Impact of instruction cache replacement policy on the tightness of WCET estimation

Aurore Junier, Damien Hardy, Isabelle Puaut
University of Rennes / IRISA

Abstract

Cache memories have been introduced to decrease the access time to the information due to the increasing gap between fast micro-processors and relatively slower main memories. Thus, there is a need for considering caches when validating the temporal behavior of real-time systems, in particular when estimating tasks’ worst-case execution times (WCETs). In this paper, we use new theoretical results to improve a static instruction cache analysis method for set-associative instruction caches with a Pseudo-LRU and a random replacement policies. The proposed method is experimented on three medium-size benchmarks to quantify the impact of the replacement policy on the tightness of WCET estimation.

1. Introduction

Cache memories have been introduced to decrease the access time to the information due to the increasing gap between fast micro-processors and relatively slower main memories. Caches are very efficient at reducing average-case memory latencies for applications with good spatial and temporal locality. Architectures with caches are now commonly used in embedded real-time systems due to the increasing demand for computing power of many embedded applications.

In real-time systems it is crucial to prove that the execution of a task meets its deadline in all execution situations, including the worst-case. This proof needs an estimation of the worst-case execution times (WCETs) of any sequential task in the system. WCETs estimates have to be safe (larger than or equal to any possible execution time). Moreover, they have to be tight (as close as possible to the actual worst-case execution time) to correctly dimension the resources required by the system.

The presence of caches in real-time systems makes the estimation of both safe and tight WCET bounds difficult due to the dynamic behavior of caches. Safely estimating WCET on architectures with caches requires a knowledge of all possible cache contents in every execution context, and requires some knowledge of the cache replacement policy. Many static analysis methods have been proposed in order to produce a safe WCET estimation on architectures with caches [8, 2, 9, 6].

Regarding set-associative instruction caches with LRU (Least Recently Used) replacement policy, static cache analysis methods have been designed, based on two classes of approaches: static cache simulation [9, 6] or abstract interpretation [8, 2]. Although our work could also be integrated in static cache simulation methods, we will concentrate in the reminder of the paper on the methods based on abstract interpretation.

In such approaches, three different analyses are applied which use fixpoint computation to determine: if a memory block is always present in the cache (Must analysis), if a memory block may be present in the cache (May analysis), and if a memory block will not be evicted after it has been first loaded (Persistence analysis).

A cache categorisation (always-hit (AH), always-miss (AM), first-miss (FM) for access into loops and not classified (NC)) can then be assigned to every instruction based on the results of the three analyses to classify the worst-case behavior of the cache for a given instruction. AH is determined by the Must analysis, FM is determined by the Persistence analysis and AM, NC are differentiated by the May analysis.

This approach originally designed for LRU set-associative caches has been extended for Pseudo-LRU and Pseudo-Round-Robin cache replacement policies in [4] but without considering the results of the Persistence analysis. In addition, recently, theoretical results were proposed in [7] for cache using Pseudo-LRU replacement policy among others.

The contributions of this paper are twofold. On the one hand, we propose a persistence analysis for the Pseudo-LRU and Random replacement policies to improve the precision of the analysis for application with loops structures. On the other hand, we integrate the theoretical results proposed in [7] in our analysis tool in order to quantify the impact of the cache replacement policy on the tightness of cache analysis.

The rest of the paper is organized as follows. Section 2
first presents an overview of the static analysis proposed in [8] for a LRU replacement policy. Section 3 describes our improvements of [4] to implement the persistence analysis for other replacement policies than LRU and to use the new theoretical results proposed in [7]. Experimental results are given in Section 4. Finally, Section 5 concludes with a summary of the contributions of this paper, and gives directions for future work.

2. Static cache analysis for LRU caches

When a block is to be evicted from the cache, the Least Recently Used (LRU) replacement policy always selects the least recently used block. The method detailed in [8] for LRU caches is based on three separate fixpoint analyses applied on the program control flow graph:

- a Must analysis determines if a memory block is always present in the cache at a given point: if so, the block is classified always-hit;
- a May analysis determines if a memory block may be in the cache at a given point: if not, the block is classified always-miss. Otherwise, if not present at this point in the Must analysis and in the Persistence analysis the block is classified not classified;
- a Persistence analysis determines if a memory block will not be evicted after it has been loaded; the classification of such blocks is first-miss.

In order to capture all possible cache contents in every execution context, these three analyses use the concept of Abstract Cache States (ACS). Abstract cache states are computed at every basic block. Two functions on the abstract domain, named Update, and Join are defined for each analysis:

- Function Update is called for every memory reference on an ACS to compute the new ACS resulting from the memory reference. This function considers both the cache replacement policy and the semantics of the analysis.
- Function Join is used to merge two different abstract cache states in the case when a basic block has two predecessors in the control flow graph, like for example at the end of a conditional construct.

Figure 1 gives an example of the Join (1.a) and Update (1.b) functions for the Must analysis for a 2-ways set-associative cache with a LRU replacement policy. As in this context sets are independent from each other, only one set is depicted. A concept of age is associated with the cache block of the same set. The smaller the block age the more recent the access to the block. For the Must analysis, memory block a is stored only once in the ACS, with its maximum age. It means that its actual age at run-time will always be lower than or equal to its age in the ACS. The Join and Update functions are defined as follows for the Must analysis with LRU replacement policy (see Figure 1):

- The Join function applied to two ACS results in an ACS containing only the references present in the two input ACS and with their maximal age.
- The Update function performs an access to a memory reference c using an input abstract cache state ACS\textsubscript{in} (the abstract cache state before the memory access) and produces an output abstract cache state ACS\textsubscript{out} (the abstract cache state after the memory access). The Update function maps c onto its ACS\textsubscript{out} set with the younger age and increases the age of the other memory blocks present in the same set in ACS\textsubscript{in}. When the age of a memory block is higher than the number of ways, the memory block is evicted from ACS\textsubscript{out}.

