UGA Université Grenoble Alpes

# THÈSE

Pour obtenir le grade de

## DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par


## Valentin TOUZEAU

Thèse dirigée par **David MONNIAUX**, Université Grenoble Alpes
et codirigée par **Claire MAIZA**, MCF, Grenoble INP

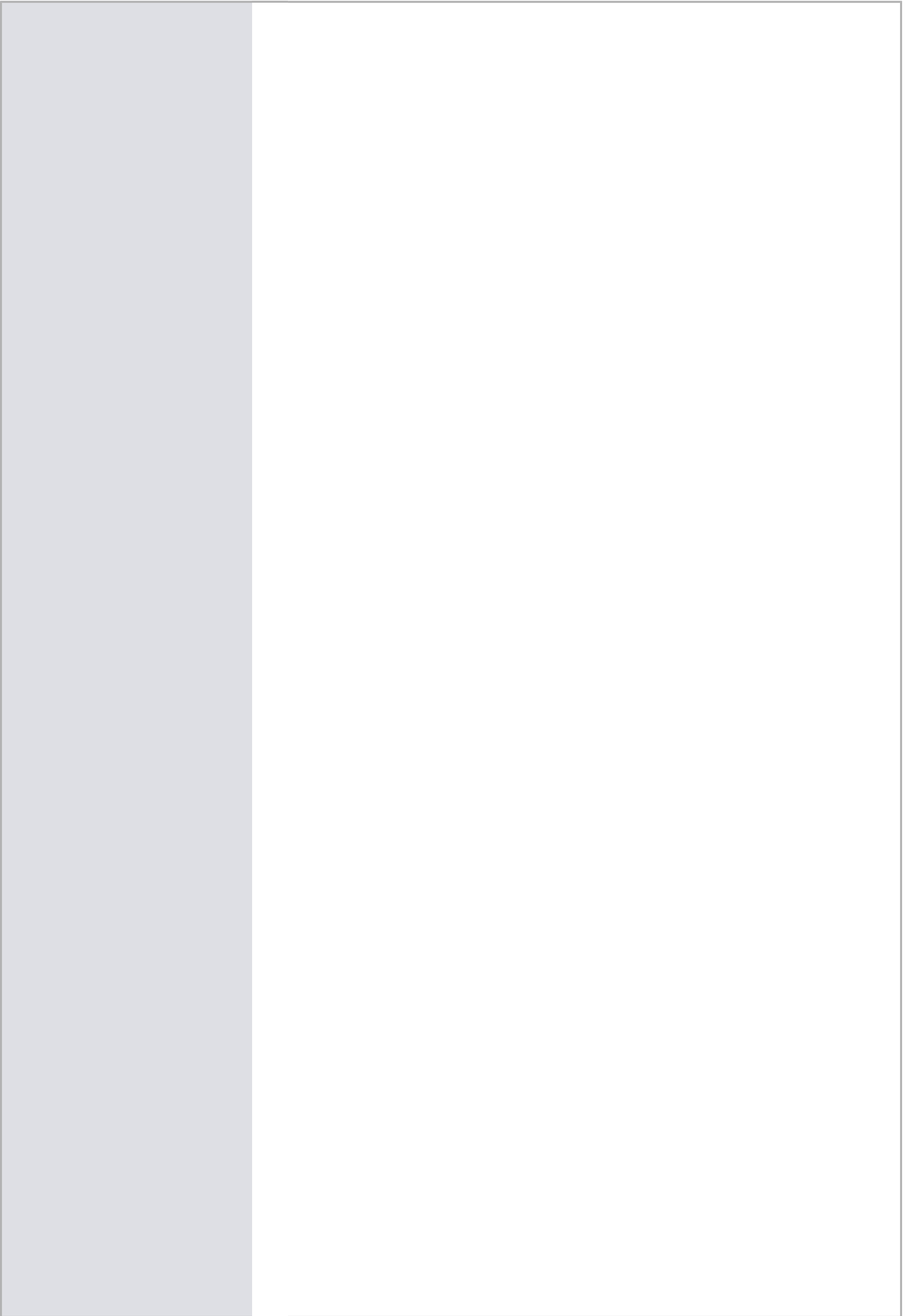
préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**


# Analyse statique de caches LRU : complexité, analyse optimale, et applications au calcul de pire temps d'exécution et à la sécurité

# Static analysis of least recently used caches: complexity, optimal analysis, and applications to worst-case execution time and security

Thèse soutenue publiquement le **8 octobre 2019**,
devant le jury composé de :

**Monsieur DAVID MONNIAUX**
DIRECTEUR DE RECHERCHE, CNRS DELEGATION ALPES, Directeur de thèse
**Madame CLAIRE MAÏZA**
MAITRE DE CONFERENCES, GRENOBLE INP, Examinateur
**Monsieur KENNETH MCMILLAN**
CHERCHEUR, LAB RECHERCHE MICROSOFT A REDMOND - USA, Rapporteur
**Monsieur BJÖRN LISPER**
PROFESSEUR, UNIVERSITE DE MÄLARDALEN (MDH) - SUEDE, Rapporteur
**Monsieur HUGUES CASSE**
MAITRE DE CONFERENCES, UNIVERSITE TOULOUSE-III-PAUL-SABATIER, Examinateur
**Monsieur JAN REINEKE**
PROFESSEUR, UNIVERSITE DE LA SARRE - ALLEMAGNE, Examinateur
**Monsieur FREDERIC PETROT**
PROFESSEUR, GRENOBLE INP, Président
**Monsieur SEBASTIEN FAUCOU**
MAITRE DE CONFERENCES, UNIVERSITE DE NANTES, Examinateur



**Abstract**

The certification of real-time safety critical programs requires bounding their execution time. Due to the high impact of cache memories on memory access latency, modern Worst-Case Execution Time estimation tools include a cache analysis. The aim of this analysis is to statically predict if memory accesses result in a cache hit or a cache miss. This problem is undecidable in general, thus usual cache analyses perform some abstractions that lead to precision loss. One common assumption made to remove the source of undecidability is that all execution paths in the program are feasible. Making this hypothesis is reasonable because the safety of the analysis is preserved when adding spurious paths to the program model. However, classifying memory accesses as cache hits or misses is still hard in practice under this assumption, and efficient cache analysis usually involve additional approximations, again leading to precision loss. This thesis investigates the possibility of performing an optimally precise cache analysis under the common assumption that all execution paths in the program are feasible.

We formally define the problems of classifying accesses as hits and misses, and prove that they are NP-hard or PSPACE-hard for common replacement policies (LRU, FIFO, NMRU and PLRU). However, if these theoretical complexity results legitimate the use of additional abstraction, they do not preclude the existence of algorithms efficient in practice on industrial workloads.

Because of the abstractions performed for efficiency reasons, cache analyses can usually classify accesses as *Unknown* in addition to *Always-Hit* (Must analysis) or *Always-Miss* (May analysis). Accesses classified as *Unknown* can lead to both a hit or a miss, depending on the program execution path followed. However, it can also be that they belong to one of the *Always-Hit* or *Always-Miss* category and that the cache analysis failed to classify them correctly because of a coarse approximation. We thus designed a new analysis for LRU instruction that is able to soundly classify some accesses into a new category, called *Definitely Unknown*, that represents accesses that can lead to both a hit or a miss. For those accesses, one knows for sure that their classification does not result from a coarse approximation but is a consequence of the program structure and cache configuration. By doing so, we also reduce the set of accesses that are candidate for a refined classification using more powerful and more costly analyses.

Our main contribution is an analysis that can perform an optimally precise analysis of LRU instruction caches. We use a method called *block focusing* that allows an analysis to scale by only analyzing one cache block at a time. We thus take advantage of the low number of candidates for refinement left by our *Definitely Unknown* analysis. This analysis produces an optimal classification of memory accesses at a reasonable cost (a few times the cost of the usual May and Must analyses).

We evaluate the impact of our precise cache analysis on the pipeline analysis. Indeed, when the cache analysis is not able to classify an access as *Always-Hit* or *Always-Miss*, the pipeline analysis must consider both cases. By providing a more precise memory access classification, we thus prune the state space explored by the pipeline analysis and hence the WCET analysis time.

Aside from this application of precise cache analysis to WCET estimation, we investigate the possibility of using the *Definitely Unknown* analysis in the domain of security. Indeed, caches can be used as side-channel to extract some sensitive data from a program execution, and we propose a variation of our *Definitely Unknown* analysis to help a developer finding the source of some information leakage.

**Résumé**

Dans le cadre des systèmes critiques, la certification de programmes temps-réel nécessite de borner leur temps d'exécution. Les mémoires caches impactant fortement la latence des accès mémoires, les outils de calcul de pire temps d'exécution incluent des analyses de cache. Ces analyses visent à prédire statiquement si ces accès aboutissent à des cache-hits ou des cache-miss. Ce problème étant indécidable en général, les analyses de caches emploient des abstractions pouvant mener à des pertes de précision. Une hypothèse habituelle pour rendre le problème décidable consiste à supposer que toutes les exécutions du programme sont réalisables. Cette hypothèse est raisonnable car elle ne met pas en cause la validité de l'analyse : tous les véritables chemins d'exécutions du programme sont couverts par l'analyse. Néanmoins, la classification des accès mémoires reste difficile en pratique malgré cette hypothèse, et les analyses de cache efficaces utilisent des approximations supplémentaires. Cette thèse s'intéresse à la possibilité de réaliser des analyses de cache de précision optimale sous l'hypothèse que tous les chemins sont faisables.

Les problèmes de classification d'accès mémoires en hits et miss y sont définis formellement et nous prouvons qu'ils sont NP-difficiles, voire PSPACE-difficiles, pour les politiques de remplacement usuelles (LRU, FIFO, NMRU et PLRU). Toutefois, si ces résultats théoriques justifient l'utilisation d'abstractions supplémentaires, ils n'excluent pas l'existence d'un algorithme efficace en pratique pour des instances courantes dans l'industrie.

Les abstractions usuelles ne permettent pas, en général, de classifier tous les accès mémoires en *Always-Hit* et *Always-Miss*. Certains sont alors classifiés *Unknown* par l'analyse de cache, et peuvent aboutir à des cache-hits comme à des cache-miss selon le chemin d'exécution emprunté. Cependant, il est aussi possible qu'un accès soit classifié comme *Unknown* alors qu'il mène toujours à un hit (ou un miss), à cause d'une approximation trop grossière. Nous proposons donc une nouvelle analyse de cache d'instructions LRU, capable de classifier certains accès comme *Definitely Unknown*, une nouvelle catégorie représentant les accès pouvant mener à un hit ou à un miss. On est alors certain que la classification de ces accès est due au programme et à la configuration du cache, et pas à une approximation peu précise. Par ailleurs, cette analyse réduit le nombre d'accès candidats à une reclassification par des analyses plus précises mais plus coûteuses.

Notre principale contribution est une analyse capable de produire une classification de précision optimale. Celle-ci repose sur une méthode appelée *block focusing* qui permet le passage à l'échelle en analysant les blocs de cache un par un. Nous profitons ainsi de l'analyse *Definitely Unknown*, qui réduit le nombre de candidats à une classification plus précise. Cette analyse précise produit alors une classification optimale pour un coût raisonnable (proche du coût des analyses usuelles May et Must).

Nous étudions également l'impact de notre analyse exacte sur l'analyse de pipeline. En effet, lorsqu'une analyse de cache ne parvient pas à classifier un accès comme *Always-Hit* ou *Always-Miss*, les deux cas (hit et miss) sont envisagés par l'analyse de pipeline. En fournissant une classification plus précise des accès mémoires, nous réduisons donc la taille de l'espace d'états de pipeline exploré, et donc le temps de l'analyse.

Par ailleurs, cette thèse étudie la possibilité d'utiliser l'analyse *Definitely Unknown* dans le domaine de la sécurité. Les mémoires caches peuvent être utilisées comme canaux cachés pour extraire des informations de l'exécution d'un programme. Nous proposons une variante de l'analyse *Definitely Unknown* visant à localiser la source de certaines fuites d'information.

# Acknowledgements

Foremost, I would like to thanks my advisors, Claire Maïza and David Monniaux for supporting me I started my master internship for years ago. This thesis would not be possible without their advices, knowledge, guidance and patience.

I am thankful to all the members of my PhD commitee (Björn Lisper, Kenneth McMillan, Hugues Cassé, Sébastien Faucou, Frédéric Pétrot and Jan Reineke) for evaluating my work and advising me concerning my manuscript.

I would like to thanks all people from the laboratory, for their kindness and guidance. I am particularly grateful to my friends, Maxime, Remy, Yanis, Clément, Noha, Aurianne, Vincent, Cyril, Alexandre, Matheus, Raphaël, Hang, Hamza, Denis and Anaïs, for all the good times spent together.

Many thanks to my family: parents, brother and sister, for supporting me during this thesis and before. Last but not least, I would like to thanks Marie for being here when I needed the most, and for being my partner in life.

# Contents

**1 Introduction** **1**
1.1 Organization of the manuscript . . . 3

**2 Context** **5**
2.1 Caches . . . 5
2.1.1 Blocks and Locality Principles . . . 6
2.1.2 Cache sets and associativity . . . 6
2.1.3 Index, Tag, Offset . . . 8
2.1.4 Replacement policies . . . 8
2.1.5 Caches and Address Translation . . . 13
2.1.6 Cache configuration in this thesis . . . 15
2.2 Static Analysis . . . 15
2.2.1 Abstract Interpretation . . . 16
2.2.2 Model Checking . . . 23
2.2.3 Cache analysis methods in this thesis . . . 28
2.3 State of the Art in Cache Analysis . . . 28
2.3.1 Cache Conflict Graph . . . 28
2.3.2 Analysis of LRU caches . . . 29
2.3.3 Ferdinand's May and Must analyses . . . 32
2.3.4 Persistence Analysis and Loop Unrolling . . . 35
2.3.5 Other replacement policies . . . 37
2.3.6 Cache analysis by Model Checking . . . 37

**3 Cache Analysis Complexity** **39**
3.1 Background . . . 39
3.2 Complexity of Replacement Policies . . . 42
3.2.1 Fixed associativity . . . 44
3.2.2 LRU . . . 45
3.2.3 FIFO . . . 49
3.2.4 PLRU . . . 56
3.2.5 NMRU . . . 60

**4 Exact Cache Analysis** **67**
4.1 Approximating the set of Definitely Unknown Accesses . . . 68
4.1.1 Reminder: Caches and Static Cache Analysis . . . 68
4.1.2 Abstract Interpretation for Definitely Unknown . . . 70
4.1.3 Definitely Unknown Proofs . . . 74
4.1.4 Experimental Evaluation . . . 79

4.2 Exact Analysis of LRU Cache by Model Checking . . . 83
4.2.1 Block Focusing . . . 84
4.2.2 Proof of Block focusing correctness . . . 86
4.3 Exact Analysis of LRU Cache by Abstract Interpretation . . . 88
4.3.1 Exact Analyses as Fixed-Point Problems . . . 90
4.3.2 Data Structures and Algorithms . . . 93
4.4 Experiments . . . 96
4.4.1 Refinement of Accesses classification by Exact Analyses . . . 96
4.4.2 Efficiency comparison of Model Checking and ZDD approach . . . 97
4.4.3 May/Must and Exact analyses execution time comparison . . . 97

**5 Applications** **99**
5.1 WCET Application . . . 99
5.2 Experiments . . . 102
5.2.1 WCET comparison . . . 103
5.2.2 Analysis time comparison . . . 104
5.3 Security . . . 108
5.3.1 Program model and semantics . . . 109
5.3.2 Our vulnerability analysis . . . 111
5.3.3 Analyses soundness . . . 114

**6 Conclusion** **117**
6.1 Future Work . . . 119
6.1.1 Program Semantics . . . 119
6.1.2 Analyzing Data Caches . . . 124
6.1.3 Reducing analysis cost . . . 125
6.1.4 Other replacement policies . . . 126

**7 Résumé en français** **138**

# Chapter 1

# Introduction

Power plants, trains, airplanes, communication satellites and spacecrafts are some examples of safety-critical real-time systems. They are safety critical because any failure of such a system can lead to a human loss or huge economic cost, and they are real-time systems because they have to react fast enough to their inputs and environments to be correct. In other words, safety critical real-time systems are systems that have to respect their deadlines, and whose failures can have disastrous consequences. An example of such system is a task that periodically reads some sensors to correct the trajectory of a plane. If the period of the task is one millisecond, the task must terminate in less than one millisecond (or less if input/output jitter is constrained).

However, because of the always increasing amount of data to treat by time unit, modern processors use a wide variety of hardware optimizations that make the execution time of such systems hard to predict. In particular, the use of cache memories aims at keeping the processor busy in presence of a DRAM-based main memory, by retaining frequently used instructions and data in a fast memory close to the processing unit. Indeed, the latency of an instruction involving the CPU alone is usually several orders of magnitude lower than the main memory latency. Thus, in the presence of caches, the latency of an individual memory access may vary considerably depending on whether the access is a *cache hit*, i.e; it can be served from an on-chip cache memory, or a *cache miss*. Ensuring that a real-time program meet its deadline is thus harder in presence of caches.

In practice, static analysis approaches for bounding the worst-case execution time (WCET) of programs have to take into account whether or not accessed information is cached. The purpose of cache analyses considered in thesis is thus to statically classify every memory access at every machine-code instruction in a program into one of the following three classes:

1. *Always Hit*: each dynamic instance of the memory access results in a cache hit;
2. *Always Miss*: each dynamic instance of the memory access results in a cache miss;
3. there exist dynamic instances that result in a cache hit and others that result in a cache miss.

This is of course, in general, an undecidable question; so all analyses involve some form of abstraction, and may classify some accesses as "unknown". More precisely, the undecidability issue is usually tackled by making the assumption that every execution path in the program is feasible. Of course, building a program that violate this assumption (for instance, by adding some dead-code) is trivial. However, the program model (which consider all paths as feasible) covers all the actual behaviors of the real program. An analysis that correctly treat all possible executions of the model thus treats all the executions of the program.

Even under this “all path feasible” assumption the problem of memory classification seems difficult, and many cache analyses use additional abstraction. In this thesis, we thus look at the impact of this assumption on the cache analysis problem. More precisely:

- We investigate the complexity of performing an optimally precise cache analysis under this hypothesis. If the problem is decidable under this assumption, some caches seem easier to analyze that others. We thus investigate the difficulty of some cache analysis related problems under this hypothesis, for different cache replacement policies (that is, the algorithm used for choosing which memory block to evict from the cache to make room for a new block), from the complexity theory point of view.
- As mentioned above, there exist analyses that can guarantee that some accesses lead to a hit, or a miss. However, there is no analysis to our knowledge that can guarantee an access belongs to the category 3 above (i.e. no known analysis ensures the existence of paths leading to both a hit and a miss for some accesses).
- Finally, the question of the efficiency of an optimally precise cache analysis in the context of the “all path feasible” hypothesis is worth investigating. In practice, analyzed binaries are industrial program with restricted control flow, and caches have relatively low associativity. It is thus interesting to look at the practical efficiency of optimally precise cache analyses.

The goal of this thesis is to address the three points above.

### Contributions

**Complexity of optimally precise cache analysis** While the intuition is that a cache retains the most recently accessed memory words, up to its size, reality is far more complex: what happens depends on the number of cache levels, the size of each level, the “number of ways” (also known as the *associativity*) of the cache and the *cache replacement policy*. Cache analyses depend on the cache replacement policy, and, in the literature, there is a clear preference for the LRU (*Least Recently Used*) policy, notably the well-known age-based abstract analysis of Ferdinand [AFMW96] and its variations. In contrast, other policies such as PLRU (pseudo-LRU), NMRU (*Not Most Recently Used*) and FIFO (*First-In, First-Out*) have a reputation for being very hard to analyze [HLTW03a] and for having poor predictability [RGBW07a]. A legitimate question is whether these problems are intrinsically difficult, or is it just that research has not so far yielded efficient analyses. Indeed, issues of static analysis of programs under different cache policies are not necessarily correlated with the practical efficiency of cache policies. Static analysis is concerned with worst-case behavior, and policies with approximately equal “average”[1] practical performance may be very different from the analysis point of view. Even though PLRU and NMRU were designed as “cheap” (easier to implement in hardware) alternatives to LRU and have comparable practical efficiency [AMM04], they are very different from the worst-case analysis point of view. We thus explore the complexity of the memory access classification problem under different replacement policies.

***Definitely Unknown* memory accesses** As mentioned above, analyses like the *May* and *Must* analyses presented in [AFMW96] rely on approximations to classify accesses as *Always-Miss*

[1]By “average” we do not imply any probabilistic distribution, but rather an informal meaning over industrially relevant workloads, as opposed to examples concocted for exhibiting very good or very bad behavior.

or *Always-Hit*. In this setting, one does not have any information on the remaining unclassified accesses. Indeed, any access not classified can be always a hit (or miss) not detected by the analysis because of the approximation, or can both lead to a hit or a miss depending on the execution path in program. We thus introduce a new category, called *Definitely Unknown*, for distinguishing those accesses that can lead to both a hit or a miss. The distinction between *Definitely Unknown* and unclassified (*Unknown*) accesses is important because it characterizes the precision of an analysis. *Unknown* accesses can be refined into *Always-Hit*, *Always-Miss* or *Definitely Unknown* by improving the precision of the analysis. However, *Definitely Unknown* accesses are a consequence of the program and the cache analyzed, not of the analysis. These accesses have no chance to be refined into *Always-Hit* or *Always-Miss* by improving the precision of the cache analysis. We then propose an analysis that, similarly to the *May* and *Must* analyses of [AFMW96], is able to approximate the sets of *Definitely Unknown* memory accesses for *LRU* caches under the "all path feasible" assumption.

**Exact cache analysis** Once the accesses that can not be refined are filtered out, one can focus on the remaining candidates. For these few candidates, more expensive analyses can be considered. In particular, it is possible to treat them one by one and, by doing so, to design specific analysis that focus on a single block, pruning the state space to explore. In this thesis, we thus propose two approaches to remove the remaining uncertainty about memory accesses, one relying on a model checker, the other performed by abstract interpretation.

**Applications** Our last contribution is an application of the analyses design in this thesis in two different domains. First, the uncertainty about the classification of a memory access can have serious consequences on the WCET analysis. For instance, it is very important to have precise information about the cache behavior when analyzing pipelined and superscalar architectures, since pipeline analysis must consider the two cases "cache hit" and "cache miss" for any memory access that cannot be shown to be *Always-Hit* or *Always-Miss* [LS99, $R^+06$], leading to a state explosion. Thus, imprecise cache analysis may have two adverse effects on WCET analysis. First, it may lead to an excessive overestimation of the WCET compared to the true WCET. An industrial user may suspect this when the upper bound on WCET given by the tools is far from experimental timings. This may discourage the user from using static analysis tools. In addition, imprecise cache analysis might results in excessively high WCET analysis time due to state explosion.

Finally, we explore some security related questions concerning caches. *Definitely Unknown* blocks might results in both hits or misses. When this variation is observable by an attacker and correlated to the variation of some secrete data, the cache might be the source of information leakage. We thus propose to use our *Definitely Unknown* analyses to detect such vulnerabilities. In addition, we propose some slight modification that allow to spot the origin of the leakage by tracking the last access to the leaking block.

