

Model Checking of Cache for WCET Analysis Refinement

Valentin Touzeau, Claire Maïza, David Monniaux

▶ **To cite this version:**

Valentin Touzeau, Claire Maïza, David Monniaux. Model Checking of Cache for WCET Analysis Refinement. 10th Junior Researcher Workshop on Real-Time Computing, Oct 2016, Brest, France. hal-01447930v2

HAL Id: hal-01447930
https://hal.science/hal-01447930v2

Submitted on 5 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model Checking of Cache for WCET Analysis Refinement *

Valentin Touzeau
Univ. Grenoble Alpes
CNRS, VERIMAG, F-38000
Grenoble, France
Valentin.Touzeau@imag.fr

Claire Maïza
Univ. Grenoble Alpes
CNRS, VERIMAG, F-38000
Grenoble, France
Claire.Maiza@imag.fr

David Monniaux
Univ. Grenoble Alpes
CNRS, VERIMAG, F-38000
Grenoble, France
David.Monniaux@imag.fr

ABSTRACT

On real-time systems running under timing constraints, scheduling can be performed when one is aware of the worst case execution time (WCET) of tasks. Usually, the WCET of a task is unknown and schedulers make use of safe over-approximations given by static WCET analysis. To reduce the over-approximation, WCET analysis has to gain information about the underlying hardware behavior, such as pipelines and caches. In this paper, we focus on the cache analysis, which classifies memory accesses as hits/misses according to the set of possible cache states. We propose to refine the results of classical cache analysis using a model checker, introducing a new cache model for the least recently used (LRU) policy.

Keywords

Worst Case Execution Time, Cache Analysis, Model Checking, Least Recently Used Cache

1. INTRODUCTION

On critical systems, one should be able to guarantee that each task will meet its deadline. This strong constraint can be satisfied when the scheduler has bounds on every tasks' execution time. The aim of a WCET analysis is to compute such safe bounds statically. In order to provide a satisfiable bound, the WCET analysis needs to model the execution of instructions at the hardware level. However, to avoid the huge latency of main memory access, one can copy parts of the main memory into small but fast memories called caches. In order to retrieve precise bounds on the execution time of instructions, it is thus mandatory to know which instructions are in the cache and which are not. In this paper we focus on instruction caches, ie. we aim at knowing whether instructions of the program are in the cache when they are executed.

For efficiency reasons, the main memory is partitioned into fixed size blocks. To avoid repeated accesses to the same block, they are temporary copied into the cache when requested by the CPU. This way, they can be retrieved faster on the next access, without requesting the main memory again. Moreover, to speed up the retrieval of blocks from the cache, caches are usually partitioned into sets of equal sizes. A memory block can only be stored in one set that is uniquely determined by the address of the block. Thus, when searching a block in the cache, it is looked for in only one set. Since the main memory is bigger than the cache, it may happen that a set is already full when trying to store a new block in it. In this case, one block of the set has to be evicted in order to store the new one. This choice does not depend on the content of the other sets and is done according to the cache replacement policy. In our case, we focus on the Least Recently Used (LRU) policy (for other policies, refer to [5]). A cache set using the LRU policy can be represented as a queue containing blocks ordered from the most recently accessed (or used) to the least recently accessed. When the CPU requests a block that is not in the cache (cache miss), it is stored at the beginning of the queue (it becomes the most recently used block) and the last block (the least recently used) is evicted. On the other hand, when the requested block is already in the cache, it is moved to the beginning of the queue, delaying its eviction. The position of a block in the queue is called the age of the block: the youngest block is the most recently used and the oldest is the least recently used.