For the other analyses (May and Persistence), the approach is similar and the Join function is defined as follows:

- May analysis: "union" of references present in the ACS and with their minimal age;
- Persistence analysis: "union" of references present in the ACS and with their maximal age.

For more details see [8].
3. Proposed static cache analysis

3.1. Analysis for pseudo-LRU replacement policy

To reduce the implementation cost of strict LRU, Pseudo-LRU replacement policies approximates the concept of age of strict LRU replacement. As Figure 2 shows us, the age is defined by a binary tree which nodes are 0 or 1 and leaves are cache lines in a set. A zero in the tree tells us that the under right tree is (approximately) younger than the left one and a one defines the contrary. Bits in the path binary tree are flipped every time a cache line is accessed.

![Pseudo-LRU replacement policy](image)

**Figure 2. Pseudo-LRU replacement policy**

**Theoretical results.** To avoid the modeling of the binary tree used in the PLRU replacement policy, as well as the internal data structures used by other cache replacement policies (e.g. MRU), some new theoretical results were recently proposed in [7]. Among others, two metrics are defined in [7]:

- **Minimum life span (mls).** This metric defines the minimal life duration of an element in a cache set for a given number of ways. After \( mls(nbWays) \) accesses to different pieces of information at the same set, it is guaranteed that all elements are still in the cache set. This metric is useful for the **Must** and **Persistence** analyses to compute the references that are guaranteed to be in the cache or persistent in the cache at every execution point.

- **Eviction distance (evict).** This metric defines the number of distinct accesses in a cache set, different from an access to a reference \( x \) to make sure that reference \( x \) is evicted from the cache. This metric is useful for the **May** analysis to determine the references that are guaranteed not to be in the cache at every execution point.

**Implementation** For the PLRU replacement policy, \( mls \) and \( evict \) are defined as follows [7]:

\[
mls(nbWays) = \log_2(nbWays) + 1
\]

\[
evict(nbWays) = \frac{nbWays}{2} \log_2(nbWays) + 1
\]

These results are used in the following manner:

- **Must** analysis for a set-associative cache with \( nbWays \) ways is achieved in the same manner as for LRU caches, except that ACS only comprises \( mls(nbWays) \leq nbWays \) instead of \( nbWays \). In other terms, **Must** analysis considers a cache with an associativity degree lower than the one of the actual cache, resulting in a loss of precision of the analysis as compared to the analysis of a LRU cache.

- **May** analysis for a set-associative cache with \( nbWays \) ways is achieved in the same manner as for LRU cache, except that ACS have \( evict(nbWays) \geq nbWays \) instead of \( nbWays \). For a given reference \( x \), more than \( nbWays \) different accesses after \( x \) are required to make sure that \( x \) gets evicted from the cache.

- **Persistence** analysis uses the same principles as for a LRU cache, except that ACS of size \( mls(nbWays) \leq nbWays \) are used.

3.2. Analysis for random replacement policy

For a random replacement policy, \( mls(nbWays) = 1 \) and \( evict(nbWays) = \infty \). The analysis is achieved as before, except that the **May** analysis is now useless since there is no absolute guarantee that a cache line will be evicted from the cache.

4. Experimental results

In this section, we quantify the loss of precision resulting from the use of replacement policies other than LRU. Experimental conditions are given in paragraph 4.1. Quantitative results are given in section 4.2.

4.1. Experimental setup

**Cache analysis and WCET estimation.** The experiments were conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 with flag O0. The WCETs of tasks are computed by the Heptane\(^2\) timing analyzer [1], more precisely its Implicit Path Enumeration Technique (IPET\(^3\)). The analysis is context sensitive (function are analyzed in each different calling context).

To separate the effect of the caches from those of the parts of the processor micro-architecture, WCET estimation only takes into account the contribution of caches to the WCET. The effects of other architectural features are not considered. In particular, we do not take into account timing anomalies caused by interactions between caches and pipelines, as defined in [5]. The cache classification not-classified is thus

\(^2\)Heptane is an open-source static WCET analysis tool available at http://www.irisa.fr/aces/software/software.html.

\(^3\)So-called IPET methods estimate WCET by solving linear equations generated from the program control flow graph [10].
assumed to have the same worst-case behavior as always-miss during the WCET computation in our experiments. The cache analysis starts with an empty cache state.

The experiments were conducted on three medium-size benchmark named adpcm, jfdctint and minver maintained by Mälardalen WCET research group\(^4\) and by considering a 2KB cache with 32B lines.

4.2. Results

The results of the cache analysis for the LRU, PLRU and Random replacement policy are given in figure 3. They are expressed in terms of percentage of cache hits detected by the analysis (the higher the percentage the tighter the analysis).

As expected, the LRU replacement policy can be analyzed more tightly than the PLRU replacement policy, which itself can be analyzed more tightly than the Random replacement policy. However, when the loops in the applications are small enough to fit entirely in mls(\text{nWays}) ways in the cache, there are no difference between the analyses. One can also notice than the higher the cache associativity, the larger the difference between the tightness of the analyses. This is explained by the larger difference between mls(\text{nWays}) when \text{nWays} gets larger.

5. Conclusion

In this paper, we have implemented a cache analysis method using a part of the theoretical bounds proposed in [7]. We have quantified the pessimism resulting from non-LRU replacement policies and discussed the circumstances in which the loss of precision is significant. In our future work, we will extend our evaluation to the other replacement policies studied in [7]. Moreover, we wish to study the impact of the cache replacement policy on the tightness of multi-level cache analysis [3], specially on multi-core platforms.

---

\(^4\)http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

### References


