b \mapsto \{b, d\} \)
\( b \mapsto \{b\} \)
\( b \mapsto \{b, d\} \)

Definition: Persistence in a Program

A block \( b \) is persistent in a program if:

\[ \forall \tau \in \text{Traces} : |CS(\tau, b)| \leq \text{associativity } k \]
Persistence in a Program

Example Traces

- $\tau_1 = a \ c$  \quad $b \mapsto \emptyset$
- $\tau_2 = a \ b \ d$  \quad $b \mapsto \{b, d\}$
- $\tau_3 = a \ b \ b \ b \ b \ b$  \quad $b \mapsto \{b\}$
- $\tau_4 = a \ b \ d \ a \ c \ c \ b \ d$  \quad $b \mapsto \{b, d\}$
- $\cdots$

Definition: Persistence in a Program

A block $b$ is persistent in a program if:

$$\forall \tau \in Traces : |CS(\tau, b)| \leq \text{associativity } k$$
Exact Persistence Analysis
**Exact-CS: Example**

LRU Cache of associativity $k = 3$
LRU Cache of associativity $k = 3$
**Exact-CS: Example**

LRU Cache of associativity $k = 3$
Exact-CS: Example

LRU Cache of associativity $k = 3$
**Exact-CS: Example**

LRU Cache of associativity $k = 3$
LRU Cache of associativity $k = 3$
**Exact-CS**: Example

LRU Cache of associativity \( k = 3 \)

\[ a \mapsto \{ \emptyset \} \]

\[ a \mapsto \{ \{a\}, \{a, b\}, \{a, c\} \} \]

\[ a \mapsto \{ \{a, b\}, \{a, c\}, \{a, b, c\} \} \]

\[ d \mapsto \{ \{a, b\}, \{a, c\}, \{a, b, c\} \} \]
LRU Cache of associativity $k = 3$
**Exact-CS:** Example

LRU Cache of associativity $k = 3$
Exact-CS: Example

LRU Cache of associativity $k = 3$
**Exact-CS: Example**

LRA Cache of associativity $k = 3$

```
\begin{align*}
    & l_0 & a \mapsto \{\emptyset\} \\
    & l_1 & a \mapsto \{\{a\}, \{a, b\}, \{a, c\}, \{a, b, c\}\} \\
    & l_2 & a \mapsto \{\{a, b\}, \{a, c\}, \{a, b, c\}\} \\
    & l_3 & a \mapsto \{\{a, b\}, \{a, c\}, \{a, b, c\}, \\
    & & \{a, b, d\}, \{a, c, d\}, \{a, b, c, d\}\}
\end{align*}
```
Exact-CS: Example

LRU Cache of associativity $k = 3$
**Exact-CS: Example**

LRU Cache of associativity $k = 3$

Diagram:

- $l_0$ with set $\{\emptyset, \{a, b\}, \{a, c\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{a, b, c, d\}\}$
- $l_1$ with set $\{{}\} \\ \{a, b\} \\ \{a, c\} \\ \{a, b, c\}$
- $l_2$ with set $\{{}\} \\ \{a, b\} \\ \{a, c\} \\ \{a, b, c\}$
- $l_3$ with set $\{{}\} \\ \{a, b\} \\ \{a, c\} \\ \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{a, b, c, d\}$
Exact-CS: Example

LRU Cache of associativity $k = 3$
**Exact-CS: Example (Optimized)**

LRU Cache of associativity $k = 3$

Exploiting Monotonicity

- only *largest* conflict set relevant for classification
- safe to remove conflict sets *subsumed* by others
**Exact-CS: Example (Optimized)**

LRU Cache of associativity \( k = 3 \)

### Exploiting Monotonicity

- **only largest** conflict set relevant for classification
- safe to remove conflict sets **subsumed** by others

**Diagram:**

- \( l_0 \)
  - \( a \) \( \{ \emptyset, \{a, b\}, \{a, c\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{a, b, c, d\} \} \)
- \( l_1 \)
  - \( b \) \( c \)
  - \( a \) \( \{a\}, \{a, b\}, \{a, c\}, \{a, b, c\} \)
- \( l_2 \)
  - \( b \) \( c \)
  - \( d \)
  - \( a \) \( \{a, b\}, \{a, c\}, \{a, b, c\} \)
- \( l_3 \)
  - \( a \) \( \{a, b\}, \{a, c\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{a, b, c, d\} \)
Exact-CS: Example (Optimized)

LRU Cache of associativity $k = 3$

Exploiting Monotonicity

- only largest conflict set relevant for classification
- safe to remove conflict sets subsumed by others
**Exact-CS: Example (Optimized)**