## 1.1 Organization of the manuscript

Chapter 2 describes the context of this thesis. It describes how caches work, introduces some static analysis methods, and present the *May/Must* analysis of LRU caches. The problem of classifying memory accesses into *hits* and *misses* is studied from the complexity theory point of view in Chapter 3. Among other things, this chapter shows that the usual replacement policies lead to high complexity classification problems (most analyses are at least NP-hard). Chapter 4 introduces several techniques that allows to derive an optimally precise classification by abstract

interpretation and/or model checking. Two applications of cache analyses are investigated in Chapter 5. First, we study the benefit of a precise cache analysis on the WCET. Then, some aspects related to security are investigated. Finally, we discuss and conclude about the results and possible extensions of our work in Chapter 6.

# Chapter 2

# Context

## 2.1 Caches

Any computer needs memory to perform arbitrary computations, and this memory is usually available in different kinds. Registers form the most frequently used one. Located in the CPU core, they provide high speed and low latency. However, the total amount of memory available as registers is usually very low. For example, RISC-V architectures only provide 16 or 32 registers of 32 bits each, for a total of 128 bytes maximum.

Because of this very low capacity, computers rely on another kind of memory, called main memory, to perform computations. This main memory offers a much higher capacity (around 8GB for a recent personal computer), at the expense of a higher latency. Even though compilers tend to maximize the use of registers over main memory to decrease computation latency, accessing the main memory is unavoidable. In case the value of the accessed memory is needed immediately, the processor is simply stalled and waits for the data to be available to pursue the execution. However, the speed of modern processors is usually far beyond the speed of the main memory. Thus, such a simple computer only consisting in registers and main memory would spend most of its time waiting for the main memory to issue the requested data and instructions. To avoid this problem, called the Memory Wall [WM95], modern processors make heavy use of *cache memories*.

Cache memories offer a capacity/speed trade-off intermediate between registers and main memory: they are bigger but slower than registers, and smaller but faster than main memory. To speed computation up, the main idea is to store into caches information that is frequently accessed in the main memory, reducing the time needed to retrieve the information and thus the overall execution time. When the CPU issues a memory access, the information is searched in the cache. If it is found, the access is said to be a *cache hit* and the information is retrieved quickly. Otherwise, the access is a *cache miss* and the information is retrieved from the memory. When this happens, a copy of the retrieved data is stored into the cache to avoid a future cache miss. For efficiency reasons, the chunk of copied data contains several consecutive bytes, and not the accessed byte only. These chunks, called cache blocks, have the same size fixed by the hardware, and are aligned in memory according to this size.

The idea of speeding up information retrieval by introducing an intermediate memory level can be repeated. Most modern CPU architectures indeed use several levels of caches, offering different capacity/speed trade-offs. The fastest level, which also offers the lowest capacity, is referred to as L1 cache, and is often split into an instruction cache and a data cache. The next cache level is referred to as L2, and is bigger but slower than L1 caches. Contrarily to L1 caches, the L2 cache usually mixes instructions and data, both data and instruction L1 caches use the L2 cache as a backing store. Finally, many architectures nowadays offer a L3 cache (or even a L4

cache [KCB+15]), that is shared between different cores. Figure 2.1 shows an example of typical cache hierarchy.

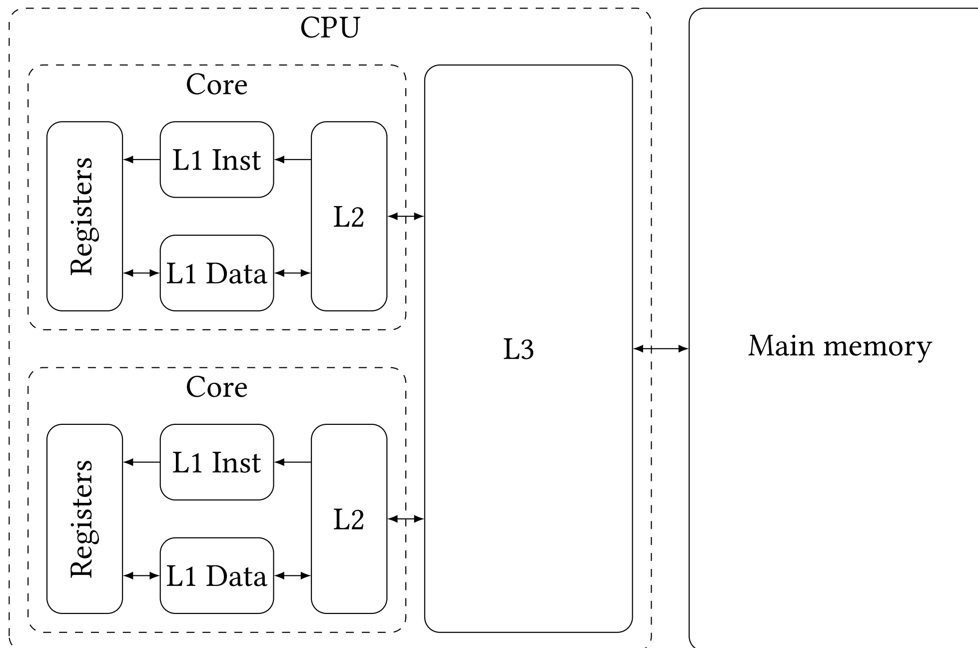



Figure 2.1 – Example of cache hierarchy (Intel Core i3 Haswell)

### 2.1.1 Blocks and Locality Principles

The efficiency of caches relies on two key principles:

- Time locality: when a word is accessed, it will be accessed again soon with high probability.
- Spatial locality: when a word is accessed, words close to it will be accessed with high probability.

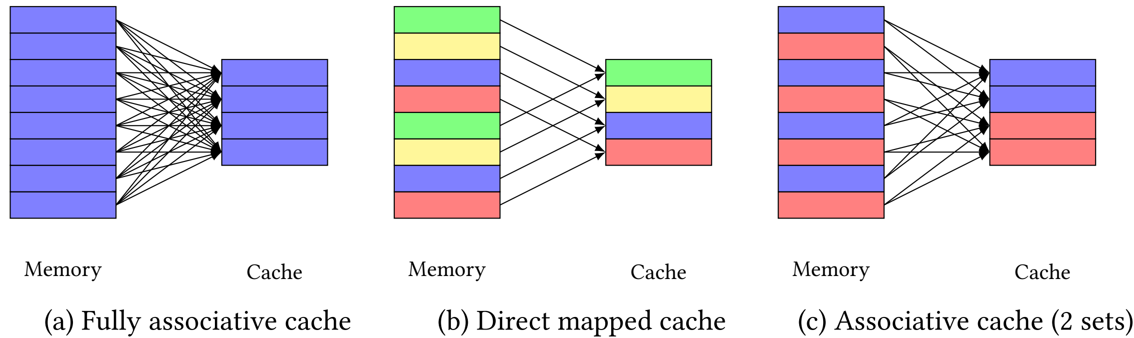
On one hand, one should load into the cache large cache blocks to profit from space locality. Indeed, by loading into the cache words that are contiguous to the recently accessed word, one increases the chances that the next word accessed will already be cached. On the other hand, caches should keep words available for the processor as long as possible to benefit from time locality. The two principles are thus adversarial: the bigger the cache blocks are, the fewer can be cached, and the faster they are evicted. In practice, a typical trade-off for the size of memory blocks is 32 to 64 bytes (see Table 2.2 later in the section). The area occupied by a memory block in the cache is called a *cache line*.

### 2.1.2 Cache sets and associativity

For efficiency reasons, a given memory block can usually only be stored in the cache at some locations and nowhere else, to reduce the time needed to retrieve it. This set of cache lines a block can be stored in is called a *cache set* and is fully determined by the block address. Generally, this cache set is given by a part of the block address (i.e. a part of the address of the first byte in the block). Thus, the cache is split into several cache sets of equal size, each of them associated to different cache blocks. The number of cache sets composing the cache is usually a power of 2 varying with the cache size [Rei09]. Two special cases can be distinguished:

- The cache contains a single cache set: the cache is said to be *fully associative*, a memory block can thus reside in any cache line.

- The cache contains as many cache sets as blocks (i.e. each cache set is composed of a single cache line): the cache is *direct mapped*, a memory block can only reside in the cache line associated to it.




(a) Fully associative cache (b) Direct mapped cache (c) Associative cache (2 sets)

Figure 2.2 – Example of mapping from main memory to cache

Figure 2.2 illustrates different possible cache set configurations. Each rectangle represents a memory block, and its color identifies its associated cache set. Arrows designate the possible cache lines a memory block can be stored in. Figure 2.2a shows the mapping of memory to a fully associative cache, a memory block can be loaded in potentially any cache line. Figure 2.2b is an example of direct mapping, a memory block can only be stored in one cache line. Finally, Figure 2.2c is the intermediate situation: the cache is composed of a few cache sets of several cache lines each.

As shown on Figure 2.2 different cache configurations are possible for a fixed cache size and block size, each of them being uniquely defined by its number of cache sets. Alternatively, a cache configuration can be identified by the number of cache lines (also named *number of ways*) contained in a single cache set. This number is called the cache *associativity* and is often denoted by $k$.

For example, caches represented on Figure 2.2a, Figure 2.2b and Figure 2.2c respectively have associativity 4, 1, and 2. Table 2.1 shows how the number of sets, the cache associativity and the cache size vary for an example processor[1], the Intel Core i3-3110M.

| Cache | Size | Number of sets | Associativity | Block size |
|---|---|---|---|---|
| L1 Data | 32 KB | 64 | 8 | 64 |
| L1 Instruction | 32 KB | 64 | 8 | 64 |
| L2 Unified | 256 KB | 512 | 8 | 64 |
| L3 Unified | 3 MB | 4096 | 12 | 64 |

Table 2.1 – Characteristics of Intel Core i3-3110M CPU

[1]Data were extracted from the `/sys/devices/system/cpu/cpuX/cache/indexY` directory on a laptop using the mentioned CPU.

### 2.1.3 Index, Tag, Offset

As mentioned above, the location of a given byte in the cache is determined by its address. More precisely, any address can be split into two distinct parts: i) the least significant bits, which form an offset indicating the location of the byte into the block ii) the remaining part of the address, which constitutes the block address. The block address can in turn be decomposed into a tag, and an index. The index identifies the unique cache set the block can be loaded in, whereas the tag is used to distinguish blocks mapping the same cache set. Figure 2.3 recaps the complete slicing of the address.

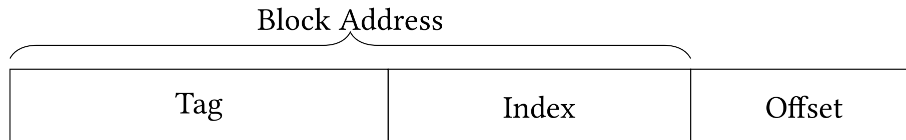



Figure 2.3 – Address splitting

### 2.1.4 Replacement policies

As mentioned previously, when a memory access results in a cache miss, the block accessed is issued by the main memory and a copy is loaded into the cache. The cache set where this block is stored is given by the address of the block. However, it happens that the cache set a block should be stored in is already full. In this case, one has to select a block from this cache set and evict it to make space available for the new block. Note that a block eviction can occur when the cache is not full, but a set is.

When a block must be evicted, it is selected according to a *replacement policy*. Many cache policies exist, offering different efficiency/implementation cost trade-off. Among the most widely used one can find LRU (Least Recently Used), PLRU (Pseudo-Least Recently Used), FIFO (First In First Out) and NMRU (Not Most Recently Used).

Cache sets are usually managed independently, i.e. the state of a cache set only depends on the accesses that map to this cache set. However, in some replacement policies (such as the Pseudo-Round Robin policy), an access to a cache set influences the other cache sets, thus the state to be considered is global. In this thesis, we consider only local cache set management.

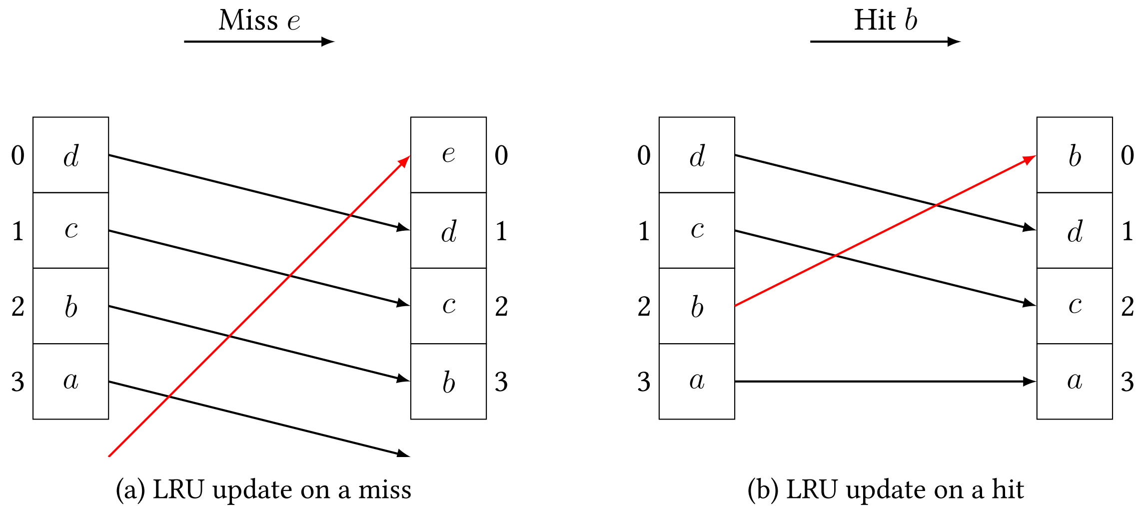
#### LRU: Least Recently Used

The Least Recently Used policy maintains a list of blocks ordered from the most recently used to the least recently used among each cache set. Upon a miss, the accessed block is loaded at the MRU (Most Recently Used) position and other blocks are simply shifted by one logical position. Finally, the Least Recently Used block is evicted. Figure 2.4a illustrates the update of a cache set containing blocks $a$, $b$, $c$ and $d$ when accessing block $e$.

Upon a hit, no block is evicted: the accessed block is simply moved to the MRU position, and blocks that had been more recently used (i.e. before the accessed block) are shifted by one position. This case is depicted by Figure 2.4b.

Note that hardware implementations of LRU do not actually move memory blocks from one cache line to another. Instead, the logical position of blocks is maintained separately in additional registers (see [TAMV19] for details of several possible implementations).

Because the blocks are ordered from the most recently used to the least recently used, the position of a block among a cache set is usually called its *age*. For example, block $c$ on Figure 2.4a



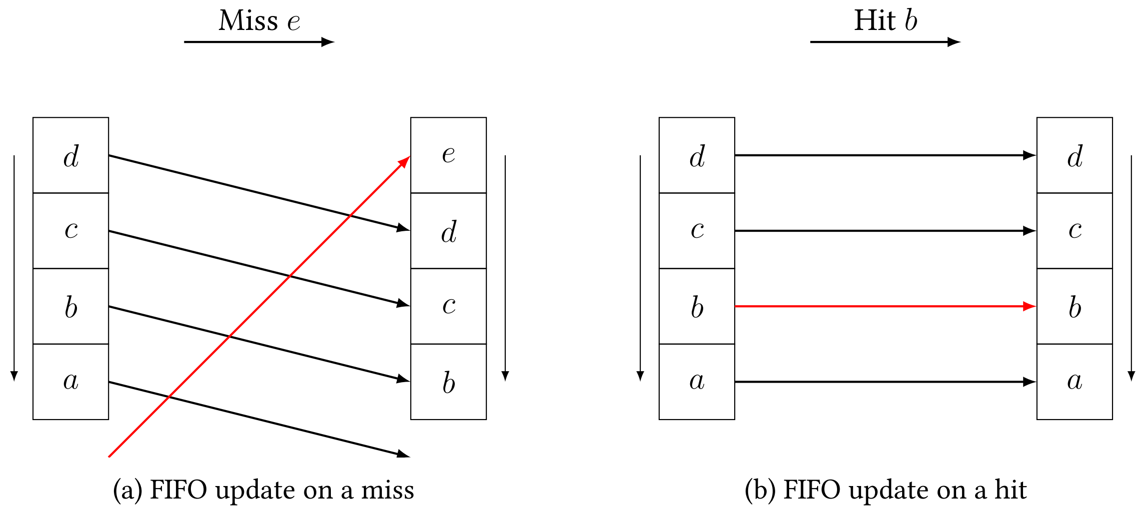

(a) LRU update on a miss

(b) LRU update on a hit

Figure 2.4 – LRU behavior

has age 1 before the access to e, and age 2 after the access. The older a block is, the closer it is from being evicted. More precisely, a block that has just been accessed will be evicted after exactly $k$ misses [Rei09].

**FIFO: First-In First-Out**




(a) FIFO update on a miss

(b) FIFO update on a hit

Figure 2.5 – FIFO behavior

Similarly to LRU, the First-In First-Out policy works like a queue: on a miss, the accessed block is inserted at the head and the block at the tail is evicted. However, contrarily to LRU, the cache set is left unchanged in case of a hit. Figure 2.5 presents both cases. One can notice that FIFO behaves as LRU on a miss (see Figure 2.5a), and as the identity function on a hit (see Figure 2.5b).

The different behavior of FIFO relatively to LRU has two main implications:

- FIFO is much simpler to implement at hardware level. A simple "pointer" going from one cache line to the next one is enough to track which block will be evicted on the next miss. Moreover, the cache update is very fast when it needs to be i.e. in case of a hit, because there is no action to perform.

- A block that has just been accessed can be evicted on the next access. This happens when an access to the last block of the queue is followed by a miss which evicts it. This behavior makes FIFO caches harder to predict. An example of this phenomenon, called “domino effect”, is given in [Lun02].

### NMRU: Not Most Recently Used

As its name suggests the Not Most Recently Used policy does not provide guarantee as strong as the LRU policy: instead of ensuring that the least recently used block is evicted, it makes sure the block evicted on a miss is not the most recently used one. To do so, one extra bit, called MRU bit, is associated to each cache line. If this bit is a 1, then the cache line might contain the most recently used block. Conversely, if the MRU bit value is 0, then the associated cache line can not contain the last block accessed.

To preserve this property, the NMRU policy works as follows:

- On a miss, the first block (from left to right) which MRU bit is 0 is replaced by the accessed block, and its MRU bit is flipped to 1.
- On a hit, the MRU bit of the accessed block is set to 1 whatever its value is.
- In both cases if the MRU-bit of the block accessed is the last one to 0, all other MRU-bits are reset to 0 (this is called a global-flip). This ensures that at any time, there is at least one cache line which MRU bit is 0.

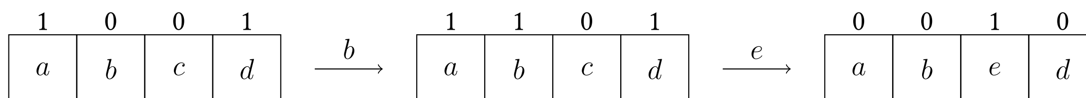



Figure 2.6 – NMRU behavior

This behavior is illustrated by Figure 2.6. The first access is an access to block $b$ which is cached. The MRU associated to $b$ is then flipped to value 1. Then, an access to $e$ is performed, and leads to a miss. The first cache line which MRU bit is 0 is the one containing $c$. $c$ is thus evicted to store $e$ and the associated MRU bit is set to 1. Finally, because this bit was the last one with value 0, all other bits are set to 0.

Note that this policy is sometimes referred as Most Recently Used, or PLRU-bit (i.e. Pseudo-Least Recently Used - *bit version*, as opposed to the *tree version* described below).

Similarly to the FIFO policy, blocks stored in an NMRU cache can be evicted quickly. After only two accesses, a block that has just been loaded can already be evicted. However, some approaches (see [GLYY14]) show that NMRU caches, although not as predictable as LRU caches, are more predictable than other replacement policies.

### PLRU: Pseudo-Least Recently Used

As indicated by its name the Pseudo-Least Recently Used policy approximates the LRU policy. To do so, cache lines of a set are organized into a full binary tree, in which leaves are cache lines and other nodes contain bits (called MRU-bits) indicating where is the next block to evict. More precisely, when a block must be evicted, one recursively looks at the value of a node, starting from the root. If the node contains 0, then the block to evict should be evicted from the left subtree,

otherwise (when the node contains 1) it should be evicted from the right subtree. This process is repeated until a leaf is reached, the associated cache line is then used to store the new block.

In case of a hit, the tree is explored from bottom to top. Each time a node is encountered, its MRU-bit is set to protect the accessed block from eviction. In other words, for a given node in the tree, three outcomes are possible on a hit:

- If none of the right and left subtrees contain the accessed block, the MRU-bit of the node is unchanged.
- If the accessed block is in the left subtree, then the MRU-bit of the node is set to 1.
- If the accessed block is in the right subtree, then the MRU-bit of the node is set to 0.

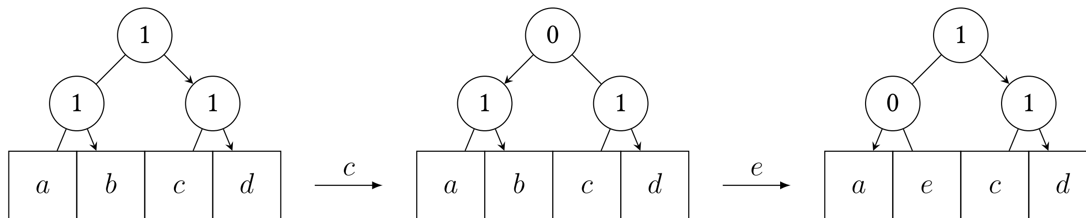



Figure 2.7 – PLRU behavior (To ease reading, we use arrows to reflect the value of nodes, i.e. when node's value is 0 the associated arrow points to left and conversely)

Figure 2.7 illustrates both hit and miss cases:

- The first memory block access is $c$, and it is cached. Thus, all arrows on the path from the root to the cache line containing $c$ are flipped to point away. The second level arrow already points towards $d$, it is thus kept unchanged. On the other hand, the first level arrow now points to the left subtree, which does not contain $c$ (the root bit is changed to 0).
- The next access is a miss because $e$ is not cached. We thus find the block to evict by following the arrows. In this case, $b$ is evicted and replaced by $e$. Finally, all bits on the path to $e$ are flipped.

This policy is named PLRU because it behaves like the LRU policy in case of misses. More specifically, on a long sequence of misses, the cache lines are replaced one by one circularly, mimicking the LRU behavior, up to a permutation of cache lines. However, in the general case, only $\log_2(k) + 1$ accesses might be enough to evict a block that has been stored (see [Rei09]), whereas LRU requires $k$ accesses. Indeed, the tree-bit of a PLRU cache has height $\log_2(k)$, and accessing $\log_2(k)$ blocks can thus change the arrows direction to point to a given cache line. With one more access (a miss), the given cache line is evicted.

### Writing policies

This thesis mainly focuses on the analysis of instruction caches. Some of the analyses described can be used to analyze data caches, but require some modification. This section describes the aspects that make data caches differ from instruction caches behavior.

In addition to the replacement policy, the behavior of a data cache is described by a write policy. Indeed, contrarily to instruction caches, which are usually read-only, data caches must handle write accesses too.