The aim of a cache analysis is to provide a classification of memory accesses as "cache hit", "cache miss" or "unknown" (not always a hit, and not always a miss) to be used as part of the WCET analysis. This classification is usually established by abstract interpretations called "May Analysis" and "Must Analysis" that respectively compute a lower and upper bound of every block's age. For more details about these analysis, refer to [3]. *Must* analysis is used to predict the cache hits (if a block must be in the cache when accessing it, access is a hit), whereas *may* analysis is used to predict the misses (if a block may not be in the cache when accessing it, access is a miss). However, if a block is in the *may* cache but not in the *must* cache, it is classified as unknown. This can happen because this access is sometimes a miss and sometimes a hit, or because the abstract interpretation is too coarse. An example is given on Figure 1. Program states (basic blocks) are on the left, whereas abstract cache states (*may* and *must*) at the exit of basic blocks are on the right. For simplicity, every basic block is stored in exactly one memory block. For example, at the exit of block 5, the minimum age of blocks a, b, c and d computed by the *may* analysis are respectively 1, 0, 2 and 1. At block 6, a is accessed and is in the cache (because there are only 4 different blocks, and they all fit together in the cache), thus it should be classified as a hit. However, it is classified as "unknown" by the *may* and *must* analysis because of an over-approximation performed by

*This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR".

the *must* analysis. Indeed, at entry of block 5, the *must* cache is $[\bot, \bot, \bot, a]$ because a is the only block that must be accessed, and b, c and d may be accessed since the last access to a. An other possibility to classify memory access

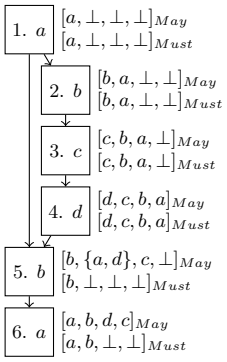


Figure 1: Example of access classified as "unknown"

as hit or miss is to use a model checker. Both the program and the cache are encoded as a transition system. Then, the question of the existence in the cache of a given block at a given program point is encoded as a logical formula. Both the formula to check and the transition system are provided to the model checker, that classifies the block as "in the cache", "out of the cache" or "unknown". Since the model deals with every reachable program/cache states separately, model checking is usually more precise than abstract interpretation. However, it is also much slower and often requires more memory during the analysis.

To avoid the precision loss of abstract interpretation without performing a heavy analysis using a model checker, we propose to mix both analysis. We first perform the classical *may/must* analysis by abstract interpretation, and then refine the classification using a model checker. Thus, we only use the model checker to classify blocks that were classified as "unknown" by the abstract interpretation. Moreover, we introduce a new abstract model that can be used by the model checker to efficiently represent LRU cache states.

2. WCET ANALYSIS

This section gives some basic notions about WCET analysis and explain the link with cache analysis.

Usually, WCET analysis are performed by following steps: First, a control flow graph is retrieved from the binary code under analysis by grouping instructions into basic blocks (sequences of instructions with only one entry point and one exit point). Then, the WCET of the program is computed by bounding the execution time of every basic block and finding the "longest" path inside the CFG.

The computation of basic blocks execution time is done by micro-architectural analysis that models pipelines and caches. Independently, several other analysis like value analysis and loop bound analysis are performed to provide information about the semantics to the WCET analyzer.

At hardware level, the uncertainty about execution times of instructions comes from pipelines (which can start executing an instruction before the previous one is finished) and caches (which avoid costly main memory accesses). The aim of a cache analysis is to classify memory accesses as cache hit or cache miss. Since an access to the main memory can be 100 times slower than an access to a cache, it is mandatory to classify memory access as hit/miss in order to provide accurate bounds on WCET estimations. Moreover, when a memory access is not classified as a hit or a miss, the WCET analysis must treat the both cases [2]. Thus, it can make the analysis slower and increase the pessimism.

To avoid this precision loss, we aim at refining the classification of unknown block using a model checker.