LRU Cache of associativity $k = 3$

![Diagram showing the LRU Cache of associativity $k = 3$]

**Exploiting Monotonicity**
- only largest conflict set relevant for classification
- safe to remove conflict sets subsumed by others

**Cutoff at Associativity**
- check existence of single conflict set $C$ with $|C| > k$
- collapse all large conflict sets into single representative $B$
**Exact-CS: Example (Optimized)**

LRU Cache of associativity $k = 3$

- **Exploiting Monotonicity**
  - only largest conflict set relevant for classification
  - safe to remove conflict sets subsumed by others

- **Cutoff at Associativity**
  - check existence of single conflict set $C$ with $|C| > k$
  - collapse all large conflict sets into single representative $B$
**Exact-CS: Example (Optimized)**

LRU Cache of associativity $k = 3$

**Exploiting Monotonicity**
- only largest conflict set relevant for classification
- safe to remove conflict sets subsumed by others

**Cutoff at Associativity**
- check existence of single conflict set $C$ with $|C| > k$
- collapse all large conflict sets into single representative $B$
Efficient Implementation
Zero-suppressed BDDs: Sets of Combinations

**Binary Decision Diagram**

- **a**
- **b**
- **c**
- **d**

- **0**: $a \land \neg b \land \neg c \land \neg d$
- **1**: $\neg a \land b \land \neg c \land \neg d$

**Zero-suppressed BDD**

- **a**
- **b**

Sets of Conflict Sets

- $\{\{a\}, \{b\}\}$
- $\subseteq 2^{\{a,b,c,d\}}$

**Sets of Non-occurring Variables**

- Automatically suppressed
Evaluation
Gain in Analysis Precision?

Effect on Run Time and Memory?
Landscape of Cache Persistence Analyses

- Global-CS
  - C-May
    - Block-CS
      - C-Must
      - C-Must
    - C-Must
  - C-Must
    - C-Must
      - Block-CS
    - Must
  - C-Must
    - Must
      - Exact-CS

C-Must × Must × Block-CS
C-Must × Must × C-May
C-Must × Must × (Must)
Analysis Precision: WCET Ratios

Cache Configuration: 32 Cache Sets, 8 Ways, 16B Line Size, with Compiler Optimizations
Analysis Cost: *Global-CS vs. Exact-CS*

Cache Configuration: 32 Cache Sets, 8 Ways, 16B Line Size, with Compiler Optimizations

Running time on average 8.8 times higher; Memory 2.2 times higher (blue line)
Cache Configuration: 32 Cache Sets, 8 Ways, 16B Line Size, with Compiler Optimizations

Running time on average 2 times higher; Memory 1.6 times higher (blue line)
Abstract—Cache persistence analysis is an important part of worst-case execution time (WCET) analysis. It has been extensively studied in the past twenty years. Despite these efforts, all existing persistence analyses are approximative in the sense of worst-case execution time (WCET) analysis. It has been shown in the 1990s up until today [9]–[17]–[19] that all prior persistence analyses are approximative in the sense that they are not guaranteed to find all persistent memory blocks. In this paper, we close this gap by introducing the first exact persistence analysis for caches with least-recently-used (LRU) replacement, starting with Madhav’s [15]–[16] and Perlis’ [17] work in the 1990s up until today [10]–[17]. Neatly, all prior persistence analyses are approximative, in the sense that they are not guaranteed to find all persistent memory blocks of a program. In this paper, we close this gap by introducing the first exact persistence analysis. We develop this analysis via a sequence of three consecutive exact abstractions. The first abstraction is based on the observation that the persistence of a memory block can be determined by examining its possible conflict sets. We then collapse all large conflict sets into single representative conflict sets which are safe to remove. Only conflict sets that are subsumed by a large conflict set remain. The second abstraction is based on the fact that the analysis can be performed in an ad hoc manner. This is possible because the conflict sets of a program can be categorized into two groups. One group consists of conflict sets that may be too large to handle and that are, in practice, safe to remove. The second group consists of conflict sets that may be too large to handle and that must be transmitted verbatim to the WCET analysis. The third abstraction is based on the observation that the analysis can be performed in an ad hoc manner. This is possible because the conflict sets of a program can be categorized into two groups. One group consists of conflict sets that may be too large to handle and that are, in practice, safe to remove. The second group consists of conflict sets that may be too large to handle and that must be transmitted verbatim to the WCET analysis.