The first aspect of the write policy consists in specifying whether memory blocks are loaded (allocated) into the cache in case of write accesses. One usually distinguishes two allocation policies:

- The *write allocate* policy simply consists in loading the block into the cache (if needed, i.e., in case of a write miss) before modifying it.
- The *no-write allocate* policy does not load the block in case of a write miss. Instead, the block is modified directly in the next level cache (or main memory). Thus, this policy does not need to evict a block in case of a write miss. The drawback is that successive writes to the same block induce several slower writes at the backing store level.

The second aspect of the write policy concerns the modification of the backing store (i.e., next level cache or main memory) on a write:

- The *write-through* policy forwards the write to the backing store: on a write, the block is modified both in the cache (if loaded, according to the allocation policy) and in the next level cache. One advantage of this policy is that it greatly simplifies the coherence protocol between cache levels. However, it might suffer efficiency penalties when handling repeated writes to the same block.
- The *write-back* policy modifies the block in the cache, mark it as “dirty”, and delays the modification of the next-level cache to the eviction time. When a block is evicted, the cache checks if it is marked dirty. If this is the case, the backing store is updated to reflect the new value of the block (i.e. the evicted block). If the block is not dirty, then the backing store and the cache are consistent and there is no need to modify the backing store.

One can thus distinguish four possible write policies:

- Write allocate, Write-through: this policy is rarely used because the need to modify the next-level cache on each write greatly reduces the advantage of allocating the block.
- Write allocate, Write-back: this policy is the most complex to implement but handles sequences of writes very efficiently. The first write to a given block might suffer a miss, but the next writes will not. Finally, the backing store is only modified once, on the block eviction.
- No-write allocate, Write-through: this policy is the dual of the previous one. It does not handle write sequences very efficiently, but the hardware implementation is simplified.
- No-write allocate, Write-back: again, this policy is rarely used. Indeed, in this configuration, write-back is only useful if the block has been loaded first on a read, because writes do not trigger cache line allocation.

This additional complexity coming from the writing policy make data cache harder to analyze. Not only these analyses must distinguish between write accesses and read accesses, but they also need to track the status of dirty bits in case of write-back caches.

### Examples of Cache Memories

Table 2.2 shows some examples of cache memories one can find in different processors. Note that this table only gives a rough idea of the processors behavior and hide many implementation details. In practice, cache memories are often pipelined, and interact with each other through

write-buffers and victim caches[2], and are tightly coupled with prefetchers or branch predictors. Table 2.2 shows common values of associativity and cache size, but does not reflect the complete cache behavior. Note that detailled information about cache implementation are not always available. Some approaches thus try to infer the cache characteristics from a set of experiments designed do distinguish replacement policies (see [AR14])

| CPU | Level | Cache Size | Number of ways | Block Size | Number of sets | Replacement Policy | Details |
|---|---|---|---|---|---|---|---|
| Kalray MPPA | L1I | 8K | 2 | 32B | 128 | LRU | - |
| | L1D | 8K | 2 | 64B | 64 | LRU | Write-back/ Write-through configurable |
| MPC7450 | L1I | 32K | 8 | 32B | 128 | PLRU | - |
| | L1D | 32K | 8 | 32B | 128 | PLRU | Write-back/ Write-through configurable |
| Intel Core XScale | L1I | 32K | 32 | 32B | 32 | FIFO | - |
| | L1D | 32K | 32 | 32B | 32 | FIFO | Write-back/ Write-through configurable |
| Intel Core i7 Nehalem | L1I | 32K | 4 | 64B | 128 | Unknown | Write-back No-write allocate |
| | L1D | 32K | 8 | 64B | 64 | Unknown | Write-back No-write allocate |
| | L2 | 256K | 8 | 64B | 512 | Unknown | Write-back |
| | L3 | 8M | 16 | 64B | 8192 | MRU | Write-back |

Table 2.2 – Examples of processor caches

### 2.1.5 Caches and Address Translation

The analysis of cache behavior require knowledge of the memory address accessed by the program. However, these addresses are not fully known at run-time, because of the address translation mechanism operating systems use to manage the main memory between processes. This section describes this mechanism and how it interferes with caches.

#### Virtual Memory

To avoid memory corruption, each process running on a system has its own memory areas that no other process can access. However, the location of these areas are unknown at compilation time, and are attributed at running time by the operating system. The solution to this problem is to add an abstraction layer to dissociate the physical memory from the memory used by the program (see [HP12, Chapter 5.7] for more detailed explanations). The program then operates on virtual memory, and all manipulated addresses are translated at run time into physical addresses. This translation is usually performed by the operating system, and accelerated through hardware support.

[2]Victim caches are caches that exclusively store blocks that are evicted from another cache.

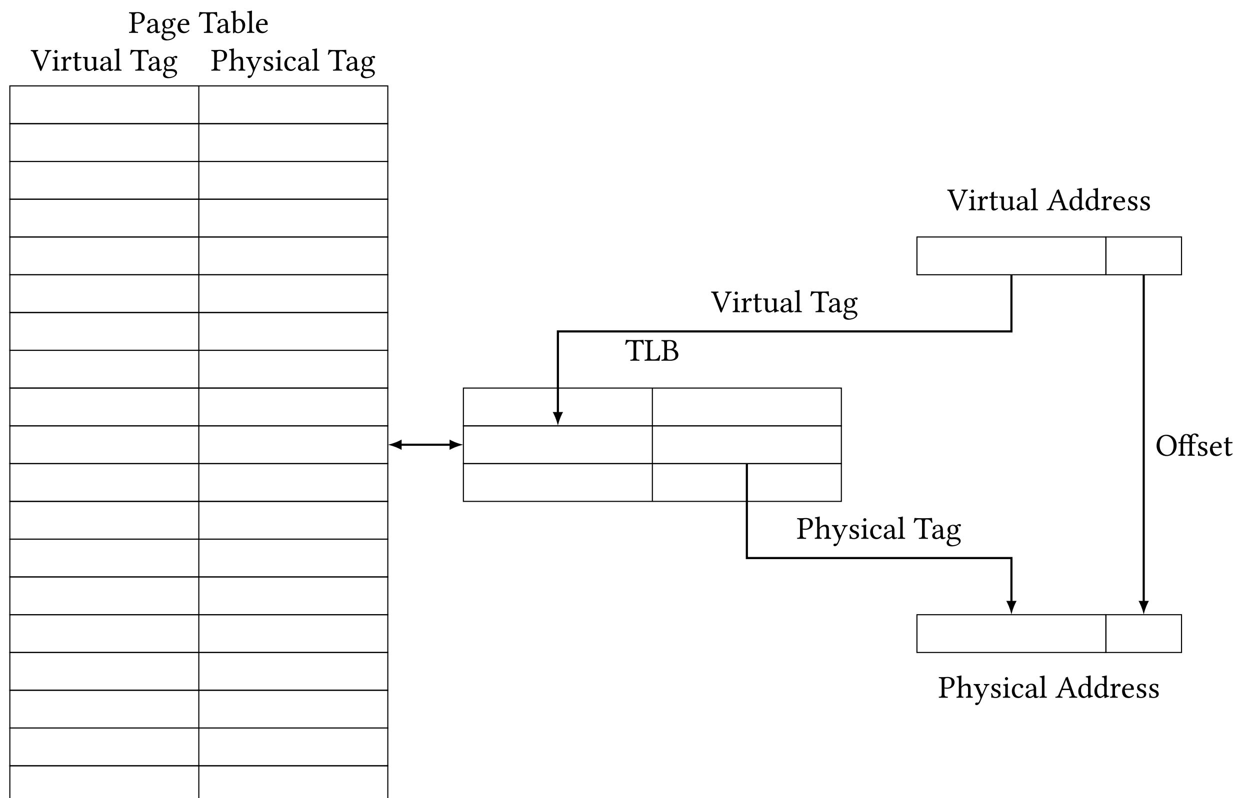



Figure 2.8 – Virtual Address Translation

For efficiency reasons, address translation is performed by splitting virtual and physical memory into contiguous blocks called pages (a usual page size is 4 KiB). Because pages are aligned in memory, the offset designating a byte inside a page is left unchanged by translation. The remaining part of an address is called the *tag*, and is translated by inspecting the page table. This table being looked up on every memory access translation, it is often cached by a dedicated cache called Translation Lookaside Buffer. The whole translation process is described on Figure 2.8.

### Virtual and Physical Indexing and Tagging

The translation of the virtual address into the physical one must be taken into account when designing cache mechanisms, because it might impact the value of the cache tags, indexes and offsets. Here are some widely used configurations:

- Using the physical address of the blocks to cache them. This is the simplest solution: translate the address first, and then use the physical address to determine the cache set the block should be stored in. However, this solution is also very slow, because one waits until the translation process is finished to perform cache look-up.

- Using the virtual address of the block to cache it. This solution can be more efficient than the previous one because cache look-up and address translation can be performed in parallel, and address translation is bypassed in case of cache hits. However, using the virtual cache index and the tag can lead to complications, because the same virtual address can refer to two different blocks when used in two different processes. In this case, either the operating system flushes the cache on a context switch, or the cache stores an address space identifier in addition to the tag to distinguish between the two synonym blocks.

- A trade-off between these two strategies consists in using a virtual index and a physical tag. The cache set a block is potentially cached in is then identified during the address translation and the physical tag is available when looking up the cache set. This solution however imposes some constraints on cache configuration because both cache offset and index should fit into the page offset.

- Finally, more complex strategies can be considered by relying on extra hardware support or by taking advantage that some caches are read-only.

In this thesis, we do not consider the address translation mechanism and we assume that every memory access is associated with an address known at compile time. This assumption is obviously safe in the case of caches using virtual tags and indexes, because the addresses manipulated by the program and the cache are identical (the virtual addresses are used by both). In this thesis, we only consider L1 instruction caches, which usually work with indexes and offsets that both fit in the page offset. Cache offset and index are thus invariant by translation. We thus additionally assume that mapping from virtual pages to physical pages is injective for pages containing instructions.

### 2.1.6 Cache configuration in this thesis

The idea of caches, i.e., storing frequently used data in fast memories to speed computation up, is simple and natural. However, the practical implementations of hardware caches involve many efficiency/cost trade-offs, and the mechanisms used to maximize performance of modern CPU are very complex.

Thus, in order to study the possible behaviors of a program in relation to the cache, this thesis makes several simplifying assumptions. Mainly, we focus on the analysis of L1 instruction caches that use the LRU policy, for several reasons:

- Analyzing data caches requires knowledge about data addresses, which can lead to complex static analysis problem (e.g., pointer aliasing analyses), out of the scope of this thesis. We thus focus on the analysis of instruction caches, because instruction addresses are more predictable than data addresses. However, in case the addresses of data accessed are known, some approaches developed in this thesis might be used to analyze data caches, by adapting them slightly to take the writing policy into account (see Section 6.1).

- Higher level caches usually mix instructions and data, and thus suffer the same analysis difficulties as data caches. In addition, analyzing caches beyond level L1 requires to take complex cache coherence protocol into account. Finally, L3 caches are often shared, and can thus suffer from interferences from other cores. Note that a part of the complexity in analyzing higher-level caches comes from the difficulty of predicting hits and misses at lower level. Thus, a precise cache analysis at level L1 would benefit higher-level cache analyses.

- Concerning the replacement policy, we focus on LRU because, as shown in chapter 3, it already leads to high complexity problems. Other replacement policies lead to even harder classification problems (see Section 6.1).

## 2.2 Static Analysis

Static analysis consists in proving properties of a program without having to execute it, as opposed to dynamic program analysis, which ensures that the program behaves correctly on some inputs

by executing it. The main reason to perform static analysis rather than dynamic analysis is that the first can guarantee that a given property is true for all program executions, whereas the second guarantees that the property is true only on the tested inputs.

The fundamental drawback of static analysis techniques is that the vast majority of properties one would like to prove about a program are undecidable. Is is thus impossible to build to a static analyzer that decides such a property and is correct for any program on any input. One way to circumvent this problem is to allow the static analyzer to answer "unknown" when deciding if a property holds or not. In this case, running a static analysis tools leads to one of the following outcomes:

- the analyzer is able to cover all inputs of the program and check that the desired property holds. The program is then accepted.
- the analyzer is able to find at least one input value for which the program behavior is incorrect. The program is then rejected.
- the analyzer answers "unknown", it is not able to ensure the correctness of all behaviors of the program. The program might be correct, but is rejected.

This section introduces two static analysis methods that tackle the undecidability problem:

- Abstract Interpretation computes over-approximations to cover all the possible program behaviors. If the over-approximations performed are sound, then this approach guarantees that the desired property holds for any execution of the program. Indeed, if the property holds for all the behaviors covered by the analysis, then it must hold for the subset of behaviors that the program can exhibit. However, if the approximation performed is too coarse, a correct program (i.e., satisfying the property) might be rejected because spurious behaviors are taken into account.
- Model Checking consists in checking the correctness of a finite model of the program by exhaustively exploring all its states [3]. Symbolic Model Checking [McM93] relies on an implicit representation of the state space. Using compact descriptions of Boolean functions, one can represent a huge state space in a reasonable model.

  One can choose an abstraction of the program as a model; such an approach is thus tightly related to abstract interpretation.

These two methods offer different performance performance/precision trade-off, and are thus complementary. We thus use both of them in this thesis.

### 2.2.1 Abstract Interpretation

Abstract interpretation [CC77] is a framework used to ensure the soundness of over-approximations. The main idea is to characterize all possible program states using a representation that is expressive enough to prove the desired property, but allows fast computations. This is illustrated on Figure 2.9: green shapes symbolize the evolution of possible programs states step by step. The solid arrows are examples of transition from one concrete state to another one. Blue ellipses represent over-approximations of the sets of concrete states, and dashed arrows

[3] In Bounded Model Checking [CBRZ01], one truncates the program execution after $k$ steps. The resulting model is then obviously finite but the method is unsound, as it might miss incorrect behaviors that happen after the $k$th step.

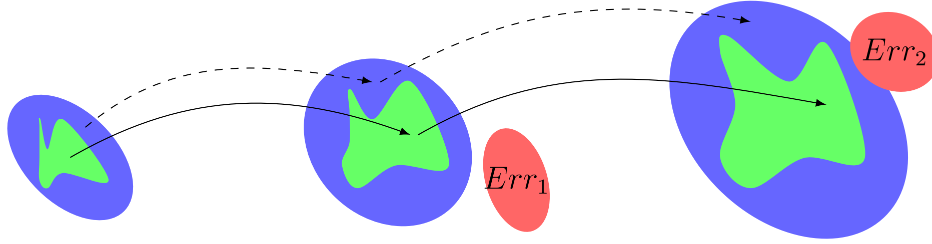
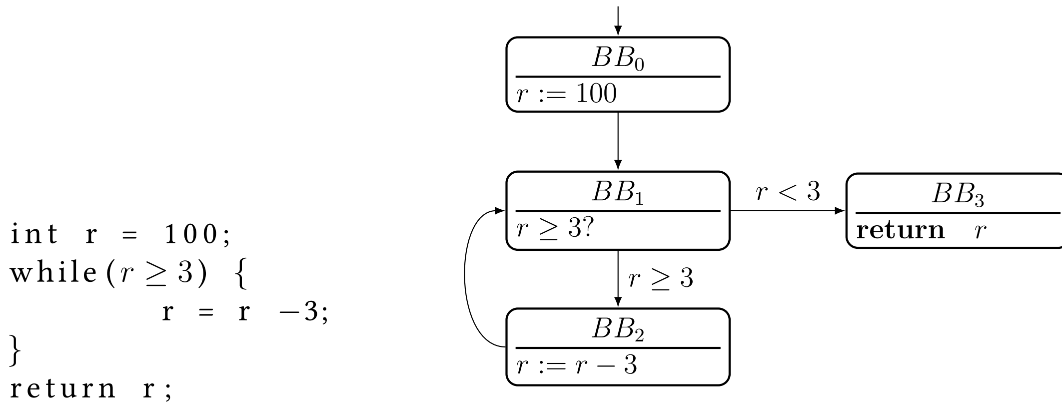



Figure 2.9 – Over-approximation of program states

symbolizes the transition between over-approximations. If the over-approximation does not cover any invalid state (in red on the figure), then the program is safe. In our example, this is illustrated by the error states $Err_1$, which are guaranteed to be unreachable because they do not intersect the blue ellipses (and thus, they do not intersect the green shape). However, it might be that the program is safe but rejected by the analyzer because of the precision loss. This phenomenon, called "false alarm", corresponds to the $Err_2$ states on Figure 2.9. They are unreachable (they do not intersect the green shape) but the analysis can not prove it (they intersect the blue area).

### Control Flow Graph

In order to formalize the notion of program execution we use the concept of Control-Flow Graph (CFG), which represents all the possible execution paths of a program. A CFG is a directed graph whose nodes, called basic blocks, symbolize sequences of instructions which flow cannot be broken: once the execution reaches the first instruction of a basic block, the whole block is executed sequentially until the end. Outgoing edges represent the flow to the next possible basic blocks that can be executed. A CFG is a data structure that represents all possible program execution paths. The program is then modeled by representing each function by its CFG.




(a) Program Example

(b) CFG example

Figure 2.10 – A simple program and its associated Control-Flow Graph

Consider the program of listing 2.10a, which performs a naive Euclidean division. We will use this program as an example in this section. Figure 2.10b then represents the CFG of this program and shows how the program executes:

- The program starts at the entry node, basic block $BB_0$, where the value 100 is stored in variable $r$.
- The execution proceeds in basic block $BB_1$. The current value of $r$ is compared to 3, and because it is greater, the execution flow goes to basic block $BB_2$.
- In $BB_2$ $r$ is decremented by 3, and thus takes value 97.
- $BB_1$ is executed again and the new value of $r$ is compared to 3.
- After 33 iterations of the loop, $r$ has value 1, and the flows finally goes to the final block $BB_3$.

In this thesis, the notation $G = (V, E, v_0)$ denotes a graph where $V$ is the set of basic blocks, $E \subseteq V^2$ is the set of edges, and $v_0 \in V$ is the entry node, which is assumed to be unique. In our example, we have:

$$\begin{aligned} V &= \{BB_0, BB_1, BB_2, BB_3\} \\ E &= \{(BB_0, BB_1), (BB_1, BB_2), (BB_2, BB_1), (BB_1, BB_3)\} \\ v_0 &= BB_0 \end{aligned}$$

### Concrete Semantics

In the context of abstract interpretation, the behavior of a program is captured by its concrete semantics, which is given by two elements:

- a *concrete domain* $D_{conc}$ which can precisely express all possible program states. A frequent choice is to use a map that associates a value to each variable of the program. In the case of our division example we have one variable $r$, and there is no need to use such a map. Instead, one can use the concrete domain[4] $D_{conc} = \mathbb{Z}$ that tracks the value of $r$.
- *a set of transformers* are introduced to model the effect of program instructions. These transformers are functions from $D_{conc}$ to $D_{conc}$ that, given the value at the entry of a basic block, computes the value at the end of this block. In our example, the transformers that capture the program behavior are the following:
  - $f_{BB_0} : r \mapsto 100$
  - $f_{BB_1} : r \mapsto r$
  - $f_{BB_2} : r \mapsto \begin{cases} r - 3 & \text{if } r \geq 3 \\ \text{undefined} & \text{otherwise} \end{cases}$
  - $f_{BB_3} : r \mapsto \begin{cases} r & \text{if } r < 3 \\ \text{undefined} & \text{otherwise} \end{cases}$

  Basic blocks $BB_1$ and $BB_3$ do not modify $r$, whereas basic blocks $BB_0$ and $BB_2$ set it to 100 and decrement it by 3 respectively (if the guard is satisfied).

Given a finite path in the CFG of the program and an initial value in the concrete domain, one can apply the sequence of transformers associated to the sequence of blocks met along the path

[4] Note that some authors call *concrete domain* what we later call the *collecting semantics domain*. Here, we use the same convention than [Rei09].

and get the concrete value describing the program state at the end of the path. In our example, given the path $(BB_0, BB_1, BB_2, BB_1, BB_2)$, one would obtain for a concrete initial value $r$:

$$
\begin{aligned}
(f_{BB_2} \circ f_{BB_1} \circ f_{BB_2} \circ f_{BB_1} \circ f_{BB_0})(r) &= (f_{BB_2} \circ f_{BB_1} \circ f_{BB_2} \circ f_{BB_1})(100) \\
&= (f_{BB_2} \circ f_{BB_1} \circ f_{BB_2})(100) \\
&= (f_{BB_2} \circ f_{BB_1})(97) \\
&= f_{BB_2}(97) \\
&= 94
\end{aligned}
$$

This value obtained at the end of basic block $BB_2$ after two loop iterations is independent of the initial value of $r$, because the first instruction of the program erases this initial value and replaces it by 100.

This process can be used to define the concrete value after the execution of any path, but one is generally more interested in the set of all possible program states at a given location, whatever the execution path is, rather than a single concrete value. In our example, one could be interested in the set of values that reach basic block $BB_2$ for instance, to check that the subtraction never results in an overflow.

This computation of all concrete values at a program location $v$ is what the collecting semantics $F : V \rightarrow \mathcal{P}(D_{conc})$ achieves, by aggregating the concrete values obtained for all paths ending at $v$.

Before giving a formal definition of the collecting semantics, we illustrate how it can be computed on our example. First, we look at the set of concrete values that can reach the entry block. At this point $r$ is not initialized yet, and we thus assume that any integer value is reachable. Hence, we have $F(BB_0) = \mathbb{Z}$. Then, after the execution of $BB_0$, all these values are replaced by the assigned value 100. This gives the following inclusion: $F(BB_1) \supseteq \{100\}$. By propagating this information, one get that $F(BB_2) \supseteq \{100\}$ because 100 satisfies the guard condition $r \geq 3$. We thus deduce that $\{97\}$ is a subset of values reaching the end of $BB_2$, and that therefore it is a subset of $F(BB_1)$. This lead to the new inclusion $F(BB_1) \supseteq \{97, 100\}$, by combining the new reachable value with the set of reachable values previously computed. By iterating this process, one gets to the point where we know $F(BB_1) \supseteq \{1, 4, \dots, 97, 100\}$. To propagate this set through the edge $(BB_1, BB_2)$, it is intersected once again with the edge condition $r \geq 3$, leading to $F(BB_2) \supseteq \{4, 7, \dots, 97, 100\}$. We thus deduce that $\{1, 4, \dots, 94, 97\}$ is subset of values reachable at the end of block $BB_2$. For the first time, this set is included in the set of values reaching $BB_1$ previously. The process finally stabilizes, and we know that the exact set of values reaching $BB_1$ is $F(BB_1) = \{1, 4, \dots, 97, 100\}$. By intersecting it with the edge guard $r < 3$, we then deduce that $F(BB_3) = \{1\}$. The final solution obtained by this iterating process is thus:

- $F(BB_0) = \mathbb{Z}$
- $F(BB_1) = \{1, 4, \dots, 97, 100\}$
- $F(BB_2) = \{4, 7, \dots, 97, 100\}$
- $F(BB_3) = \{1\}$

Intuitively, the values that reach the beginning of a block can be computed from the values coming from predecessors. This leads to the formal definition of the collecting semantics $F$, as the least fixpoint solution of the following equation:

$$\forall v \in V, F(v) = F_0(v) \bigcup_{(u,v)\in E} \{f_u(c), c \in F(u)\},$$

$$\text{where } F_0(v) = \begin{cases} \{c_0\} & \text{if } v = v_0 \\ \{\} & \text{otherwise} \end{cases}$$

$F(u)$ is the set of values that reach the beginning of the basic block $u$. Thus, $\{f_u(c), c \in F(u)\}$ is the set of values that reach the end of $u$. By computing the union of all those sets over all predecessors $u$ of $v$, one obtains the set of values reaching the basic block $v$. The term $F_0(v)$, where $F_0 : V \to D_{conc}$, is added to take the entry value into account. It is always empty, except for the entry node, which is reached with initial value $c_0$.