3. RELATED WORK

Some previous works use a model checker for performing WCET analysis of programs. The approach of Lv, Yi, Guan and Yu [6] is the following. Using abstract interpretation, they classify memory access as "cache misses", "cache hits" or "unknown". Every memory access is translated in a timed automaton that describe the access to the memory bus by introducing some delay. Finally, these automata are connected together according to the CFG and the model checker explores the transition system, computing a WCET estimation. This approach allows them to perform WCET analysis of multicore systems. Although they are using a model checker in a WCET analysis, our approach is different and complementary. Indeed, they are using the model checker to refine the timing analysis itself, and not the classification of memory accesses as we do. Therefore, our refinement of the cache content can be added to the first step of their analysis to retrieve better bounds. The work of Chattopadhyay and Roychoudhury [1] is closer to our approach. They use the model checker to analyze behavior of caches shared by several cores. Moreover, by instrumenting the program at source code level they can take the program semantics into account and do not treat infeasible path, make the analysis more precise. Since they are performing a *may/must* analysis as a first step, we believe our analysis can be used to refine this first step, before taking shared caches into account.

4. OUR ABSTRACT MODEL

To perform the cache analysis using a model checker, we have to provide models both for the program and for the cache. We use these models to answer the following questions: "At a given program point, is a given block in the cache whatever the path to reach this point is ?" (classify as hit) and "Is the given block never in the cache at the given program point, whatever the path used to reach this point is ?" (classify as miss).

To model the cache, we use an abstraction of the real cache state to avoid the state space explosion problem one can meet when using a model checker. In the following formal description of our model, we adapt the notations from Jan Reineke's PhD [7]:

DEFINITION 1. *Cache size*

$k \in \mathbb{N}$ *is the size of the cache set (in blocks)*

DEFINITION 2. *Set of memory blocks*

M *represents the set of memory blocks.*
$M_\bot = M \cup \{\bot\}$ *includes the empty line.*

DEFINITION 3. *Set of Cache States*

$C^{LRU} \subset M_{\perp}^{k}$ *symbolizes the set of reachable cache states*

$[b_1, b_2, \ldots, b_k] \in C^{LRU}$ *represents a reachable state*

b_1 *is the least recently used block*

In addition of these notation, to define our abstract model, we introduce the following notations: $\mathcal{A}$, represents the set of abstract cache states, $\varepsilon_a \in \mathcal{A}$, represents cache states that does not contain a, and $[S]_a \in \mathcal{A}$, represents cache states that contains a and some other blocks younger than a (forming the set S), where $a \in M$ is a memory block.

DEFINITION 4. *Set of Abstract Cache States*

$\mathcal{A} = \mathcal{P}(C^{LRU})$ *is the power set of reachable cache states.*

DEFINITION 5. *Abstract Cache States*

$$\varepsilon_a = \left\{ [b_1, ..., b_k] \in C^{LRU}, \forall i \in [\![1, k]\!], b_i \neq a \right\} \in \mathcal{A}$$

$$[S]_a = \Big\{ [b_1, ..., b_k] \in C^{LRU}, \exists i \in [\![1, k]\!], b_i = a \wedge (b_j \in S \Leftrightarrow j < i) \Big\} \in \mathcal{A}$$

The idea behind the abstract model we define below is to track only one block (noted a). Indeed, to know whether a block is in a LRU cache, you only have to count the number of accesses made to pairwise different blocks since the last access to it. In other words, you do not have to know the age of others blocks, you are only interested in knowing if they are younger than the block you are tracking. Therefore, we group together cache states that have the same set of blocks younger than the block we want to track.

To every cache state p, we associate an abstract state $\alpha_a(p)$ which consists of the set of values younger than a in the cache or a special value in the case where a is not in the cache.

DEFINITION 6. *Abstraction of Cache States*

$$\alpha_a : C^{LRU} \to \mathcal{A}$$

$$\alpha_a\left([b_1, ..., b_k]\right) = \begin{cases} \varepsilon_a & \text{if } \forall i \in [\![1, k]\!], b_i \neq a \\ [\{b_1, ..., b_{i-1}\}]_a & \text{if } \exists i \in [\![1, k]\!], b_i = a \end{cases}$$

For example, when tracking block a, the abstract cache state associated to the exit of block 1 of Figure 1 is $[\{\}]_a$, symbolizing that a is the least recently used block at this point (the set of younger blocks is empty). At the exit of block 4, the abstract cache state is $[\{b, c, d\}]_a$.