In this paper, we study persistence analysis, an instance of quantitative cache analysis. Persistence analyses consider all memory accesses in a program, or a fragment of a program such as a loop, that access the same memory block. A memory block is persistent if all memory accesses referring to this memory block may cumulatively result in at most one cache miss during any possible program execution. For a motivating example, consider Figure 1, which contains the control-flow graph of a simple program. The program consists of a loop, in which, in each loop iteration, either memory block $x$ or memory block $y$ is accessed. As neither block $x$, nor block $y$ has been accessed in any loop iteration, it is impossible to classify a cache analysis to classify any of the memory accesses in the program as a guaranteed cache hit; and so a WCET analysis would have to pessimistically account for misses upon all memory accesses. However, provided the cache is large enough to hold blocks $x$ and $y$, simultaneously, among all memory accesses to $x$ (and similarly to only the very first one in a cache miss). Both $x$ and $y$ are persistent, and WCET analysis can safely account for at most two misses in total. Given a program, the goal of persistence analysis is to determine which of the memory blocks accessed in the program are persistent. Persistence analysis has been extensively studied for caches with least-recently-used (LRU) replacement, starting with Madhav’s [15]–[16] and Perlis’ [17] work in the 1990s. Neatly, all prior persistence analyses are approximative, in the sense that they are not guaranteed to find all persistent memory blocks of a program.

1) **Classification of Memory Accesses:** To classify individual memory access as a cache hit or cache miss.
2) **Quantitative Cache Analysis:** To determine the number of cache misses resulting from a set of memory accesses. In this paper, we study persistence analysis, an instance of quantitative cache analysis. Persistence analysis considers all memory accesses in a program, or a fragment of a program such as a loop, that access the same memory block.

**Results:**
- In this paper, we close this gap by introducing the first exact persistence analysis for caches with least-recently-used (LRU) replacement, starting with Madhav’s [15]–[16] and Perlis’ [17] work in the 1990s. Neatly, all prior persistence analyses are approximative, in the sense that they are not guaranteed to find all persistent memory blocks of a program.

**Conclusion:** The presented analysis is competitive with state-of-the-art inexact analysis in terms of both memory consumption and analysis run time, which is unsurprisingly shortening as we show that persistence analysis is NP-complete. We also observe that while prior analysis are not exact in theory, they come close to being exact in practice. Modern processors can perform several arithmetic and logical operations in a single cycle. On the other hand, a single access to main memory can take hundreds of cycles. To bridge this performance gap, modern processors include one or multiple levels of cache. Caches are small but fast memories that store parts of main memory to quickly serve accesses to commonly used instructions and data. Memory accesses that “hit” the cache are served from the cache at a low latency, while accesses that “miss” the cache are served from main memory at a much higher latency. The execution time of a program thus highly depends on how effectively the processor’s cache hierarchy is implementing the high latency of main memory.

Real-time systems are systems that, in order to function correctly, have to perform their computations with limited amounts of wall-clock time. To verify a system’s real-time behavior, a major task is to bound each software component’s worst-case execution time (WCET). In the presence of caches, WCET analysis [1] has to account for the software’s cache behavior. Hence, it is important that each memory access could result in a cache miss would yield extremely pessimistic WCET bounds. Thus, static cache analysis [2] have been developed to simplify and precisely characterize a program’s cache behavior on a particular cache architecture. These can broadly be categorized into two groups.

- **Exact-Cache Example:** Given a program, the goal of persistence analysis is to determine which of the memory blocks accessed in the program are persistent. Persistence analysis has been extensively studied for caches with least-recently-used (LRU) replacement, starting with Madhav’s [15]–[16] and Perlis’ [17] work in the 1990s. Neatly, all prior persistence analyses are approximative, in the sense that they are not guaranteed to find all persistent memory blocks of a program. In this paper, we close this gap by introducing the first exact persistence analysis. We develop this analysis via a sequence of three consecutive exact abstractions. The first abstraction is based on the observation that the persistence of a memory block can be determined by examining its possible conflict sets. We then collapse all large conflict sets into single representative conflict sets which are safe to remove. Only conflict sets that are subsumed by a large conflict set remain. The second abstraction is based on the fact that the analysis can be performed in an ad hoc manner. This is possible because the conflict sets of a program can be categorized into two groups. One group consists of conflict sets that may be too large to handle and that are, in practice, safe to remove. The second group consists of conflict sets that may be too large to handle and that must be transmitted verbatim to the WCET analysis.

**Conclusion:** The presented analysis is competitive with state-of-the-art inexact analysis in terms of both memory consumption and analysis run time, which is unsurprisingly shortening as we show that persistence analysis is NP-complete. We also observe that while prior analysis are not exact in theory, they come close to being exact in practice.
References