The solution $F$ of this fixpoint equation is guaranteed to exist[5]. However, the iterating process described above is not guaranteed to reach a fixpoint in finite time.

### Abstract Semantics

To avoid this expensive computation and/or ensure that the process finishes, one computes an approximation of the concrete collecting semantics instead of the collecting semantics itself. The main idea is that a set $X \subseteq D_{conc}$ of concrete values will be abstracted by a single value $\alpha(X)$. This value $\alpha(X)$ then represents all values in $X$, and possibly other additional concrete values (in blue in Figure 2.9). The function $\alpha : \mathcal{P}(D_{conc}) \to D_{abs}$ is called an abstraction and $D_{abs}$ is named the abstract domain.

In our example, one possible abstraction is to use an integer interval instead of a set of integers. For instance, the set $\{97, 100\}$ will be abstracted by $\alpha(\{97, 100\}) = [97, 100]$. Thus, $\alpha(\{97, 100\})$ covers the set $\{97, 100\}$, plus the extra values 98 and 99. This abstraction function $\alpha$ is then defined by:

$$\begin{aligned} \alpha : \mathcal{P}(\mathbb{Z}) &\to \textbf{Intervals} \\ X &\mapsto [\min X, \max X] \end{aligned}$$

To make the usage of an abstraction meaningful, $D_{abs}$ and $\alpha$ are required to satisfy some properties (see [CC77]):

- $D_{abs}$ must be equipped with an order relation (i.e., reflexive, transitive and antisymmetric) $\sqsubseteq$ that makes it a complete lattice $(D_{abs}, \sqsubseteq, \sqsupseteq, \sqcup, \sqcap)$. This relation represents the relative preciseness of abstract values: if $x \sqsubseteq y$ then $x$ is more precise than $y$.
- $\alpha$ is monotonic: $X \subseteq Y \Rightarrow \alpha(X) \sqsubseteq \alpha(Y)$, i.e. it conserves the preciseness relation.

The requirement that $D_{abs}$ should be a lattice ensures that any abstract values $x$ and $y$ can be joined together in one that approximates both: $x \sqcup y$. This is needed to compute the abstract value at the entry of a block that has many predecessors. In the Euclidean division example, after the two loop iterations the value at the beginning of basic block $BB_1$ is built from the value $[100, 100]$ coming from $BB_0$, and the value $[94, 97]$ coming from $BB_2$. These two values are joined together into the single interval $[94, 100]$. Note that this join operation has lost precision, because it covers values that were not represented in any of the two joined intervals (namely 98 and 99). If $x$ and

[5] This results from Tarski's fixpoint theorem [Tar55], knowing that F is continuous and $D_{conc}$ is a complete lattice.

$y$ are the abstract values at the end of predecessors, then $x \sqcup y$ is the most precise abstract value that covers all concrete values represented by $x$ and $y$.

An equivalent possibility is to express the relation between concrete and abstract value by introducing a concretization function $\gamma : D_{abs} \to \mathcal{P}(D_{conc})$ instead of the abstraction $\alpha$. This concretization associates to an abstract value the set of concrete values it approximates. For instance, the concretization one would use in our example is the following:

$$\gamma : \textbf{Intervals} \to \mathcal{P}(\mathbb{Z})$$
$$[a, b] \mapsto \{x \in \mathbb{Z}, a \leq x \leq b\}$$

To ensure consistency of the definitions, the $(\alpha, \gamma)$ pair is required to be a Galois connection [CC77], i.e. it must satisfy the following properties:

- $\forall X \in \mathcal{P}(D_{conc}), (\gamma \circ \alpha)(X) \supseteq X$
- $\forall x \in D_{abs}, (\alpha \circ \gamma)(x) \subseteq x$

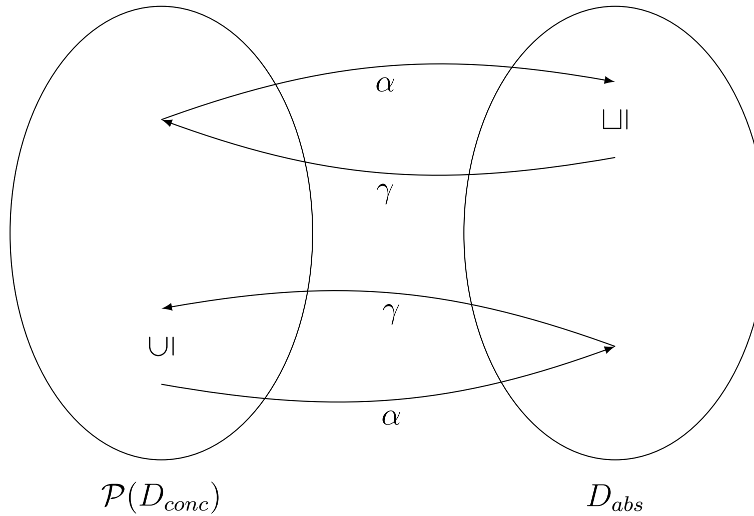



Figure 2.11 – Composition of abstraction and concretization

These conditions ensure that one cannot lose safety, but only precision, by repetitively applying $\alpha$ and $\gamma$ to a value. By abstracting a concrete value $x$ by $\alpha(x)$, we over-approximate it. If the set resulting from the concretization of $\alpha(x)$ does not contain $x$, then this over-approximations is incorrect. For example, by abstracting and concretizing the set $\{97, 100\}$, one obtains $\gamma(\alpha(\{97, 100\})) = \gamma([97, 100]) = \{97, 98, 99, 100\}$. This value covers the input $\{97, 100\}$, illustrating the correctness of the $(\alpha, \gamma)$ pair. Figure 2.11 illustrates how these compositions of concretization and abstraction behave. By applying one after the other, one can only go "higher" in the lattice and lose precision. Note that when $(\alpha, \gamma)$ is a Galois connection, $\alpha$ uniquely defines $\gamma$ and conversely (see [NNH99, p. 239] for a proof).

Once an abstraction is chosen, one must define the abstract transformer associated to each basic block. These abstract transformers are the abstract counterpart of the concrete transformers and express the program behavior in the abstract domain. One way to define the abstract transformers is to compose the concrete transformers with the abstraction and concretization functions. The obtained functions, $f_v^\# = \alpha \circ f_v \circ \gamma$, are the most precise abstract transformers that can be used, and are called the best abstract transformers. In our example, the best abstract transformers are:

- $f^{\#}_{BB_0} : [a, b] \mapsto [100, 100]$
- $f^{\#}_{BB_1} : [a, b] \mapsto [a, b]$
- $f^{\#}_{BB_2} : [a, b] \mapsto \begin{cases} \emptyset & \text{if } [a, b] \cap [3, +\infty] = \emptyset \\ [a' - 3, b' - 3] & \text{if } [a, b] \cap [3, +\infty] = [a', b'] \end{cases}$
- $f^{\#}_{BB_3} : [a, b] \mapsto \begin{cases} \emptyset & \text{if } [a, b] \cap [-\infty, 2] = \emptyset \\ [a', b'] & \text{if } [a, b] \cap [-\infty, 2] = [a', b'] \end{cases}$

This family of transformers can be used to derive the abstract collecting semantics $F^{\#} : V \rightarrow D_{abs}$, which is defined similarly to the concrete collecting semantics but in the abstract domain. More formally, $F^{\#}$ is a solution of the fixpoint equation:

$$\forall v \in V, F^{\#}(v) = F^{\#}_0(v) \bigsqcup_{(u,v) \in E} \{f^{\#}_u(F^{\#}(u))\},$$
$$\text{where } F^{\#}_0 = \alpha \circ F_0$$

In our case, applying the iteration process described previously for the concrete collecting semantics yield the following solution:

- $F^{\#}(BB_0) = [-\infty, \infty]$
- $F^{\#}(BB_1) = [0, 100]$
- $F^{\#}(BB_2) = [3, 100]$
- $F^{\#}(BB_3) = [0, 2]$

Note that those values completely cover the results obtained by the concrete collecting semantics.

**Analysis Soundness**

The main benefit of the abstract interpretation framework is the soundness proofs it automatically provides. Indeed, it is guaranteed that the fixpoint reached by the abstract collecting semantics covers all the values collected in the concrete domain:

$$\forall v \in V, F(v) \subseteq \gamma(F^{\#}(v))$$

In addition, this soundness proof can be generalized to any set of safe transformers. Let $(\widehat{f^{\#}_v})_{v \in V}$ and $\widehat{\sqcup}$ verifying the two following safety conditions:

$$\forall v \in V, \forall x \in D_{abs}, f_v(\gamma(x)) \subseteq \gamma(\widehat{f^{\#}_v}(x))$$
$$\forall (x, y) \in D^2_{abs}, \gamma(x) \cup \gamma(y) \subseteq \gamma(x \widehat{\sqcup} y)$$

These conditions ensure that abstract transformers correctly captured their concrete counterpart. A safe over-approximations cannot become unsafe by applying the abstract transformer. This means that no concrete behavior of the program can be lost during the abstract interpretation. Note that those conditions trivially hold for the best abstract transformers and the abstract lattice join operator. When those conditions hold, any fixpoint $\widehat{F^{\#}}$ of the following equation:

$$\forall v \in V, \widehat{F^{\#}}(v) = \widehat{F_0^{\#}}(v) \widehat{\bigsqcup}_{(u,v)\in E} \{\widehat{f_u^{\#}}(x), x \in \widehat{F^{\#}}(u)\},$$
$$\text{where } \widehat{F_0^{\#}} \sqsupseteq \alpha \circ F_0$$

is guaranteed to satisfy the soundness property:

$$\forall v \in V, F(v) \subseteq \gamma(\widehat{F^{\#}}(v))$$

Note that the iteration process described above (sometimes referred as chaotic iteration, or worklist algorithm) is not guaranteed to terminate in general. However, all abstract domains introduced in this thesis are finite, and thus, the process is guaranteed to terminate without additional requirements.

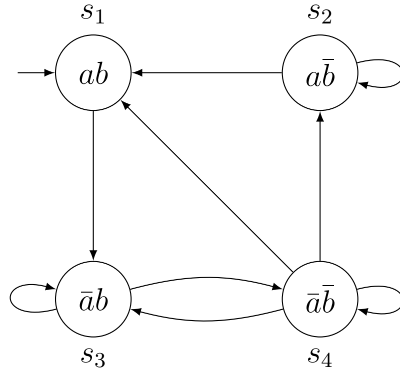
Many cache analyses introduced in this thesis are formalized in the abstract interpretation framework. The different concrete and abstract domains introduced then represented several aspect of the cache state/behavior.

### 2.2.2 Model Checking

Model Checking [EC82] is a static analysis method that is used to ensure that some system satisfies a given property. The system is described by a set of variables and is modeled as finite state machine, and the property to check is modeled as a temporal formula over those variables. The role of the model checker is then to ensure that the finite state machine evolves safely according to the formula.

```
init(a)  :=  true
init(b)  :=  true
next(a)  :=  case
             b  :  false
             a  :  true
             true  :  {false , true}
next(b)  :=  case
             a & b  :  true
             true  :  {false , true}
```

(a) Example of boolean program




(b) Example of Finite State Machine

Figure 2.12 – A simple program and its associated Kripke structure

To give the intuition of how model checkers work, this section focuses on the model checking of Boolean programs (i.e. programs manipulating boolean variables only). Most modern model checkers support integer variables, sometimes relying on a Satisfiability Modulo Theory solvers, but we do not consider those complex solvers in this thesis. Consider the Boolean (non-deterministic) program of Listing 2.12a. This program is composed of two parts:

- The initialization of all variables. In our cases, both $a$ and $b$ are initially set to true.
- The definition of the transition relation, which express the evolution of variable from one step to the next. Here, both $a$ and $b$ evolve differently depending on their current value. for instance, the next value of $a$ is computed as follows:

- If $b$ is true, then next value of $a$ will be false.
- Otherwise, if $a$ is true, the next value of $a$ will be true.
- Finally, when the default case applies (i.e when both $a$ and $b$ are false), the next value of $a$ is chosen non-deterministically.

Note that such a program can use bounded integers. One can simply use one boolean variable for each bit of the integer, and encode arithmetic operations into the transition relation. This might however highly increase the program complexity.

The program 2.12a contains two boolean variables $a$ and $b$, and is thus composed of four states: one for each valuation of the variables. Those states can then be connected according to the transition relation, forming the finite state machine depicted on Figure 2.12b. For example, in state $s_1$, where both $a$ and $b$ are true, one knows from the transition relation that $a$ and $b$ will respectively take the values false (first case of the next(a) affectation) and true (first case of the next(b) affection). The only state reachable from $s_1$ is thus the state $s_3$, where $a$ is false and $b$ is true. Note that non-deterministic affectations lead to several successors for the same state.

Formally, the program model is a Kripke structure $\mathcal{M} = (S, T, I, L)$, where $S$ is the set of states, $T \subseteq S^2$ is the transition relation, $I \subseteq S$ is the set of initial states and $L : S \to \mathcal{P}(Var)$ is a label function with associates to a state the set of variables that are true in this state. For instance, in the model shown on Figure 2.12b we have:

- $S = \{s_1, s_2, s_3, s_4\}$
- $T = \{(s_1, s_3), (s_2, s_1), (s_2, s_2), (s_3, s_3), (s_3, s_4), (s_4, s_1), (s_4, s_2), (s_4, s_3), (s_4, s_4)\}$
- $I = \{s_1\}$
- $L(s_1) = \{a, b\}$ , $L(s_2) = \{a\}$, $L(s_3) = \{b\}$ and $L(s_4) = \{\}$

**Temporal logic**

Temporal logics are extensions of propositional logic with temporal operators. They can thus express relations between states of a program model and distant by several time steps. Temporal formulas are useful to express complex properties of a program, involving several states at different time. For example, one can express for the program 2.12a that there is no way to go from a state where both $a$ and $b$ are true to a state where they are both false in a single step. This property can be expressed by the formula $(a \wedge b) \Rightarrow \neg X(\neg a \wedge \neg b))$. The $X$ in the formula stands for “neXt” and is temporal operator that express property of the next state. Informally, $s_4$ is the only state where both $a$ and $b$ are both false. Thus, $X(\neg a \wedge \neg b)$ designates states from which $s_4$ is immediately reachable (namely $s_3$ in our example). The precise semantics of temporal operators is presented later in this section. First, we define the syntactically valid formulas.

Temporal formulas can express property about a single state, or property about a whole path, and are classified into state formulas (noted $\phi$) and paths formulas (noted $\psi$). The grammar of Figure 2.13 shows all combination allowed in the process of building those two kinds of formulas.

To formalize the semantics of temporal formulas, we first define the notion of path in a Kripke structure: a path $w$ is an infinite word of states $w = (w_0 w_1 \ldots)$, where all $w_i \in S$, and $(w_i, w_{i+1}) \in T$. We also note $w^i$ the path extracted from $w$ that starts at step $i$: $w^i = (w_i w_{i+1} \ldots)$.

In addition to the usual proposition logic operators $\neg$, $\vee$, $\wedge$ that conserve their usual semantics, six other operators (All, Exist, neXt, Future, Globally and Until) are introduced:

- The state formula $A\psi$ holds in state $s$ if $\psi$ holds for all paths starting at $s$.

$$
\begin{aligned}
\phi &:= \textbf{true} \\
\phi &:= \textbf{false} \\
\phi &:= p, \text{ where } p \in Var \\
\phi &:= \neg\phi_1 \\
\phi &:= \phi_1 \vee \phi_2 \\
\phi &:= \phi_1 \wedge \phi_2 \\
\phi &:= A\psi_1 \\
\phi &:= E\psi_1
\end{aligned}
\qquad
\begin{aligned}
\psi &:= \phi_1 \\
\psi &:= \psi_1 \vee \psi_2 \\
\psi &:= \psi_1 \wedge \psi_2 \\
\psi &:= X\psi_1 \\
\psi &:= F\psi_1 \\
\psi &:= G\psi_1 \\
\psi &:= \psi_1 U \psi_2
\end{aligned}
$$

Figure 2.13 – Temporal formula grammar

- Similarly, the state formula $E\psi$ holds in state $s$ if $\psi$ holds for at least one path starting at $s$.
- $X\psi$ holds on the path $w$ if $\psi$ holds on the path $w^1$.
- $F\psi$ holds on the path $w$ if $\psi$ holds on at least one of the path $w^i$, where $i \geq 0$.
- $G\psi$ holds on the path $w$ if $\psi$ holds on all paths $w^i, i \geq 0$.
- $\psi_1 U \psi_2$ holds on the path $w$ if there is an $i \geq 0$ such that $\psi_2$ holds on $w^i$ and $\psi_1$ holds on all the paths $w^j, j \geq i \geq 0$.

Using these operators, one can express correct behaviors of the program. For example, property $AG((\neg p) \vee Fq)$ states that if variable $p$ is true at any time during an execution, then $q$ will become true later on.

This logic system that allows mixing path quantifiers and temporal operators freely is called CTL* [EH83]. In the following, we describe two subsets of CTL* that are less expressive but lead to easier model checking problems: Linear Temporal Logic [Pnu77] (LTL) and Computational Tree Logic [EC82] (CTL).

LTL is the fragment of CTL* obtained by restraining the use of path quantifiers. More formally, all LTL formulas have the form $A\psi$, where $\psi$ does not contain any of the quantifier $A$ or $E$. Branching over paths is thus only allowed at the very beginning of the formula.

CTL is the subset of CTL* obtained by allowing temporal operators $X$, $F$, $G$ and $U$ only immediately after a path quantifier $A$ or $E$, but nowhere else. This is equivalent to extend the propositional logic with operators $AX$, $AF$, $AG$, $AU$, $EX$, $EF$, $EG$ and $EU$. CTL can thus be formalized using state formula only. An example of CTL formula is $AG(p \wedge EXq)$, which states that at any point of any execution $p$ is true and there is one successor state where $q$ is true.

One way to check the validity of a CTL formula for a given Kripke structure is to use the labeling algorithm. This algorithm works by computing for each sub-formula the set of states that satisfy it. Starting from the leaves, the syntax tree of the formula is decorated in a bottom-up fashion with these sets. First, each atomic proposition $p$ is associated to the set of states where it is true: $\{s \in S, p \in L(s)\}$. For a non-atomic formula, the satisfying set is built from the sets associated to children sub-formulas. For example, the satisfying set of $\phi_1 \wedge \phi_2$ is built by intersecting the satisfying sets of $\phi_1$ and $\phi_2$. Temporal operators $EX$ and $AX$ are handled by computing the preimage of a set by the transition relation $T$. For instance, the satisfying set of $AX\phi$ is the set of states which immediate successors are all in satisfying set of $\phi$. For the more complex operators $F$, $G$ and $U$, the satisfying sets are computed as a fixpoints. Adding a state to

the set might lead to new states satisfying the formula. Those new states are thus added to the satisfying set, and so on, until there is no additional state to add.

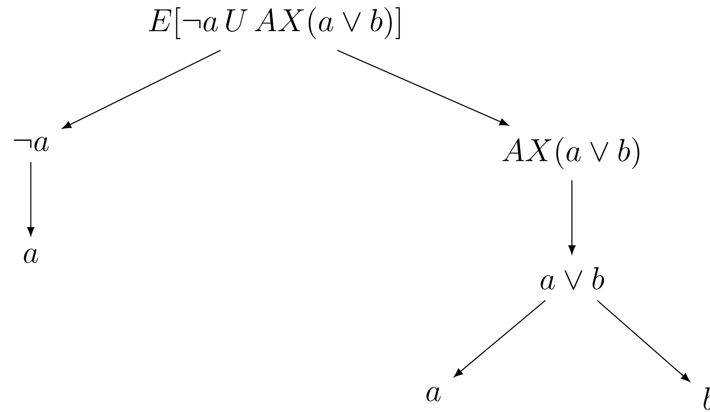



Figure 2.14 – the syntax tree of formula $\phi = E[\neg a\, U\, (AX(a \vee b))]$

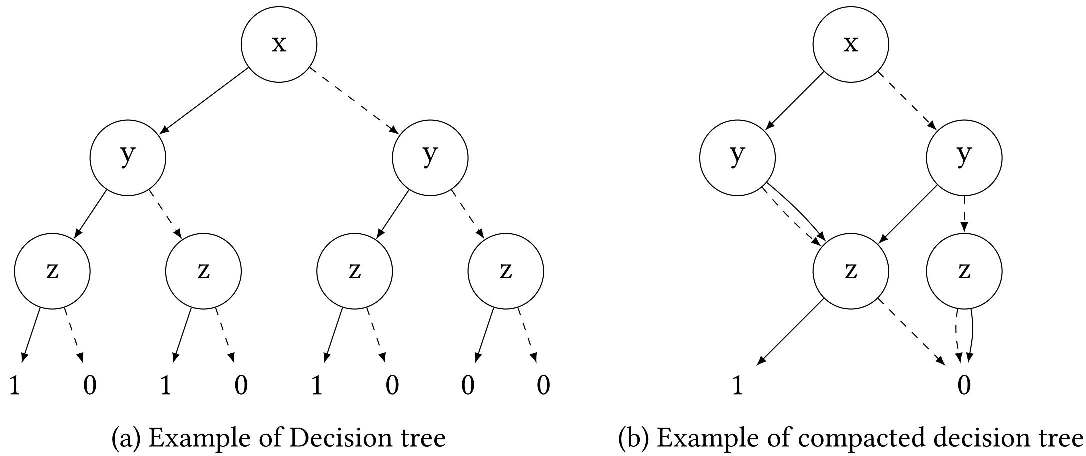
For example, checking that the property $\phi = E[\neg a\, U\, (AX(a \vee b))]$ holds in the initial state of the Kripke structure of Figure 2.12b is done step by step, in the following way:

- First, the formula is converted to its syntax tree, represented on figure 2.14. Each node is associated to a sub-formula for which the satisfying set must be computed.
- Then, the leaves of the tree can be evaluated. In our case, property $a$ is true in states $s_1$ and $s_2$ and the satisfying set associated to the atomic formula $a$ is thus $\{s_1, s_2\}$. Similarly, the satisfying set of $b$ is $\{s_1, s_3\}$.
- Using this information, one can compute the satisfying set of the parent node $a \vee b$, by performing the union of satisfying sets of sub-formulas. a state $s$ satisfies the property $a \vee b$ if it belongs to the set $\{s_1, s_2, s_3\}$.
- One can thus examine the parent node $AX(a \vee b)$. By definition of $AX$, the states satisfying this property are the ones which all immediate successors satisfy $a \vee b$, i.e. they all belongs to the satisfying set of $a \vee b$. The satisfying set of $AX(a \vee b)$ is thus $\{s_1, s_2\}$. Indeed, $s_3$ and $s_4$ do not belong to this satisfying because one of they successor (namely $s_4$), does not belong to $\{s_1, s_2, s_3\}$.
- Computing the satisfying set associated to $\phi$ is then done by exploring the Kripke structure in a backward manner. At beginning, the set associated to $\phi$ is $\{s_1, s_2\}$, the satisfying set of $AX(a \vee b)$. Then, $s_4$ is added, because it satisfies $\neg a$ and one of its successors ($s_1$ or $s_2$) belongs to $\{s_1, s_2\}$. Finally, the process stabilizes and one get the satisfying set of $\phi$: $\{s_1, s_2, s_4\}$. The initial state $s_1$ of the Kripke thus satisfies property $\phi$.