Additionally, we define the partial function $update^{a}_{LRU(k)}$, which models the effect of accessing a block on an abstract state. When the abstract cache state does not contain a (i.e. is equal to ε_a), it remains unchanged until an access to a is made. When a is accessed, every new block access appears into the set S. When the cardinal of S reaches $k-1$ (a is the least recently used block), a new access to a different block evicts a (and new abstract cache state is reset to ε_a). If an access to a is done in the meantime, the set S of younger block is reset to $\{\}$.

DEFINITION 7. *Abstract State Update*

$$update^{a}_{LRU(k)} : \mathcal{A} \times M \to \mathcal{A}$$

$$update^{a}_{LRU(k)}(\varepsilon_a, c) = \begin{cases} [\{\}]_a & \text{if } a = c \\ \varepsilon_a & \text{if } a \neq c \end{cases}$$

$$update^{a}_{LRU(k)}([S]_a, c) = \begin{cases} [\{\}]_a & \text{if } c = a \\ [S]_a & \text{if } c \neq a \text{ and } c \in S \\ [S \cup \{c\}]_a & \text{if } c \neq a \text{ and } c \notin S \text{ and } |S| < k-1 \\ \varepsilon_a & \text{if } c \neq a \text{ and } c \notin S \text{ and } |S| = k-1 \end{cases}$$

Considering the example of Figure 1, the model checker associates two different abstract states to block 5 depending on the incoming flow from block 1 or block 4. These states are respectively $[\{\}]_a$ and $[\{b, c, d\}]_a$. Thus, applying the update function for treating the access made to b in block 5, we obtain $[\{b\}]_a$ and $[\{b, c, d\}]_a$. Therefore, we know that a is not evicted from the cache by block 5 and access made to a in block 6 is not classified as unknown anymore but as a hit.

The second part of our model is the model of the program. Since we focus on instruction caches, the model we use for the program is a graph obtained from the CFG by splitting basic blocks (when needed) into blocks of the size of a memory block. Thus, a path in the model represents the sequence of memory access that the instruction cache handles during the execution of the program. However, because we only track one memory block at a time, it is also possible to simplify the control flow graph according to this block. Indeed, one can slice the CFG according to the cache set associated to the block we want to track, removing every memory access to an other cache set. Moreover, we can remove from the obtained graph every node that is not an access to a and that does not contribute to a eviction. Thus, it appears that every node that does not contain a in their entry *may* cache can be removed.

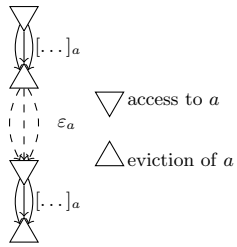


Figure 2: Simplifying CFG according to access to a

This simplification of the CFG is illustrated on Figure 2. Plain arrows represent program flow potentially manipulating block a, whereas dashed arrows represent flow that does not and that can be simplified in only one arrow. At some point after an access to block a, we can be sure that a is not in the cache anymore. Therefore, it is possible to remove all nodes (dashed arrows) from this point until the next access to a.

5. IMPLEMENTATION / EXPERIMENTS

This section describe the prototype we build and the experiments we made to valid our proof of concept. The workflow of our analysis is illustrated on Figure 3.

Our implementation does not use directly the binary code to analyze but runs on the LLVM bytecode representation of it. We first build the CFG of the program from the bytecode using the LLVM framework. Since the