### Decision Diagrams

Working with the Kripke structure associated to a program is usually very costly, because of the exponential size of the structure in the number of variables. One solution to avoid the state space explosion is to use decision diagrams to represent it compactly.

Decision diagrams are representations of Boolean formulas that are similar to decision trees. Given a Boolean formula, one can build the associated decision tree by removing the variables




(a) Example of Decision tree

(b) Example of compacted decision tree

Figure 2.15 – A decision tree and the compacted version

one by one. To remove a variable, one simply substitutes it in the formula by true or false, leading to two subtrees: one where the variable has been replaced by true, the other where it has been replaced by false. The process is repeated on both subtrees until no variable remain. A leaf then represents the truth value of the original formula when setting the variables according to the path that reaches it. For example, the decision tree associated to the formula $\varphi = z \wedge (x \vee y)$ is shown on Figure 2.15a. We use $1$ to represent the value true, and $0$ to represent false. The variable shown on each node is the one that is evaluated. We use a solid arrow when it is evaluated to true, and a dashed arrow when evaluated to false.

This representation as decision tree can be compacted in several ways. A first improvement consists in using a single occurrence of all identical subtrees. This process leads to a Directed Acyclic Graph, as shown on Figure 2.15b.

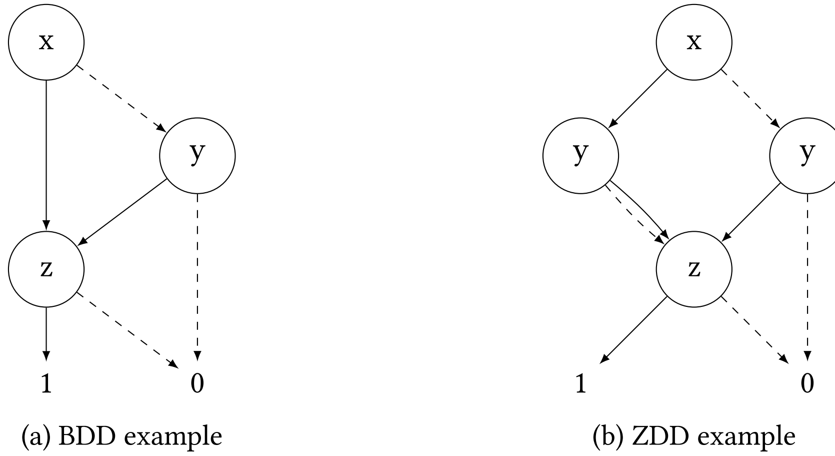
This DAG structure can be additionally compacted, by applying one of the two following reduction rules:

- Removing nodes for which both subtrees are identical. This leads to a new structure called Binary Decision Diagrams [Jr.78] (BDD). In our example, the obtained diagram is represented on Figure 2.16a. As in decision trees, a path in a BDD is equivalent to a partial assignment of the variables. The absence of a variable on a path means that the skipped variable has no influence on the truth value of the formula.

- Removing nodes for which the true subtree (plain arrow) points to false. The obtained structure (illustrated on Figure 2.16b) is called a Zero-Suppressed Decision Diagram [Min93] (ZDD). These diagrams are particularly useful to represent formulas that evaluate to false for most variable assignments. Indeed, many nodes can be removed in this case, leading to a very small representation.

These structures have two properties that make them suitable to represent the state space of a Kripke structure. First, for a given order of the variables, every formula has a canonical BDD (or ZDD) representation. Moreover, many operations can be performed efficiently on these structures: conjunction, intersection and projection of over a set of variables (and negation for BDDs). Finally, the re-use of some part of the diagrams at several location make them suitable for memoïzation.

Note that those representations only form one possible solution to solve efficiently model checking problems. Other algorithms exist. Among them, one can cite IC3 [SB11], and all the counter-example guided abstraction-refinement [CGJ+00] approaches.




(a) BDD example

(b) ZDD example

Figure 2.16 – Comparison of Binary and Zero-Suppressed Decision Diagrams

### 2.2.3 Cache analysis methods in this thesis

In this thesis, we introduce several precise cache analyses, performed using by abstract interpretation and model checking techniques.

On the one hand, model checking has the advantage of being usable in a "black-box" manner: one does not need to understand the complex algorithms and heuristics used by a model checker to use it. Providing a model of the program and a property to prove is sufficient. However, this advantage might also be a drawback because understanding why a model checker fails to prove some property or why it needs more time than expected requires deep knowledge about the model checker internals.

On the other hand, abstract interpretation requires more work up front (one has to provide an abstract domain and safe transformers and join operators), but offers more control on the computation performed.

Some analyses in this thesis get the best of these two worlds, by combining abstract interpretation methods using ZDDs for efficient representation of abstract values.

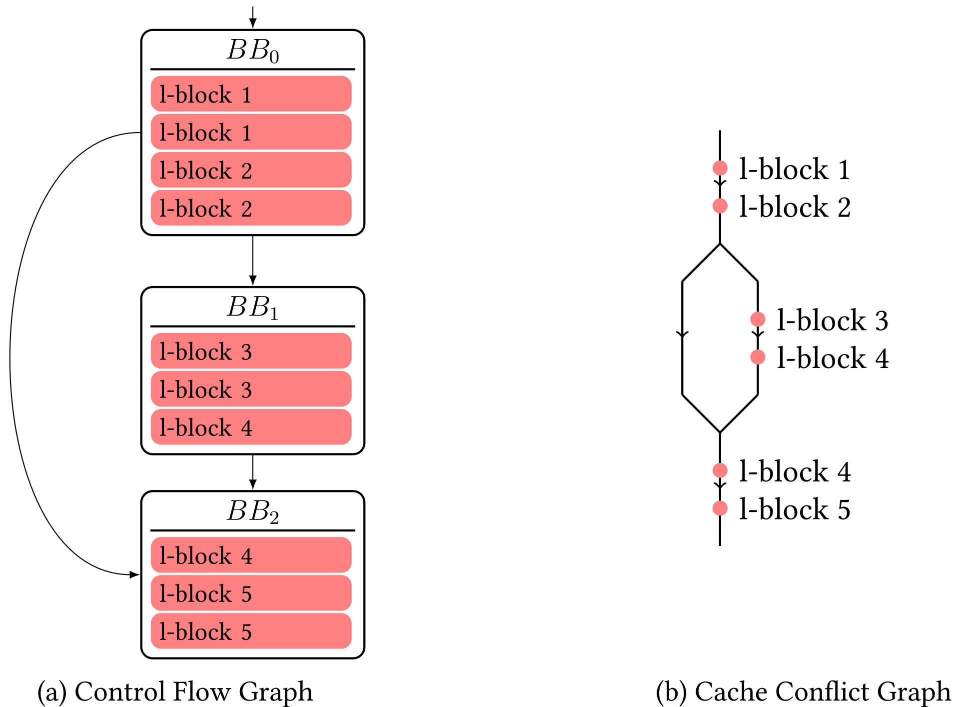
## 2.3 State of the Art in Cache Analysis

As mentioned in Chapter 1, it is interesting to statically predict the cache behavior, by classifying memory accesses into the ones that lead to cache hits and the ones that lead to cache misses. The aim of a cache analysis is to provide such kind of classification.

### 2.3.1 Cache Conflict Graph

A first observation that eases cache analysis is that cache sets usually work independently of each other and can be analyzed separately[6]. Thus, when analyzing one cache set, accesses other cache sets are simply ignored. This simplification is possible for many replacement policies, and one only needs to abstract a single cache set to perform the analysis. In this thesis, we focus on the LRU replacement policy, and we thus analyze cache sets one by one. A convenient structure for modeling a program in this case is the cache conflict graph (CCG) associated to the program. Each node of a CCG represents one or more consecutive accesses to a single memory block that

[6] This assumes instructions are fetched in order and that there is no speculative prefetch.




(a) Control Flow Graph

(b) Cache Conflict Graph

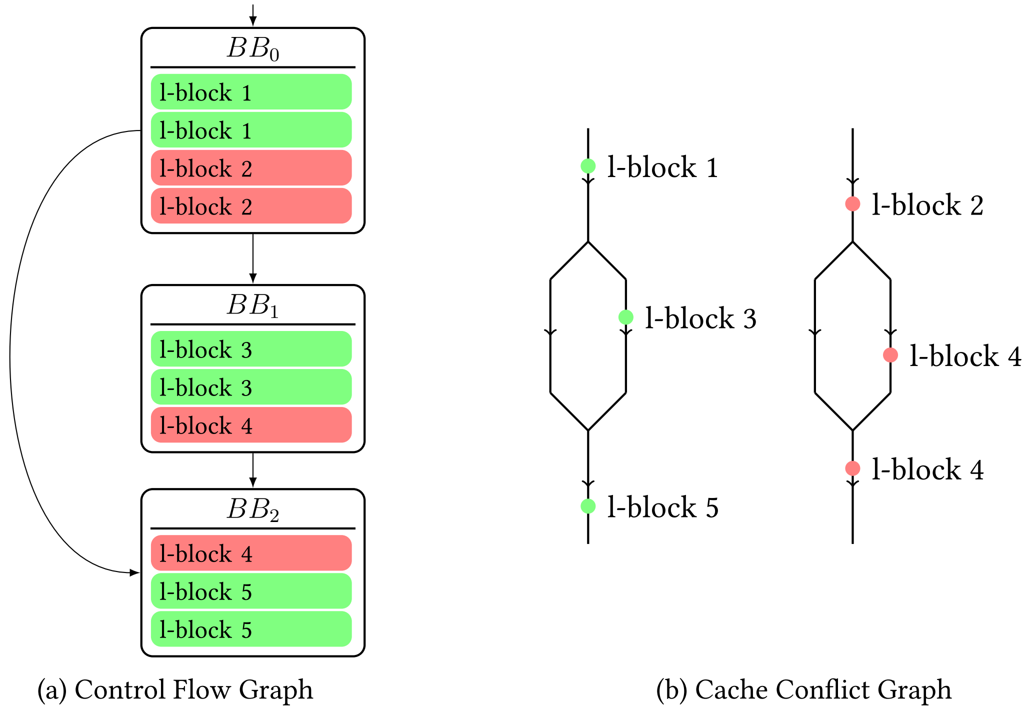
Figure 2.17 – A CFG and the associated CCG for a fully associative cache

maps the given cache set, whereas edges represent the program control-flow (similarly to edges in a CFG). For example, Figure 2.17b represents the CCG associated to the program whose CFG is represented on Figure 2.17a, in the case of a single cache set. The example program contains 10 instructions, spread into 5 memory blocks (one usually uses the term "line blocks", or "l-block" to distinguish them from basic blocks). The CCG is then obtained from the CFG by splitting the basic blocks at the boundaries of l-blocks. Note that a single l-block, like l-block 4 in our example, can be accessed in several basic blocks. In this case, it is simply referenced twice in the cache conflict graph.

Figure 2.18b shows the CCGs of the same program (on Figure 2.18a), in the case of an associative cache with two cache sets. The mapping from l-block to cache sets is given by colors. One CCG is then built for each cache set, and it does not represent references to other cache sets. Note that in the case of a non fully-associative cache, a CCG might not be strictly a graph, but rather a multi-graph. Indeed, if two consecutive references to the same cache set are far from one another, there might be several paths going from one to the other without any other reference to the same cache set in between. In other words, from a memory access, there might be several possible paths to reach the next access to the same cache set.

### 2.3.2 Analysis of LRU caches

The LRU replacement policy manages each cache set separately, and each of them is thus analyzed in isolation. An LRU cache set is completely defined if one knows the exact age of blocks in this cache set. Thus, one possible concrete domain to describe an LRU cache set is to use a function that maps each block to its age. When a block is cached, its age is between $0$ and $k-1$, where $k$ is the cache associativity. If not cached, one simply associates to the block an age value beyond the cache associativity (usually $k$ or $+\infty$) For example, after accessing blocks $a$, $b$, $c$, $d$, $e$ and $f$ (in this order), a 4-ways cache is described by a concrete value that maps $a$, $b$, $c$, $d$, $e$ and $f$ respectively




(a) Control Flow Graph

(b) Cache Conflict Graph

Figure 2.18 – A CFG and the associated CCG for a 2-sets associative cache

to 4, 4, 3, 2, 1 and 0. The most recently accessed block $f$ is mapped to 0, whereas evicted blocks $a$ and $b$ both map to $k = 4$.

Using *Blocks* to denote the set of memory blocks used by the program, the concrete domain is then:

$$D_{LRU} = Blocks \rightarrow \{0, \ldots, k\}$$

A convenient notation to represent a concrete cache set is to list the blocks cached according to their ages. The list has length $k$ and thus represent blocks which have an age between 0 and $k - 1$. Uncached blocks have age $k$ and are not represented. For example, the concrete state that maps block $a$ to age 0, block $b$ to age 1 and every other block to $k$ is represented by $[a, b, \bot, \bot]$. The symbol $\bot$ is used to represent an empty line, i.e. a line that does not contain a valid cache block because of a cache flush.

Using this notation, Figure 2.19 shows the fixpoint reached by the collecting semantics when analyzing the represented CCG. When a block is accessed, it is moved to the head of the list and blocks that were younger move by one position toward the tail. One can notice at the end of the program that after $k = 4$ consecutive accesses, the state of the cache converges to a single value. This particularity of the LRU replacement policy makes it highly predictable; it is not true for other replacement policies. Note also that we assumed an empty cache state at the beginning of the program, because representing all possible concrete values would lead to a huge set.

To formally define the concrete transformer associated to a basic block, we use an auxiliary function *update* that, given the current cache state $q$ and the block accessed $b$, computes the

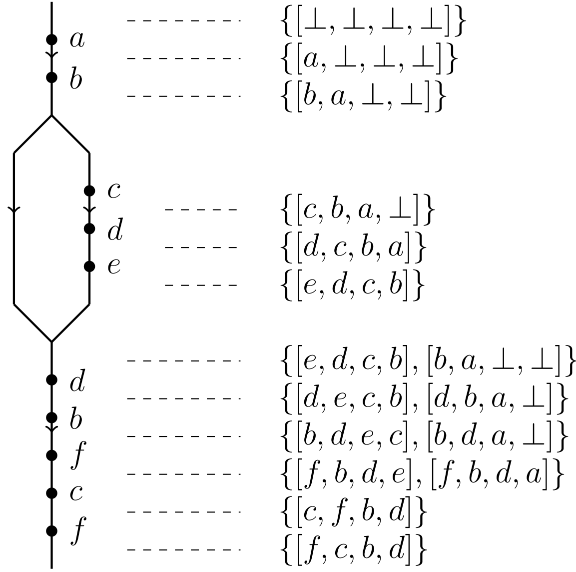


Figure 2.19 – LRU collecting semantics fixpoint

resulting cache state:

$$\begin{aligned} \mathit{update} : D_{LRU} \times \mathit{Blocks} &\to D_{LRU} \\ (q,b) &\mapsto q' \end{aligned}$$

$$\text{where } \forall b' \in \mathit{Blocks}, q'(b') = \begin{cases} 0 & \text{if } b = b' \\ q(b') & \text{if } b \neq b' \wedge q(b') > q(b) \\ q(b')+1 & \text{if } b \neq b' \wedge q(b') \leq q(b) \wedge q(b') < k \\ k & \text{otherwise} \end{cases}$$

This *update* function is the one used on Figure 2.19 to compute the new cache state obtained when accessing a block $b$. Four possible cases can be distinguished to compute the new age of a block $b'$:

- If $b'$ is the block accessed, then it is moved at the most recently used position, i.e. its new age is 0.
- If $b'$ is older than the accessed block, then $b'$ is not shifted during the update.
- If $b'$ is cached and younger than the accessed block, then $b'$ is shifted by one logical position toward the eviction.
- If $b'$ was already evicted, then it stays out of the cache

Note that this concrete domain allows representing impossible cache configurations by mapping several blocks to the same age (different from $k$). Indeed, a real cache line can only store one block at a time. This is however not a problem because a consistent concrete value (i.e that does not map two blocks to the same age except for age $k$) always leads to a consistent value when updated. The strictness of the comparison between ages of $b$ and $b'$ can thus be changed: the case $q(b) = q(b') \wedge q(b) < k$ never happens for a consistent concrete cache.

### 2.3.3 Ferdinand's May and Must analyses

Ferdinand's approach was the first cache analysis proposition relying on abstract interpretation. Its scalability and soundness make it one of the most widespread cache analyses in WCET estimation tools. The approach analyzes LRU caches and consists in two separated analyses:

- The must analysis is used to predict that some memory accesses always lead to a hit. This is done by computing an over-approximation of the age of all block stored in the cache.
- Conversely, the may analysis is used to predict misses, and it relies on a safe under-approximation of the age of all blocks to do so.

To ensure that some memory access results in a hit, the must analysis computes the "must abstract state" associated to a program point. This must cache state guarantees that some blocks must be cached at a given location by over-approximating their ages. Indeed, a memory block which has an age lower than $k$ when it is accessed is guaranteed to be a hit. The must analysis thus maintains upper-bounds on the ages of blocks using conservative transformers, and those bounds are then compared to the cache associativity to classify blocks.

The must abstract domain is composed of functions that map blocks to valid upper-bounds: $D_{Must} = Blocks \rightarrow \{0, \ldots, k\}$. A set of concrete cache states is then represented by a function that maps a block to the maximum age it may have in this set. For example, the set $\{[a, b, \bot, \bot], [b, c, a, \bot]\}$ is represented the abstract value $[\bot, b, a, \bot]$[7]. $a$, having age 0 in the first concrete cache state and age 2 in the second, is mapped to 2, which is a valid upper-bound. $c$ has ages $k$ and 1 in the concrete states and is thus mapped to $k$.

More formally, the must analysis is defined by the following Galois connection:

$$\begin{aligned}\alpha_{Must} : \mathcal{P}(D_{LRU}) &\rightarrow D_{Must}\\ Q &\mapsto \hat{q}_{Must}\\ \text{where } \forall b \in Blocks, \hat{q}_{Must}(b) &= \max_{q \in Q} q(b)\\ \gamma_{Must} : D_{Must} &\rightarrow \mathcal{P}(D_{LRU})\\ \hat{q}_{Must} &\mapsto \{q \in D_{LRU}, \forall b \in Blocks, q(b) \leq \hat{q}_{Must}(b)\}\end{aligned}$$

Note that contrarily to a concrete value, a must abstract value can map several blocks to the same upper-bound below $k$. We thus extend the previously used notation by bracing blocks that map to the same value. For example, $\alpha_{Must}(\{[a, d, b, \bot], [c, b, a, \bot]\}) = [\bot, \bot, \{a, b\}, \bot]$ maps both $a$ and $b$ to the bound 2.

In order to compute the abstract collecting semantics fixpoint, one should define the abstract join operator and the abstract transformers associated to basic blocks.

In the case of the join, one can simply use the maximum of the two given bounds: if a block $b$ is guaranteed to have age less than $x$ on one path and less than $y$ on the second path, then its age is necessarily less than $\max(x, y)$. This gives us the following join operator:

$$\begin{aligned}\sqcup_{Must} : D_{Must} \times D_{Must} &\rightarrow D_{Must}\\ (\hat{q}_{Must}, \hat{q}'_{Must}) &\mapsto \hat{q}''_{Must}\\ \text{where } \forall b \in Blocks, \hat{q}''_{Must}(b) &= \max(\hat{q}_{Must}(b), \hat{q}'_{Must}(b))\end{aligned}$$

[7]We reuse the same notation used to represent concrete cache states in Section 2.3.2 to represent abstract cache states.

This operator is trivially sound: $\gamma_{Must}(\hat{q}_{Must} \sqcup_{Must} \hat{q}'_{Must}) \supseteq \gamma_{Must}(\hat{q}_{Must}) \cup \gamma_{Must}(\hat{q}'_{Must})$.

As for the concrete transformers, the abstract transformers can be defined as the composition of an update function that expresses how a must abstract cache state is modified when accessing a block $b$. This $update_{Must}$ function looks like its concrete counterpart, but manipulates abstract values:

$$
\begin{aligned}
update_{Must} : D_{Must} \times Blocks &\to D_{Must} \\
(\hat{q}_{Must}, b) &\mapsto \hat{q}'_{Must}
\end{aligned}
$$

$$
\text{where } \forall b' \in Blocks, \hat{q}'_{Must}(b') = \begin{cases} 0 & \text{if } b = b' \\ \hat{q}_{Must}(b') & \text{if } b \neq b' \wedge \hat{q}_{Must}(b') \geq \hat{q}_{Must}(b) \\ \hat{q}_{Must}(b') + 1 & \text{if } b \neq b' \wedge \hat{q}_{Must}(b') < \hat{q}_{Must}(b) \wedge \hat{q}_{Must}(b') < k \\ k & \text{otherwise} \end{cases}
$$

This function shows four cases analogous to the concrete one, but reasoning about upper-bounds is more challenging. Intuitively, increasing the upper-bound by one on every access is safe, because the concrete age of a block can not increase faster. Thus, the tricky part is to derive a condition under which it is safe not to increase the upper-bound, as done in the second case. One can check its correctness as follows:

If $\hat{q}_{Must}(b')$ is greater than the concrete age of $b'$ then it is safe to keep it untouched because it would still hold even if $b'$ is shifted. We can thus focus on the case where $\hat{q}_{Must}(b')$ is a tight bound. Then, the condition $\hat{q}_{Must}(b') \geq \hat{q}_{Must}(b)$, together with the tightness of $\hat{q}_{Must}(b')$, ensures that $b$ is younger than $b'$ in the concretization. Thus, $b'$ is not be shifted when accessing $b$, and the previous bound still holds.

More formally, the soundness of the $update_{Must}$ function is expressed by the following lemma, which is proved in [AFMW96]:

$$
\forall \hat{q}_{Must} \in D_{Must}, \forall b \in Blocks, \gamma_{Must}(update_{Must}(\hat{q}_{Must}, b)) \supseteq \{update(q, b), q \in \gamma_{Must}(\hat{q}_{Must})\}
$$

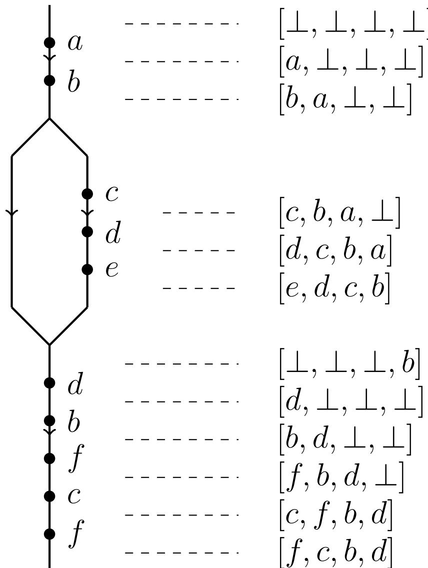



Figure 2.20 – Result of Ferdinand's Must analysis

Figure 2.20 shows how the described analysis works on the previous example of Figure 2.19. Three noticeable facts are illustrated in this example:

- Firstly, the bounds computed by the must analysis indeed cover all the concrete values obtained previously.
- Secondly, the analysis can ensure some accesses are always hits. For instance, the very last access to $f$ at the end of the program is guaranteed to be a hit, because $f$ is in the must cache when accessed.
- Finally, the analysis is incomplete and fails to classify some accesses. For example, the last access to $b$ is necessarily a hit (at most 3 distinguished blocks have been accessed since $b$ was loaded) but the must analysis is too coarse to classify it as Always-Hit. Indeed, $b$ is not in the must cache when accesses for the second time.

The may analysis is very similar to the must analysis. Instead of maintaining a safe upper-bound for each block, it uses a lower-bound. When this lower bound is equal to $k$, one knows for sure that the block is not cached. For example, the set of concrete values $\{[a, b, \bot, \bot], [b, c, a, \bot]\}$ is represented by the abstract value $[\{a, b\}, c, \bot, \bot]$.

Formally, the may analysis uses the following abstraction/concretization Galois connection:

$$
\begin{aligned}
\alpha_{May} : \mathcal{P}(D_{LRU}) &\to D_{May} \\
Q &\mapsto \hat{q}_{May} \\
\text{where } \forall b \in \mathit{Blocks}, \hat{q}_{May}(b) &= \min_{q \in Q} q(b) \\
\gamma_{May} : D_{May} &\to \mathcal{P}(D_{LRU}) \\
\hat{q}_{May} &\mapsto \{q \in D_{LRU}, \forall b \in \mathit{Blocks}, q(b) \geq \hat{q}_{May}(b)\}
\end{aligned}
$$

Concerning the abstract transformers, the may analysis looks like its must counterpart:

$$
\begin{aligned}
\mathit{update}_{May} : D_{May} \times \mathit{Blocks} &\to D_{May} \\
(\hat{q}_{May}, b) &\mapsto \hat{q}'_{May} \\
\text{where } \forall b \in \mathit{Blocks}, \hat{q}'_{May}(b') &= \begin{cases} 0 & \text{if } b = b' \\ \hat{q}_{May}(b') & \text{if } b \neq b' \wedge \hat{q}_{May}(b') > \hat{q}_{May}(b) \\ \hat{q}_{May}(b') + 1 & \text{if } b \neq b' \wedge \hat{q}_{May}(b') \leq \hat{q}_{May}(b) \wedge \hat{q}_{May}(b') < k \\ k & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that in the case of the may analysis, which computes lower bounds, a safe choice when updating a block not accessed could be to keep the previous bound unmodified. Contrarily to the must analysis, one has to be careful when increasing the lower-bound value. Again, the critical case is when the lower-bound on $b'$ is tight. Yet, in this case the condition $\hat{q}_{May}(b') \leq \hat{q}_{May}(b)$ guarantees that $b'$ is younger than $b$, and $\hat{q}_{May}(b')$ may be increased safely.

Unsurprisingly, the join operator is obtained by taking the minimum of the available bounds:

$$
\begin{aligned}
\sqcup_{May} : D_{May} \times D_{May} &\to D_{May} \\
(\hat{q}_{May}, \hat{q}'_{May}) &\mapsto \hat{q}''_{May} \\
\text{where } \forall b \in \mathit{Blocks}, \hat{q}''_{May}(b) &= \min(\hat{q}_{May}(b), \hat{q}'_{May}(b))
\end{aligned}
$$

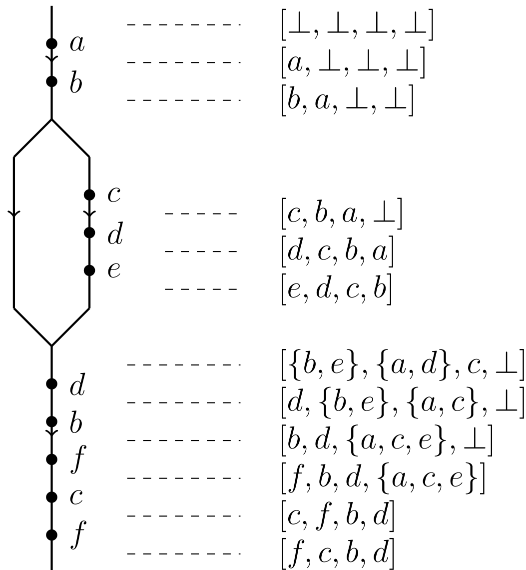



Figure 2.21 – Result of Ferdinand's May analysis

Figure 2.21 illustrates the may analysis on the previous example. As for the must analysis, one can notice that:

- The bounds computed by the may analysis cover the concrete collecting semantics.
- The analysis ensures some accesses, like the very first access to $a$, are always misses.
- Finally, the analysis is incomplete: the last access to $c$ is a miss but $c$ belongs to the may cache at the access location.

### 2.3.4 Persistence Analysis and Loop Unrolling

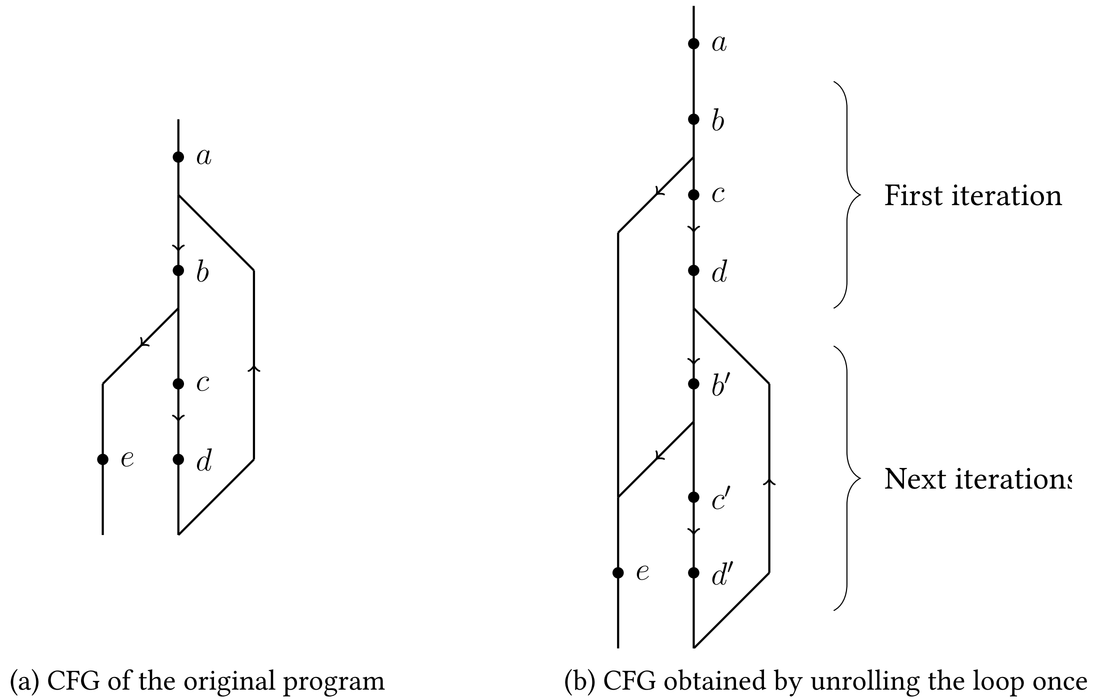
Program loops can be handled by may and must analyses without any change: backward edges are simply treated like any other edge. The fixpoint computation by chaotic iterations is still guaranteed to finish in presence of loops. Indeed, the abstract domain being finite (the number of blocks in the program is finite, and the cache associativity is fixed), it can not contain any infinite strictly increasing chain.

However, loop and if-then-else tests can introduce imprecision because of joins. This precision loss is mainly due to small loops (loops which body completely fits into the cache) that tend to behave differently in the first iteration of the loop. Indeed, blocks forming the loop are usually accessed for the first time in the first loop iteration, leading to misses. If the loop is small enough, they stay in the cache from one loop iteration to the next and all accesses beyond the first iteration are hits. May and must analyses are however not expressive enough to classify an access as a miss in the first iteration and as a hit afterward. If both hits and misses are possible, then the access is left unclassified by the analyses: it is in the may cache, but not in the must cache.

Some approaches have been specifically designed to tackle this problem. Among these approaches, one can cite the "loop unrolling" method and all the variation of persistence analyses.

Loop unrolling consists in modifying the CFG of the program by duplicating the nodes of the loops to distinguish the first iteration from the others [8]. This modification does not change the possible sequences of accesses the program might perform, but simply allows the may and must

[8] This unrolling inside an analyzer is different from unrolling the first iteration using a compiler prior to analyzing




(a) CFG of the original program

(b) CFG obtained by unrolling the loop once

Figure 2.22 – Modification of the CFG by unrolling a loop

analysis to treat the first occurrence of an access differently. An example of loop unrolling is shown on Figure 2.22. On the left side (Figure 2.22a), the original program is composed of five accesses, to blocks $a$, $b$, $c$, $d$ and $e$, and have a loop around accesses $b$, $c$ and $d$ ($b$ is the test, and $c$ and $d$ form the loop body). In this situation, starting from an empty initial cache, the usual may and must analyses would not be able to classify accesses to $b$, $c$ and $d$: they can not be classified as Always-hit because the first iteration leads to misses, and they can not be classified as Always-miss because of the remaining iterations. However, by unrolling the first iteration of the loop, one obtain the equivalent program of Figure 2.22b. On this new program, the may analysis classifies the first accesses to $b$, $c$ and $d$ as Always-miss (assuming an empty initial cache), whereas the must analysis classifies other accesses (noted with a prime) as Always-hit. This approach, which consists in virtually unrolling loops and inlining function calls, is described in [MAWF98].

Another family of approaches, called persistence analyses, can also be used to deal with loops in the case of LRU caches. Instead of classifying accesses as hits or misses, these analyses try to classify memory blocks as persistent or not. Intuitively, a block is said to be persistent if it can not be evicted from the cache once loaded. This information is particularly useful in the case of WCET estimation tools, because the miss penalty of a persistent block accessed in a loop can be considered only once. Contrarily to the may and must analyses which classify single accesses, persistence analyses consider the whole set of accesses to a memory block and try to bound the number of misses this set might suffer.

A naive approach consists in computing for each program point the set of memory blocks that have potentially been accessed since the beginning of the program: when an access is performed, the block accessed is added to the set of blocks already met. The size of the set obtained at a memory access location is then compared to the associativity to know if the block accessed is

the code. A compiler unrolling a loop will duplicate instructions, putting them at different addresses and thus possibly in different cache blocks. In contrast, the analyzer duplicates instructions (or simulates duplication by iteration techniques) while keeping them at the same address for purposes of instruction cache analysis.

persistent so far or not. By doing so, one can deduce that the accesses to blocks $a$, $b$, $c$ and $d$ of the example on Figure 2.22a can suffer at most one miss each for a cache of associativity $k = 4$. Indeed, the whole loop fits in the cache.

Obviously, such a naive approach where sets of potentially accessed blocks keep growing would fail on any real program. Thus, some approaches introduce a notion of scopes to focus on one loop at a time and reset the sets of reachable blocks. Other persistence analyses are based on the may and must analyses but use additional mechanisms to handle loops gracefully [BC08]. Many variations of this approach have been proposed, offering different precision/computation trade-off. See [Rei18] for an overview of existing persistence analyses.

The closest analysis to our work is the persistence analysis presented in [HJR11]. Indeed, it uses the notion of younger sets to classify memory blocks as persitent or not. The younger set of associated to a block $a$ is the set of blocks younger than $a$ at the given location. By knowing the younger set of a block, one knows exactly its age and when it will be evicted. By overapproximating the younger set associated to blocks, the approach described in [HJR11] is able to retrieve precise information about blocks to classify them as persistent.

### 2.3.5 Other replacement policies

Since the proposal of May and Must analyses for LRU caches, other replacement policies have been investigated. Approaches for FIFO [GR09, GYLY13], PLRU [GR10, GLBD14] and NMRU [GLYY14] have been proposed. Another possibility is also to rely on the usual may and must analyses to analyze a program under an LRU policy and to use the result to deduce the behavior of the program under another policy. This method, called relative competitiveness [Rei09], may be less precise than an analysis specifically designed for a given policy but can be used for all the policies previously mentioned.

As mentioned previously, this thesis focuses on the LRU replacement policy. Contrarily to the may and must analyses, which provide over and under-approximations of block ages, we propose exact analyses. However, we rely on the may and must analyses as an inexpensive precomputation, and use more precise analyses for blocks that are left unclassified and require more precise analyses.

### 2.3.6 Cache analysis by Model Checking

In addition to abstract interpretation approaches, one can find in literature cache analyses relying on model checking, as [LGY$^+$10]. These analyses consist in encoding both the program and the cache in a timed automaton as follows:

- The CFG of the program is encoded by defining a variable *state* describing the current basic block being executed. One then defines the transition relation such that *next(state)* can take the value of any successor of the basic block given by *state*.
- For each loop, one variable is added to count the number of iteration. This variable is set to 0 when entering the loop, and is incremented each time the backedge is taken. Finally, a guard avoid the loop body to be reached once the loop bound is reached.
- A variable of array type is used to model the cache content. The transition relation is modified to encode the replacement policy when an access is reached.

[optional] A variable *wcet* is initialized to 0. Each time a basic block is executed, its WCET is added to this variable. In case the cache model predict a miss, a penalty is applied to the *wcet* variable.

One can then check the satisfiability of the LTL formula $G((state = exit) \Rightarrow (wcet < N))$ for different values of $N$. The wcet is then found by binary search on $N$. Some approaches, like [CR11] estimate a WCET by encoding the program into an Integer Linear Problem, but use a model checker to catch cache conflicts not found by usual abstract interpretation. The main idea of this approach is to extract the location of accesses from the binary to encode them into the program source (at C level). Then a model checker (like CBMC in this case) is used to find the cache conflict while taking the program semantics into account. This fine-grained information is then encoded into the ILP.

# Chapter 3

# Cache Analysis Complexity

Performing computation requires resources. Mainly, two resources are usually considered when evaluating the cost of an algorithm: memory consumption, and execution time. By estimating the space (i.e. memory) and time required to compute the solution to a given problem, one can evaluate how hard the problem is. More precisely, the complexity of a given problem is defined as the cost of the best algorithm solving this problem, in the worst-case.

To our knowledge, problems related to the analysis of caches have not been studied from the complexity point of view. In this chapter, we thus look at the theoretical complexity of classifying memory accesses as hits or misses for several replacement policies.

## 3.1 Background

### Deterministic and Non-Deterministic Computation Model

Evaluating the *time* or *space* consumption of an algorithm requires to fix on the machine that executes this algorithm. The usual choice is to assume the algorithm is running on a Turing machine. The time consumption of an algorithm is then defined as the number of transitions taken by the machine before reaching the problem solution, whereas the space consumption characterizes the number of tape cells used during the execution. However, these time and space measurements obviously depend on the input of the machine, i.e. the instance of the problem. The complexity of an algorithm thus expresses the relation between the time or space required to solve the problem and the size of the input, i.e. the number of bits needed to encode the input. More precisely, the complexity of an algorithm is the function that, given the size of the problem instance, gives the cost of the algorithm execution. Because the exact cost is usually difficult to express, one uses an asymptotic upper-bound of the worst-case cost when the input size tend to infinity (see [AB09, Chap. 1] for more formal definitions).

Commonly, problems solvable by an algorithm which time complexity is bounded by a polynomial when running on a deterministic Turing machine are considered tractable. In this chapter, we focus on decision problems, i.e. problems that can be posed as a yes/no questions, such as "Does the given graph contains a cycle?". In particular, we note P the set of decision problem that can be answered in *polynomial time* by a deterministic Turing machine. These problems are considered as easy to solve, because increasing linearly the size of the instance only lead to a polynomial increase in the number of computation steps to solve it. On the other hand, algorithm which time complexity is exponential (or more) are considered intractable. Note that there exist algorithms whose complexity is neither polynomial nor exponential.

Similarly to the P class of problems, one can define the NP class of decision problems that

can be solved in polynomial time on a *nondeterministic* Turing machine[1]. For a given state of the machine and a given symbol read on the tape, a deterministic Turing machine always behaves the same, i.e. only one next state is reachable. Contrarily, a non-deterministic Turing machine might have several transitions available for a given symbol and a given state. When this happen, the transition taken is then the one that "get closer" to the problem solution. In other word, one can consider that a non-deterministic machine is "lucky"[2] and always takes the best transition available. Alternatively, one can consider that all transition are taken at the same time in parallel (and that the tape is copied as many time as needed), and that all execution continue concurrently. The NP class can also be defined as the set of problems which solution can be *verified in polynomial time* by a deterministic Turing machine.

A typical example of problem in NP is the SAT problem (boolean satisfiability problem) that consists in finding a valuation of boolean variables that makes a propositional formula true. Indeed, given a valuation of variables, checking the truth value of a formula is simply done by a bottom-up exploration of the formula syntax tree. A consequence of this definition is that every problem in P is in NP, because deterministic Turing machines are a special case of non-deterministic Turing machines. The problem "Is P equal to NP?" is still an open question.

Using the notion of NP problems, one can define the class of NP-hard problems, that are "at least as hard as any problem in NP". More formally, a decision problem $A$ is NP-hard if given a polynomial time algorithm $f_A$ solving this problem, one can build for any problem $B$ in NP a polynomial procedure $f_B$ that solves $B$ in polynomial time using $f_A$. $f_A$ is called an *oracle* and the process of turning $f_A$ in $f_B$ is called a *reduction*. $A$ is then NP-hard if any problem $B$ in NP can be reduced to $A$ in polynomial time.

Finally, NP-complete problems are problems that are both NP and NP-hard. They are thus the "hardest" problems in NP, because they are at least as hard as any other NP problem. By definition, finding a polynomial procedure that solves a NP-complete problem would imply that any NP problem can be solved in polynomial time and would thus prove that P equals NP. SAT is the prototypical example of a NP-complete problem (this is the result of Cook-Levin theorem). Some variation have also been proven to be NP-complete. Noticeably, the problem 3-SAT is also NP-complete. 3-SAT is similar to SAT, but boolean formula are required to be in conjunctive normal form (i.e. a conjunction of disjunctive clauses), where each clause contains at most three litterals.

### Space complexity

When considering spatial complexity, one can define classes similar to their time complexity counterparts P, NP, NP-hard and NP-complete. PSPACE defines the set of decision problems that can be solved by a deterministic machine that uses a polynomial number of tape cells in the worst case. NPSPACE is similar but allows the use of an non-deterministic Turing machine to solve the problem. According to Savitch's theorem [Sav70], PSPACE = NPSPACE, i.e. a problem that can be solved by using polynomial memory space on a non-deterministic can also be solved using polynomial memory space on a deterministic machine. Moreover, doing so will only increases the time needed by a polynomial factor. In the following, we thus use the term PSPACE to denote the NPSPACE complexity class.

PSPACE-hard then denotes the set of problems that are "at least as hard" as any PSPACE problem. More precisely, a decision procedure that solves a PSPACE-complete problem could be

[1] Note that such machines are theoretical computation model and do not exist yet in practice.

[2] The term of "angelic machine" is sometimes used, in opposition to "demoniac machine" that takes the worst possible transition.

used to build a solution to any PSPACE problem. Unsurprisingly, the PSPACE-complete class then denotes the intersection of PSPACE and PSPACE-hard problem sets.

Note also that at most one tape cell is written each time a transition is taken by the Turing machine. Thus, only a polynomial number of cells can be written in polynomial time, and we have NP $\subseteq$ PSPACE. Similarly, PSPACE-hard $\subseteq$ NP-hard.

Finally, the current knowledge about the mentioned complexity classes can be summarized by Figure 3.1. We have P $\subseteq$ NP $\subseteq$ PSPACE $=$ NPSPACE and, as a consequence, PSPACE-Hard $\subseteq$ NP-Hard.

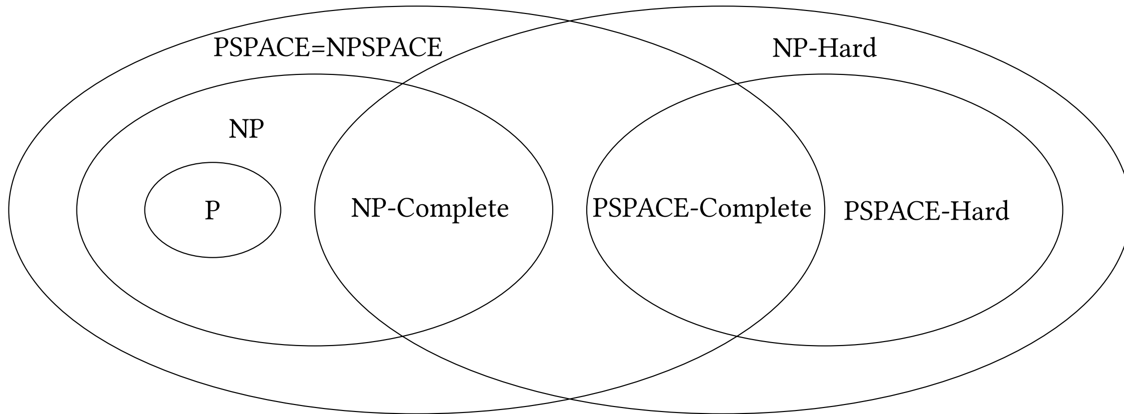



Figure 3.1 – Relations between complexity classes

As an example of PSPACE-complete problem, we introduce the reachability problem for boolean register machine:

**Definition 1.** *A* Boolean register machine *is defined by a number $r$ of registers and a directed (multi-)graph with an initial vertex and a final vertex, with edges adorned by instructions of the form:*

**Guard** $v_i = b$ *where* $1 \leq i \leq r$ *and* $b \in \{\mathbf{f}, \mathbf{t}\}$,

**Assignment** $v_i := b$ *where* $1 \leq i \leq r$ *and* $b \in \{\mathbf{f}, \mathbf{t}\}$.

*The* register state *is a vector of $r$ Booleans. An edge with a guard $v_i = b$ may be taken only if the $i$-th register contains $b$; the register state is unchanged. The register state after the execution of an edge with an assignment $v_i := b$ is the same as the preceding internal state except that the $i$-th register now contains $b$.*

*The* reachability problem *for such a system is the existence of a valid execution starting in the initial vertex with all registers equal to $0$, and leading to the final vertex.*

**Lemma 2.** *The reachability problem for Boolean register machines is PSPACE-complete.*

*Proof.* Such a machine is easily simulated by a polynomial-space nondeterministic Turing machine; by Savitch's theorem the reachability problem is thus in PSPACE.

Any Turing machine using space $P(|x|)$ on input $x$ can be simulated by a Boolean register machine with $O(P(|x|))$ registers, encoding the tape of the Turing machine, and a number of transitions polynomial in the description of the Turing machine. □

This last lemma is extensively used in the following section to prove the PSPACE-hardness of some cache-related decision problems. Finally, the following lemma will also be used to study the theoretical complexity of cache analysis problems when the program model does not have any cycles.