Program	Size	4 ways		8 ways		16 ways	
		Un	Nc	Un	Nc	Un	Nc
recursion	26	34.6%	11.1%	53.8%	7.1%	53.8%	21.4%
fac	26	34.6%	11.1%	46.1%	8.3%	46.1%	41.6%
binarysearch	48	12.5%	0%	56.2%	29.6%	52.0%	12.0%
prime	57	10.5%	0%	29.8%	35.2%	57.8%	18.1%
insertsort	58	23.7%	28.5%	28.8%	11.7%	55.9%	9.0%
bsort	62	30.6%	57.8%	53.2%	6.0%	62.9%	5.1%
duff	64	10.9%	0%	37.5%	12.5%	37.5%	12.5%
countnegative	65	21.5%	21.4%	43.0%	21.4%	52.3%	20.5%
st	137	14.5%	30.0%	43.7%	13.3%	69.3%	5.2%
ludcmp	179	11.1%	5.0%	39.6%	15.4%	67.5%	4.1%
minver	265	20.7%	29.0%	44.1%	12.8%	63.0%	10.7%
statemate	582	7.5%	2.2%	7.9%	4.3%	8.2%	2.0%

Table 1: Precision of *May/Must* analysis and Model Checker

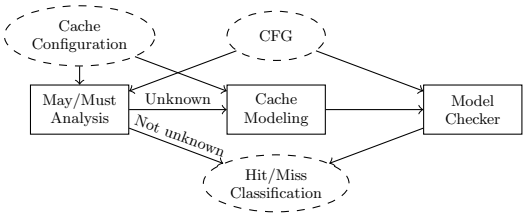


Figure 3: Workflow of our prototype

LLVM bytecode does not affect any address to instructions, we have to provide a mapping of instructions to the main memory. For our prototype, we assume that every instruction has the same size in memory. Thus, memory blocks contain a fixed number of instructions and can be obtain by splitting basic blocks of the CFG into fixed size blocks. Using this mapping and the CFG, our prototype performs a *may/must* analysis of the program. For every block access classified as unknown, we build an abstract model of the cache and provide it to the model checker together with the CFG (simplified as explain above). It would be possible to use real addresses from binary code and a correspondence between LLVM bytecode and binary code, as done in [4], but this requires significant engineering and falls outside of the scope of this experiment.

We experiment our prototype with benchmarks of the TacleBench[1]. Table 1 contains the results of our experiments. Size of program is given in number of memory block. We run our experiments with caches of only one cache set, with different sizes: 4, 8 or 16 ways. For every experiment, we measure both the amount of accesses classified as "unknown" by the *may/must* analysis (column "Un") and the amount of accesses newly classified as "always in the cache" or "always out of the cache" among the accesses left "unknown" by the *may/must* analysis (column Nc). During these experiments, our analysis classifies up to 57.8% of the accesses left unclassified by the abstract interpretation analysis.

6. CONCLUSION

We proposed to refine classical cache analysis by using a model checker. To avoid the common problem of state space explosion meet when dealing with model checking, we introduce a new abstract cache model. This model allows us to compute the exact age of a memory block along an execution path of the program. Thus, we can select the memory block we want to refine. Moreover, it allows us to simplify the program model too, by removing some nodes useless to the refinement. Finally, we implement a prototype and test it on a benchmark. Our experiments shows that our approach is able to refine up to 60% of the memory access classified as unknown by the abstract interpretation.

Our prototype runs on LLVM bytecode, and use an unrealistic memory mapping. As future work, we aim at implementing an analyzer that runs on the binary code. To finally validate our approach, it is also possible to compare the performance of our analysis to other analysis refining *may/must* analysis, like persistence analysis or analysis performing virtual inlining and unrolling.

[1] http://www.tacle.eu/index.php/activities/taclebench

7. REFERENCES

[1] S. Chattopadhyay and A. Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *RTSS 2011*. IEEE Computer Society, 2011.

[2] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. Predictability considerations in the design of multi-core embedded systems.

[3] C. Ferdinand. *Cache behavior prediction for real-time systems*. Pirrot, 1997.

[4] J. Henry, M. Asavoae, D. Monniaux, and C. Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In Y. Zhang and P. Kulkarni, editors, *SIGPLAN/SIGBED, LCTES '14*. ACM, 2014.

[5] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi. A survey on static cache analysis for real-time systems. *LITES*, 2016.

[6] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS2010*, pages 339–349. IEEE Computer Society, 2010.

[7] J. Reineke. *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, Saarland University, 2009.