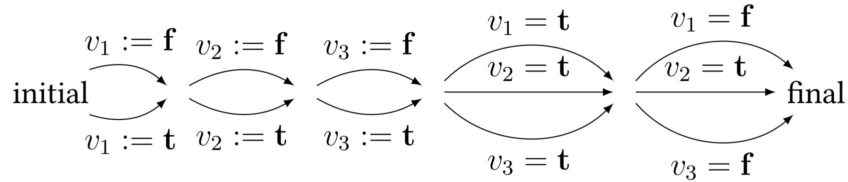



Figure 3.2 – Reduction of CNF-SAT over 3 unknowns with clauses $\{v_1 \vee v_2 \vee v_3, \bar{v_1} \vee v_2 \vee \bar{v_3}\}$ to a Boolean 3-register machine

**Lemma 3.** *The reachability problem for Boolean register machines with acyclic control-flow is NP-complete.*

*Proof.* A path from initial to final vertices, along with register values, may be guessed nondeterministically, then checked in polynomial time, thus reachability is in NP.

Any CNF-SAT problem with $r$ Boolean unknowns may be encoded as a Boolean $r$-register machine as follows: a sequence of $r$ disjunctions between $v_i := \mathbf{t}$ and $v_i := \mathbf{f}$ for each variable $i$, and then for each clause $v_{i_1^+} \vee \cdots \vee v_{i_{n^+}^+} \vee v_{i_1^-} \vee \cdots \vee v_{i_{n^-}^-}$, a disjunction between edges $v_{i_i^+} = \mathbf{t}$ for all $1 \leq i \leq n^+$ and edges $v_{i_i^-} = \mathbf{f}$ for all $1 \leq i \leq n^-$ (see Figure 3.2). □

## 3.2 Complexity of Replacement Policies

As in the remaining of this thesis, we consider that the control-flow graph (and thus the cache conflict graph) carries only identifiers of memory blocks to be accessed, abstracting away the data that are read or written, as well as arithmetic operations and guards. Therefore, we take into account executions that cannot take place on the real system. This is the same setting used by many static analyses for cache properties. Some more precise static analyses attempt to discard some infeasible executions — e.g. an execution with guards $x < 0$ and $x > 0$ with no intervening write to $x$ is infeasible. In general, however, this entails deciding the reachability of program locations. This problem is undecidable if the program operates over unbounded integers, and already PSPACE-complete if the program operates on a finite vector of bits. Clearly we cannot use such a setting to isolate the contribution of the cache analysis itself.

In this chapter, we explore the questions of cache analysis efficiency as decision problems. As mentioned previsouly, cache analysis is interested in classifying accesses as *Always-Hit* and *Always-Miss*. Informally, the *Always-Hit* problem can be stated as "Does any path in the given graph lead to hit at the given access?". This can be reformulated in the equivalent *Exist-Miss* problem: "Is there any path in the given graph that lead to a miss at the given access?". Similarly, this section looks at the complexity of the *Exist-Hit* problem rather than the *Always-Miss* equivalent version[3].

**Definition 4** (Exist-Hit Problem)**.** *The* exist-hit *problem is, for a given replacement policy:*

[3] This formulation is chosen to avoid the manipulation of complementary problem that are in the co-NP complexity class.

| | |
|---|---|
| *Inputs* | *a cache conflict graph with nodes adorned with memory block* |
| | *a starting node $S$ in the graph* |
| | *a final location $F$ in the graph* |
| | *the cache* associativity *(*number of ways $k$*), in unary* |
| | *a memory block $a$* |
| *Outputs* | *a Boolean: is there an execution trace from $S$ to $F$, starting with an empty initial cache and ending with a cache containing $a$?* |

**Definition 5** (Exist-Miss Problem)**.** *The* exist-miss *problem is defined as above but with an ending state* not *containing $a$.*

**Remark 6.** *In definitions 4 and 5, it does not matter if the associativity is specified in unary or binary. An associativity larger than the number of blocks in the program always produces hits once the block is loaded, thus the problems becomes trivial. This also applies to FIFO, PLRU, NMRU caches and, more generally, to any cache analysis problem starting from an empty cache with a replacement policy that never evicts cache blocks as long as there is a free cache line.*

We shall also study the variant of this problem where the initial cache contents are arbitrary:

**Definition 7.** *The exist-hit (respectively, exist-miss) problem with arbitrary initial state contents is defined as above, except that the output is "are there a legal initial cache state $q_0$ and an execution trace from $S$ to $F$, starting in cache state $q_0$ and ending with a cache containing (respectively, not containing) $a$?".*

We shall here prove that

- for policies *LRU*, *FIFO*, *pseudo-RR*, *PLRU*, and *NMRU*, the exist-hit and exist-miss problems are *NP-complete* for *acyclic* cache conflict graphs (CCG);
- for *LRU*, these problems are still *NP-complete* for *cyclic* CCGs;
- for *PLRU*, *FIFO*, *pseudo-RR*, *PLRU*, and *NMRU*, these problems are *PSPACE-complete* for *cyclic* CCGs;
- for *LRU*, *FIFO*, *pseudo-RR*, and *PLRU*, the above results extend to exist-miss and exist-hit problems from an *arbitrary starting cache state.*

Under the usual conjecture that PSPACE-complete problems are not in NP, this may justify why analyzing properties of FIFO, PLRU and NMRU caches is harder than for LRU.

Real-life CPU cache systems are generally complex (multiple levels of caches) and poorly documented (often, the only information about replacement policies is by reverse engineering). For our complexity-theoretical analyses we need simple models with clear mathematical definitions; thus we consider only one level of cache, and only one cache set per cache [4].

[4]A real cache system is composed of a large number of "cache sets" (see Section 2.1): a memory block may fit in only one cache set depending on its address, and the replacement policy applies only within a given cache set. For all commonly found cache replacement policies except pseudo-round-robin, and disregarding complex CPU pipelines, this means that the cache sets operate completely independently, each seeing only memory blocks that map to it; each can be analyzed independently. It is therefore very natural to consider the complexity of analysis over one single cache set, as we do in this chapter.

### 3.2.1 Fixed associativity

In a given hardware cache, the associativity is fixed, typically $k = 2, 4, 8, 12$ or $16$. It thus makes sense to study cache analysis complexity for fixed associativity. However, such analysis can always be done by explicit-state model-checking (enumeration of reachable states) in polynomial time:

**Theorem 8.** *Let us assume here that the associativity $k$ is fixed, as well as the replacement policy (among those cited in this article). Then exist-hit and exist-miss properties can be checked in polynomial time, more precisely in $O(|G|^{k+1})$ where $|G|$ is the size of the CCG.*

*Proof.* Let $(V, E)$ be the CCG; its size is $|G| = |V| + |E|$. Let $Blocks$ the set of possible cache blocks. Without loss of generality, for all policies discussed in this article, the only blocks that matter in $Blocks$ are those that are initially in the cache (at most $k$) and those that are found on the control edges. Let us call the set of those blocks $Blocks'$; $|Blocks''| \leq |V| + k$.

The state of the cache then consists in $k$ blocks chosen among $|Blocks'|$ possible ones, plus possibly some additional information that depends on the replacement policy (e.g. the indication that a line is empty); say $b$ bits per way. The number of possible cache states is thus $(2^b|Blocks'|)^k$.

Let us now consider the finite automaton whose states are pairs $(A, q)$ where $A$ is a node in the CCG and $q$ is the cache state, with the transition relation $(A, q) \rightarrow (A', q')$ meaning that the processor moves in one step from access $A$ with cache state $q$ to control node $A'$ with cache state $q'$. The number of states of this automaton is $|V|.(2^b|B'|)^k$, which is bounded by $|G|.(|G|+k)^k.2^{bk}$, that is, $O(|G|^{k+1})$.

Exist-miss and exist-hit properties amount to checking that certain states are reachable in this automaton. This can be achieved by enumerating all reachable states of the automaton, which can be done in linear time in the size of the automaton. □

It is an open question whether it is possible to find algorithms that are provably substantially better in the worst-case than this brute-force enumeration. Also, would it be possible to separate replacement policies according to their growth with respect to associativity? It is however unlikely that strong results of the kind "PLRU analysis needs at least $K.|G|^k$ operations in the worst case" will appear soon, because they imply $\text{P} \neq \text{NP}$ or $\text{P} \neq \text{PSPACE}$.

**Theorem 9.** *Consider a policy among PLRU, FIFO, pseudo-RR (with known or unknown initial state) or NMRU with known initial state (respectively, LRU), and a problem among exist-miss and exist-hit. Assume $(H)$: for this policy, for any algorithm $\mathcal{A}$ that decides this problem on this policy, and any associativity $k$, there exist $K(k)$ and $e(k)$ such that for all $g_0$ there exists $g(k, g_0) \geq g_0$ such that the worst-case complexity of $\mathcal{A}$ on graphs of size $g$ is at least $K(k).g(k, g_0)^{e(k)}$. Assume also $e(k) \rightarrow \infty$ as $k \rightarrow \infty$, then P is strictly included in PSPACE (respectively, NP).*

*Proof.* Suppose $(H')$: there exists a polynomial-time algorithm $\mathcal{A}$ solving the analysis problem for arbitrary associativity, meaning that there exist a constant $K'$ and an exponent $e'$ such that $\mathcal{A}'$ takes time at most $K'.(k + g)^{e'}$ on a graph of size $g$ for associativity $k$.

Let $k$ be an associativity. From $(H)$ there is a strictly ascending sequence $g_m$ such that the worst-case complexity of $\mathcal{A}$ on graphs of size $g_m$ is at least $K(k).g_m^{e(k)}$. From $(H')$, $K(k).g_m^{e(k)} \leq K'.(k + g_m)^{e'}$. When $g_m \rightarrow \infty$ this is possible only if $e(k) \leq e'$.

Since $e(k) \rightarrow \infty$ as $k \rightarrow \infty$, the above is absurd. Thus there is no polynomial-time algorithm $\mathcal{A}$ for solving the analysis problem for the given policy. We prove later in this paper that these analysis problems are PSPACE-complete for PLRU, FIFO, NMRU, pseudo-RR, and NP-complete for LRU; the result follows. □

### 3.2.2 LRU

The "Least Recently Used" (LRU) replacement policy is simple and intuitive: the memory block least recently used is evicted when a cache miss occurs (see 2.1 for more details). More precisely, the state of an LRU cache with associativity $k$ is a word of length at most $k$ over the alphabet of cache blocks, composed of pairwise distinct letters; an empty cache is defined by the empty word. When an access is made to a block $a$, if it belongs to the word (*hit*), then this letter is removed from the word and prepended to the word. If it does not belong to the word (*miss*), and the length of the word is less than $k$, then $a$ is prepended to the word; otherwise, the length of the word is exactly $k$ — the last letter of the word is discarded and $a$ is prepended.

#### Fundamental properties

Our LRU analyses, as well as all our results on LRU in this thesis, are based on the following easy, but fundamental, property of LRU caches:

**Proposition 10.** *After an execution path starting from an empty cache, a block $a$ is in the cache if and only if there has been at least one access to $a$ along that path and the number of distinct blocks accessed since the last access to $a$ is at most $k-1$.*

**Example 11.** *Assume a $4$-way cache, initially empty. After the sequence of accesses $bcabdcdb$, block $a$ is in the cache because $bdcdb$ contains only $3$ distinct blocks $b, c, d$. In contrast, after the sequence $bcabdceb$, block $a$ is no longer in the cache because $bdceb$ contains $4$ distinct blocks $b, c, d, e$.*

#### Exist-Hit

In the following, we show that the *exit-hit* problem is *NP-complete* (i.e. both in *NP* and *NP-hard*) for the LRU policy. To prove that *exist-hit* is in *NP-hard*, we show that it is at least as hard as any *CNF-SAT* problem. More precisely, given a *CNF-SAT* problem, we show how to encode it in a *exist-hit* problem such that any cache analyzer that would solve the *exist-hit* problem could be turned into a SAT-solver. This encoding roughly consists in two parts: the first one is used to assign a value to the problem variables, and the second one is used to check whether the chosen values solve the given formula.

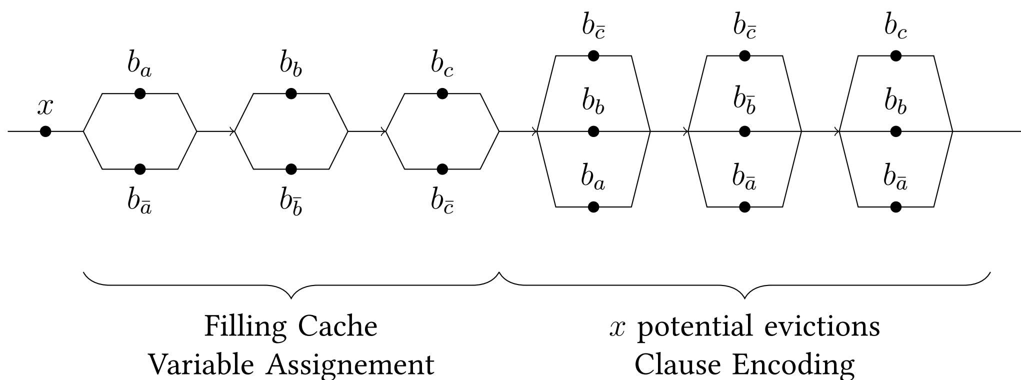



Figure 3.3 – There is a path from *init* to *end* with at most $3$ different accessed blocks if and only if the formula $(\bar{c} \vee b \vee a) \wedge (\bar{c} \vee \bar{b} \vee \bar{a}) \wedge (c \vee b \vee \bar{a})$ has a model. Thus, for a LRU cache with associativity $k = 4$, there is an execution from *start* to *end* ending in a cache state containing $x$ if and only if this formula has a model.

**Theorem 12.** *The exist-hit problem is NP-complete for LRU and acyclic CCGs.*

*Proof.* The problem is in NP: a path may be chosen nondeterministically then checked in polynomial time.

Now consider the following reduction from CNF-SAT (see Figure 3.3 for an example). Let $n_V$ be the number of variables in the SAT problem. We set the cache associativity to $k = n_V + 1$. With each variable $v$ in the SAT problem we associate two cache blocks $b_v$ and $b_{\bar{v}}$. The CCG we build starts with an access to fresh block $x$. Then, we happen two sequences of switches:

- For each variable $v$ in the SAT problem, a switch between the two blocks associated to $v$ and $\bar{v}$ respectively. On Figure 3.3, this sequence of switches is on the left. When executing this sequence, exactly $n_V$ blocks are loaded in the cache. More precisely, for each variable $v$, either $b_v$ or $b_{\bar{v}}$ is loaded. Thus, at the end of the sequence, the cache contains $x$, and one block for each variable. The cache is thus full, and the next block that will be evicted is $x$.

- For each clause in the SAT problem, a switch between accesses to the blocks associated to the literals in the clause. On Figure 3.3, this sequence of switches is on the right. Consider for instance the first switch between $b_{\bar{c}}$, $b_b$ and $b_a$. Because of the first sequence of switches, any cache miss leads to the eviction of the block $x$. If $b_{\bar{c}}$ has been loaded, then $b_{\bar{c}}$ can be accessed again to reach the next switch without evicting $x$ (which keeps the least-recently used position). Otherwise, one can go through $b_a$ or $b_b$ (provided one of them has been loaded before). If none of the three blocks have been loaded during the first sequence of switches, $x$ will be evicted.

Each path through the sequence of switches with at most $n_V$ different blocks corresponds to a SAT valid assignment, and conversely.

Finaly, there exists an execution such that at *end* the cache contains $x$ if and only if there exists a SAT valid assignment. □

The objection can be made that the reduction in this proof produces cache conflict graphs in which the same block occurs an arbitrary number of times — the number of times the corresponding literal occurs in the CNF-SAT problem, plus one. This may be appropriate for a data cache (the same data may be accessed many times within a loop-free program) but is unrealistic for an instruction cache[5]: a given cache block may not overlap with arbitrarily many basic blocks in the machine code [6]. However, we can refine the preceding result to account for this criticism.

**Theorem 13.** *The exist-hit problem is NP-complete for LRU for acyclic CCGs, even when the same cache block is accessed no more than thrice.*

*Proof.* We use the same reduction as in Theorem 12, but from a CNF-SAT problem where each literal occurs at most twice, as per the following lemma. □

**Lemma 14.** *CNF-SAT is NP-hard even when restricted to sets of clauses where the same literal occurs at most twice, the same variable exactly thrice.* [7]*.*

[5]Unless procedure calls are "inlined" in the graph.

[6]Consider a cache with 64-byte cache lines, as typical in x86 processors. In order for several basic blocks to overlap with that cache line, each, except perhaps the last one, must end with a branch instruction, which, in the shortest case, takes 2 bytes. No more than 32 basic blocks can overlap this cache line, and this upper bound is achieved by highly unrealistic programs.

[7]We thank Pálvölgyi Dömötör for pointing out to us that this restriction is still NP-hard.

*Proof.* In the set of clauses, rename each occurrence of the same variable $v_i$ as a different variable name $v_{i,j}$, then add clauses $v_{i,1} \Rightarrow v_{i,2}$, $v_{i,2} \Rightarrow v_{i,3}$, …, $v_{i,n-1} \Rightarrow v_{i,n}$, $v_{i,n} \Rightarrow v_{i,1}$ to establish logical equivalence between all renamings. Each literal now occurs once or twice, each variable thrice. Each model of the original formula corresponds to a model of the renamed formula, and conversely. □

**Remark 15.** *The exist-hit problem is easy when the same cache block occurs only once in the graph. Assume that the aim is to test whether there exists an execution leading to a cache containing $x$ at the final location $end$. Either there exists one reachable access $A$ to $x$ in the CCG, or there is none (in the latter case, $x$ cannot be in the cache at location $end$). Then there exists an execution leading to a cache state containing $x$ at location $end$ if and only if there exists a path of length at most $k-1$ between $A$ and $end$, which may be tested for instance by breadth-first traversal. The complexity question remains opens when cache blocks occur at most twice.*

We proved that the *exist-hit* problem is *NP-complete* for acyclic CCGs. In the following, we thus investigate the case of arbitrary CCGs. Because the acyclic case is a special case of graph, the problem is still *NP-hard* (the same reduction can be used). However, by dropping the acyclic constraint, the problem might not be in *NP* anymore. In particular, one can not guess nondeterministically a path and check in polynomial time if it leads to a miss, because cycles can create arbitrarily long paths. To show that the *exist-hit* problem remains in *NP* in case of cyclic CCG, we thus prove that if there exists a path leading to hit, then there is at least one "short" path also leading to hit, where "short" stands for "verifiable in polynomial time". To show this, we reason about the set of blocks loaded along a path. We thus introduce the following notation:

**Definition 16.** *The* content *of a path $\pi$, denoted by $c(\pi)$, is the set of blocks accessed along this path.*

**Theorem 17.** *The exist-hit problem is still in NP for LRU when the graph may be cyclic.*

*Proof.* Consider a path $\pi$ from $start$ to $end$ such that the cache contains $a$ when $end$ is reached. Let $\pi_i$ the last access to $a$ in $\pi$. We then note $\pi^1$ and $\pi^2$ the paths obtained by splitting $\pi$ at $\pi_i$. More precisely, $\pi^1 = \pi_0 \dots \pi_i$ and $\pi^2 = \pi_{i+1} \dots F$. Because $\pi$ leads to a hit at $end$, we have $|c(\pi^2)| < k$. By removing all cycles from $\pi^1$ and $\pi^2$, we obtain $\pi^{1'}$ and $\pi^{2'}$ of lengths at most $|Access|$ (the number of accesses in the CCG) and such that $c(\pi^{2'}) \subseteq c(\pi^2)$. Then, the path $\pi' = \pi^{1'}.\pi^{2'}$ leads to a hit and has length at most $2|Access|$. Thus, the nondeterministic search for a witness hit path may be restricted to simple paths of length at most $2|Access|$, which ensures membership in NP. □

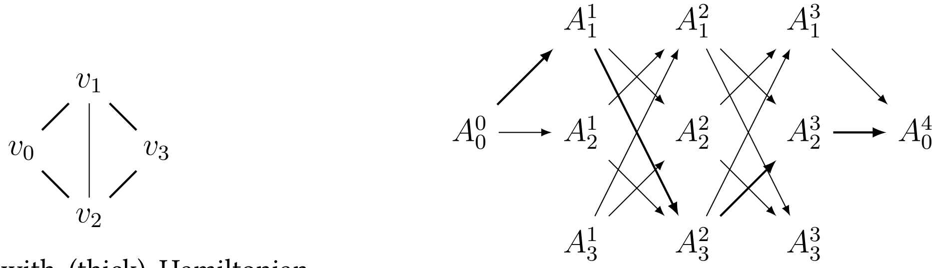
### Exist-Miss

**Theorem 18.** *The exist-miss problem is NP-complete for LRU for acyclic CCGs.*

*Proof.* Obviously, the problem is in NP: a path may be chosen nondeterministically, then checked in polynomial time.

We reduce the Hamiltonian circuit problem to the exist-miss problem (see Figure 3.4 for an example). Let $(V, E)$ be a graph, let $n = |V|$, $v = \{v_0, \dots, v_{n-1}\}$ (the ordering is arbitrary). Let us construct an acyclic CCG $G$ suitable for cache analysis as follows:

- two accesses $A_0^0$ and $A_0^n$ to a block $b_0$ associated to the vertex $v_0$.
- for each variable $v_i$, $i \geq 1$, $|V| - 1$ accesses $A_i^j$, $1 \leq j < n$, to the block $b_i$ associated to $v_i$. (this arranges these accesses in layers indexed by $j$)




(a) Graph with (thick) Hamiltonian cycle

(b) Acyclic CCG obtained by the reduction. Blocks accessed are not shown; the path corresponding to the Hamiltonian cycle is highlighted with thick lines.

Figure 3.4 – Reduction from Theorem 18 from the Hamiltonian cycle problem to the exist-miss problem for LRU caches.

- for each pair $A_i^j$, $A_{i'}^{j+1}$ of accesses in consecutive layers, an edge if and only if there is an edge $(v_i, v_i')$ in $E$.

There is a Hamiltonian circuit in $(V, E)$ if and only if there is a path in $G$ from $A_0^0$ to $A_0^n$ such that no block (except $b_{v_0}$) is accessed twice; thus if and only if there exists a path from $A_0^0$ to $A_0^n$ with at least $n$ distinct accessed blocks. Then, there exists a trace such that $b_0$ is not in the cache at $A_0^n$ if and only if this Hamiltonian circuit exists. □

The proof of Theorem 17 does not carry over to the exist-miss case. By removing cycles, we ensure the existence of a path with at most as many distinct blocks accessed, which is desired to show the existence of a hit. However, in the case of a miss we need to guarantee there are enough distinct blocks to lead to a miss. If we remove cycles, we may remove blocks that are necessary to lead a miss. We thus need a different proof to show that the *exist-miss* problem is still in *NP* for cyclic CCGs.

**Lemma 19.** *Let $G$ be a CCG with at most $N = |Blocks|$ blocks, and let $A_1, A_2$ be two accesses in $G$. From any path from $A_1$ to $A_2$ we can extract a path from $A_1$ to $A_2$ with the same contents and length at most $|Access|^2$.*

*Proof.* Consider a path $\pi$ from $A_1$ to $A_2$. $\pi$ can be segmented into sub-paths $\pi_1, \ldots, \pi_m$, each beginning with the first occurrence of a new block not present in previous sub-paths.

Each sub-path $\pi_i$ consists of an initial access $\pi_i^0$ followed by $\pi_i'$. From $\pi_i'$ one can extract a simple path $\pi_i''$ — that is, $\pi_i''$ has no repeated accesses — of length at most $|Access| - 1$. The concatenated path $\pi_1^0\pi_1'' \cdots \pi_m^0\pi_m''$ has the same contents as $\pi$, starts and ends with the same accesses, and has at most $|Access|^2$ accesses.

Figure 3.5 gives an example of short path extraction. The first step is to find the first occurence of all accesses in the path (in red) to split $\pi$ accordingly. We thus obtain $\pi_1 = A_1$, $\pi_2 = A_2A_1$, $\pi_3 = A_3A_1A_2A_1A_1$ and $\pi_4 = A_4A_1A_3A_2A_1$. We then remove the cycles from the subpaths $\pi_i'$ obtained. This gives us $\pi_1'' = \varepsilon$, $\pi_2'' = \pi_3'' = \pi_4'' = A_1$. The final short path obtained is thus $A_1A_2A_1A_3A_1A_4A_1$.

□

**Theorem 20.** *The exist-miss problem is still in NP for cyclic CCGs.*

*Proof.* Follows from the preceding lemma: search for a witness path of length at most $|Access|^2$ in the CCG. □

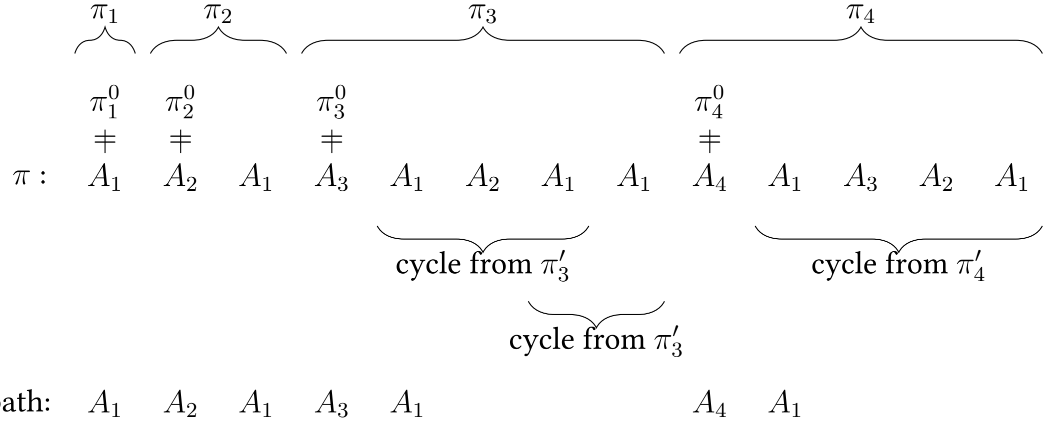



Figure 3.5 – Construction of a small path with identical content

#### Extensions

**Remark 21.** *The above theorems hold even if the starting cache state is unspecified, because what matters in their proofs is the contents of the path from the initial access to $x$ to the final location, and not the initial contents of the cache.*

**Remark 22.** *The proofs of NP-hardness for exist-hit and exist-miss on acyclic graphs for LRU carry over to FIFO.*

### 3.2.3 FIFO

FIFO (First-In, First-Out), also known as "round-robin", caches follow the same mechanism as LRU (a bounded queue ordered by age in the cache), except that a block is not rejuvenated on a hit.

They are used in Motorola PowerPC 56x, Intel XScale, ARM9, ARM11 [Rei09, p.21], among others.

#### Fundamental properties

The state of a FIFO cache with associativity $k$ is a word of length at most $k$ over the alphabet of cache blocks, composed of pairwise distinct letters; an empty cache is defined by the empty word. When an access is made to a block $a$, if it belongs to the word (*hit*) then the cache state does not change. If it does not belong to the word (*miss*), and the length of the word is less than $k$, then $a$ is prepended to the word; otherwise, the length of the word is exactly $k$ — the last letter of the word is discarded and $a$ is prepended.

**Lemma 23.** *The exist-hit and the exist-miss problems are in NP for acyclic control flow graphs.*

*Proof.* Guess a path nondeterministically and execute the policy along it. □

**Lemma 24.** *The exist-hit and the exist-miss problems are in PSPACE for general graphs.*

*Proof.* Simulate the execution of the policy using a polynomial-space nondeterministic Turing machine. Based on Savitch's theorem, both problems are in PSPACE. □

**Reduction to Exist-Hit**

This section describes how we reduce the reachability problem for the Boolean register machine to the exist-hit problem for the FIFO cache. The main idea of this reduction is to represent the boolean registers as a cache state, and the directed graph of the machine as a CCG. More precisely, we provide a systematic method to encode guards and assignment as sequences of accesses.

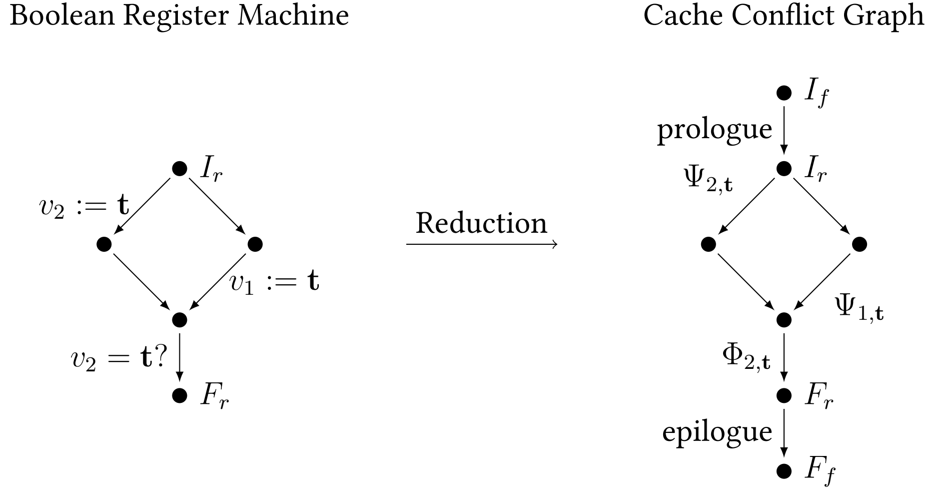



Figure 3.6 – Example of reduction

The main process is illustrated on Figure 3.6:

- A prologue initializes the cache to a state that reflects the initial boolean registers of the machine.
- Each assignment $v_i := b$ is turned into an sequence of accesses $\Psi_{i,b}$ (referred as assignment gadget in the following). This sequence modifies the cache state to mimic the assignment.
- Each guard $v_i = b$ is turned into a gadget $\Phi_{i,b}$ that keep the cache unchanged if and only if the guard is satisifed, and turn it into an "invalid" state otherwise.
- An epilogue is appended to filter out the "invalid" states. The aim of the epilogue is to turn the current cache state into a state that leads to a hit if and only if the current state is "valid".

The main difficulty in this reduction is that the Boolean register machines may terminate traces if a guard is not satisfied, whereas the cache problem has no guards and no way to terminate traces. Our workaround is that cache states that do not correspond to traces from the Boolean machine are irremediably marked as incorrect. Formally, we use an encoding into FIFO cache states that allows us to distinguish between *well-formed* and *not well-formed* cache states. Guards are then translated into access sequences that turn *well-formed* cache states into *not well-formed* ones when not satisfied, and that keep the cache state unmodified otherwise. The remaining of this section describes the encoding of registers, prologue, epilogue, guards and assignments.

**Encoding registers**

The associativity of the cache is chosen as $k = 2r - 1$, where $r$ is the size of the boolean register of the machine. The alphabet of cache blocks is $\{(a_{i,b})_{1 \le i \le r, b \in \{\mathbf{f},\mathbf{t}\}}\} \cup \{(e_i)_{1 \le i \le r}\} \cup \{(f_i)_{1 \le i \le r-1}\} \cup$

$\{(g_i)_{1\leq i\leq r-1}\}$. The internal state $v_1, \ldots, v_r$ of the register machine is to be encoded as the FIFO word

$$a_{r,v_r} e_r a_{r-1,v_{r-1}} e_{r-1} \ldots a_{1,v_1}. \tag{3.1}$$

Note that $e_1$ is not cached in this state.

Informally, a FIFO cache state containing the block $a_{i,\mathbf{f}}$ encodes a boolean register where $b_i = \mathbf{f}$, whereas a FIFO cache $a_{i,\mathbf{t}}$ encodes a register where $b_i = \mathbf{t}$. The idea is to simulate a delay-line memory that stores the state of all the booleans $b_i$s. When a register $b_i$ reaches the end of the delay line (the associated block $a_{i,v_i}$ is evicted), it is refreshed (i.e. the associated blocks is access again). The gadgets used to encode the guards and assignments are designed such that exactly one of the block $a_{i,\mathbf{t}}$ and $a_{i,\mathbf{f}}$ is stored in the cache at any time. In addition to the $a_{i,b}$, the cache state contains some $e_i$ blocks that are used to distinguish the *well-formed* cache states from the *not well-formed* ones. *Well-formed* states only contains ordered paired $(a_{i,b}, e_i)$ (except that one of the $e_i$ is not cached), whereas *not well-formed* cache states contain pairs in the reverse order $(e_i, a_{i,b})$. The $f_i$ and $g_i$ blocks will be used later.

**Encoding guards**

For simplicity reasons, the sequence of accesses associated to guards and assignment are decomposed into $r$ subsequences. Each of these subsequence updates or checks one of the boolean registers. To reason about these subsequenced, we introduce the notion of shift:

**Definition 25.** *We say that a FIFO word is* well-formed at shift $1$*, or* well-formed *for short if it is of the form*

$$a_{r,v_r} e_r a_{r-1,v_{r-1}} e_{r-1} \ldots a_{1,v_1} \tag{3.2}$$

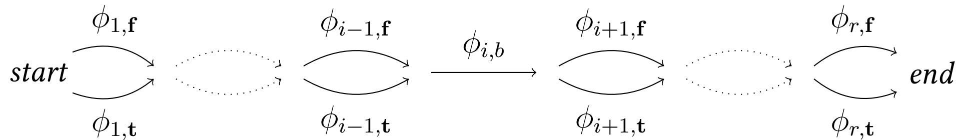
*We say that a FIFO word is well-formed at shift $i$ ($2 \leq i \leq r$) if it is of the form*

$$a_{i-1,v_{i-1}} \ldots e_2 a_{1,v_1} e_1 a_{r,v_r} \ldots a_{i+1,v_{i+1}} e_{i+1} a_{i,v_i} \tag{3.3}$$

*In both cases, we say that the FIFO word* corresponds *to the state $v_1, \ldots, v_r$.*

We now define the guard gadget, i.e. the automatic translation of a guard into sequence of accesses that checks whether the current cache state represents registers satisfying the guard.

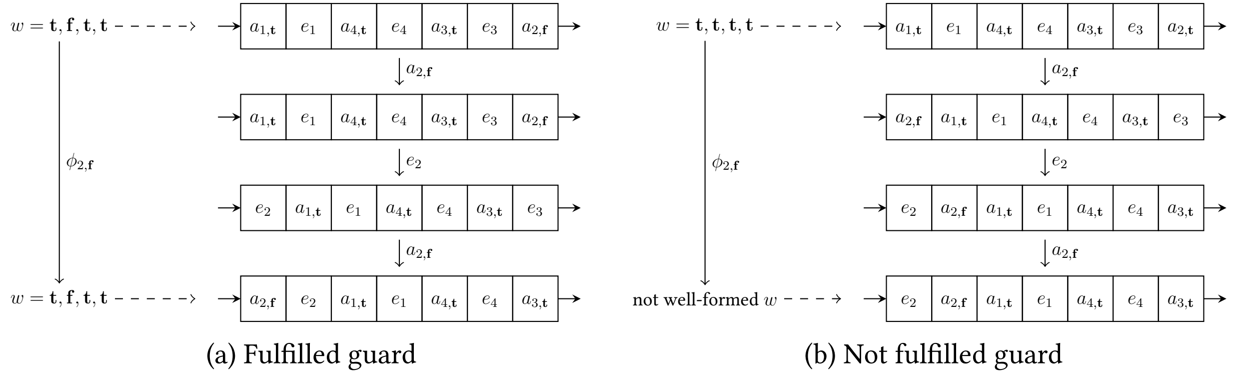
**Definition 26.** *The sequence $\Phi_{i,b}$ of accesses associated to a guard edge $v_i = b$ is:*




*where $\phi_{j,b}$ denotes the sequence of accesses $a_{j,b} e_j a_{j,b}$.*

Informally, a subsequence $\phi_{j,b}$ checks whether the register $v_j$ is equal to $b$. If so, the registers represented by the cache state is unchanged. Otherwise, the current cache is turned into a non well-formed cache state. This is illustrated on Figure 3.7. On Figure 3.7a, the entry cache state represents the registers $v_1 = \mathbf{t}, v_2 = \mathbf{f}, v_3 = \mathbf{t}, v_4 = \mathbf{t}$, well-formed at shift 2 so that $\phi_{2,\mathbf{f}}$ can be used to check that $v_2$ is false. We first check that $v_2$ is false by accessing the associated block $a_{2,\mathbf{f}}$. Because our cache state represents a set of registers where $v_2$ is false, this access results in a hit and the cache state is unchanged. Then, we shift the cache state by accessing $e_2$ (which evict $a_{2,\mathbf{f}}$), and $a_{2,\mathbf{f}}$ to place it at the beginning. On the other hand, when the guard is not fulfilled, we end up in a situation where the cache is not well-formed anymore. This situation is illustrated on

Figure 3.7b, where the entry cache state represents registers that are all true. Then the first access to $a_{2,\mathbf{f}}$ leads to a miss and the block is thus enqueued before $e_2$. $e_2$ is then accessed, placing it at the beginning of the cache state. The last access to $a_{2,\mathbf{f}}$ results in a hit and does not change the cache state. Finally, we end up in a situation where $e_2$ and $a_{2,\mathbf{f}}$ are swapped in comparison to the final state obtained when the guard is satisfied.



(a) Fulfilled guard (b) Not fulfilled guard

Figure 3.7 – Example of guard execution at shift 2

When both $\phi_{j,\mathbf{t}}$ and $\phi_{j,\mathbf{f}}$ are in parallel in the guard gadget, both values of $v_j$ are allowed. This construction is simply used to shift the cache state to the appropriate state. Then the subsequence $\phi_{i,b}$ is used to ensure $v_i$ has value $b$ as shown above, and the remaining of the gadget simply shifts the cache state to compensate the first sequence of shifts.

One can notice that when the guard is not fulfilled, the final cache states obtained is not arbitrary. Instead, the final cache states is similar to a *well-formed* state, where some contiguous blocks have been swapped. This is formalized by introducing *well-phased* states. In our proofs, *well-phased* states are used to guarantee one can not go back from a non *well-formed* state to a *well-formed* one.

**Definition 27.** *We say that a FIFO word is* well-phased at shift $1$*, or* well-phased *for short if it is of the form*

$$\beta_r \alpha_r \beta_{r-1} \dots \alpha_2 \beta_1 \tag{3.4}$$

*where, for each $i$:*

- *either $\alpha_i = e_i$ and $\beta_i = a_{i,b_i}$ for some $b_i$,*
- *or $\beta_i = e_i$ and $\alpha_i = a_{i,b_i}$ for some $b_i$.*

*We say that a FIFO word is well-phased at shift $i$ ($2 \le i \le r$) if it is of the form*

$$\beta_{i-1}\alpha_{i-1} \dots \beta_1 \alpha_1 \beta_r \alpha_r \dots \beta_{i+1} \alpha_{i+1} \beta_i \tag{3.5}$$

The following lemmas then expresses the correctness of the subsequence $\phi_{i,b}$.

**Lemma 28.** *Assume $w$ is well-formed at shift $i$, corresponding to state $\sigma = (\sigma_1, \dots, \sigma_r)$. If $\sigma_i = b$, then executing $\phi_{i,b}$ over FIFO state $w$ leads to a state well-formed at shift $i+1$ ($1$ if $i = r$), corresponding to $\sigma$ too. If $\sigma_i = \neg b$, then executing $\phi_{i,b}$ over FIFO state $w$ leads to a state well-phased, but not well-formed, at shift $i+1$ ($1$ if $i = r$).*

*Proof.* Without loss of generality we prove this for $i = 1$ and $b = \mathbf{f}$. Assume $w = a_{r,v_r} e_r \dots a_{2,v_2} e_2 a_{1,\mathbf{f}}$; then the sequence $\phi_{1,\mathbf{f}} = a_{1,\mathbf{f}} e_1 a_{1,\mathbf{f}}$ yields $a_{1,\mathbf{f}} e_1 a_{r,v_r} e_r \dots a_{2,v_2}$. Assume now $w = a_{r,v_r} e_r \dots a_{2,v_2} e_2 a_{1,\mathbf{t}}$; then $\phi_{1,\mathbf{f}}$ yields $e_1 a_{1,\mathbf{f}} a_{r,v_r} e_r \dots a_{2,v_2}$. $\square$

**Lemma 29.** *Assume $w$ is well-phased, but not well-formed, at shift $i$, then executing $\phi_{i,b}$ over FIFO state $w$ leads to a state well-phased, but not well-formed, at shift $i+1$.*

*Proof.* Without loss of generality, we shall prove this for $i = 1$. Let $w = \beta_r\alpha_r \dots \beta_2\alpha_2\beta_1$.

First case: $\beta_1 = e_1$. $\phi_{1,b} = a_{1,b}e_1a_{1,b}$ then leads to $e_1a_{1,b}\beta_r\alpha_r \dots \beta_2$, which is well-phased, but not well-formed due to the first two letters, at shift 2.

Second case: $\beta_1$ is either $a_{1,\mathbf{f}}$ or $a_{1,\mathbf{t}}$; assume the former without loss of generality. Then there exists $j > 1$ such that $\alpha_j = a_{j,v_j}$ and $\beta_j = e_j$ (because $w$ is not well-formed).

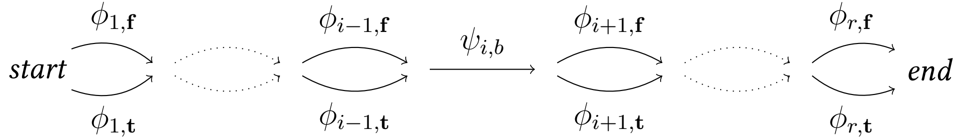
$\phi_{1,\mathbf{f}}$ then leads to $a_{1,\mathbf{f}}e_1\beta_r\alpha_r \dots \beta_2\alpha_2$, which is well-phased, but not well-formed due to the $\beta_j, \alpha_j$, at shift 2.

$\phi_{1,\mathbf{t}}$ leads to $e_1a_{1,\mathbf{t}}\beta_r\alpha_r \dots \beta_2\alpha_2$, which is well-phased, but not well-formed due to the first two letters, at shift 2. ☐

**Encoding assignments**

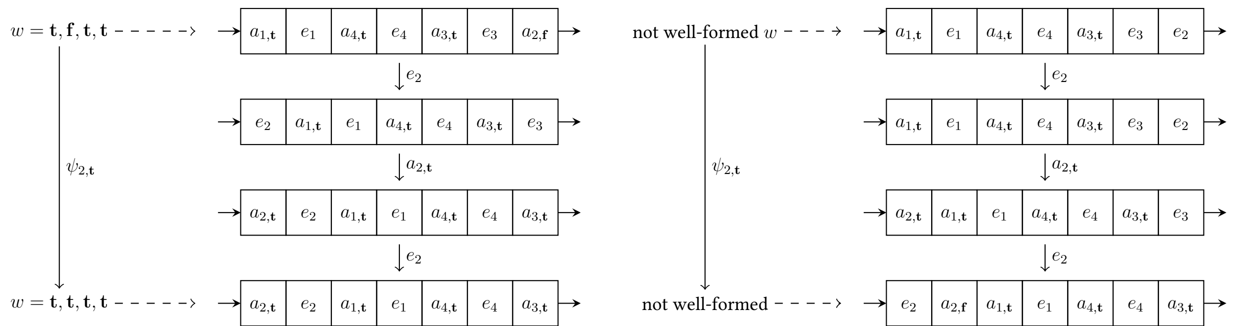
Now that we can handle guards, we propose the following gadget to handle assignments:

**Definition 30.** *Each assignment edge $v_i := b$ is replaced by the gadget $\Psi_{i,b}$:*




*where $\psi_{i,b}$ denotes the sequence of accesses $e_ia_{i,b}e_i$.*

The main idea of this gadget is to shift the cache state until the desired block $e_i$ is evicted. Then, the cache state is modified to reflect the assignment. This modification is illustrated on Figure 3.8. The case of a well-formed cache is shown on Figure 3.8a. In this example, the blocks $e_2$ and $a_{2,\mathbf{t}}$ are inserted at the beginning of the cache (both result in cache misses). The obtained cache state is well-formed, and left unmodified by the third access. Conversely, the case of a non well-formed but well-phased cache is illustrated on Figure 3.8b. More precisely, we show the case where the cache is ill-formed because of a guard implying the register $v_2$.




(a) Assignement when the input state is well-formed (b) Assignement when the input is not well-formed

Figure 3.8 – Example of assignment execution

We now prove the correctness of the assignment gadget.

**Lemma 31.** *Assume $w$ is well-formed at shift $i$, corresponding to state $\sigma = (\sigma_1, \dots, \sigma_r)$. Executing $\psi_{i,b}$ over FIFO state $w$ leads to a state well-formed at shift $i+1$ ($1$ if $i = r$), corresponding to $\sigma$ where $\sigma_i$ has been replaced by $b$.*

*Proof.* Without loss of generality we prove it for $i = 1$ and $b = \mathbf{f}$. Assume $w = a_{r,v_r}e_r \ldots a_{2,v_2}e_2a_{1,v_1}$; then the sequence $\psi_{1,\mathbf{f}} = e_1a_{1,\mathbf{f}}e_1$ yields $a_{1,\mathbf{f}}e_1a_{r,v_r}e_r \ldots a_{2,v_2}$. □

**Lemma 32.** *Assume $w$ is well-phased, but not well-formed, at shift $i$, then executing $\psi_{i,b}$ or over FIFO state $w$ leads to a state well-phased, but not well-formed, at shift $i + 1$.*

*Proof.* Without loss of generality, we shall prove this for $i = 1$. Let $w = \beta_r\alpha_r \ldots \beta_2\alpha_2\beta_1$.

First case: $\beta_1 = e_1$. $\psi_{1,b} = e_1a_{1,b}e_1$ then leads to $e_1a_{1,b}\beta_r\alpha_r \ldots \beta_2\alpha_2$, which is well-phased, but not well-formed due to the first two letters, at shift 2.

Second case: $\beta_1$ is either $a_{1,\mathbf{f}}$ or $a_{1,\mathbf{t}}$; assume the former without loss of generality. Then there exists $j > 1$ such that $\alpha_j = a_{j,v_j}$ and $\beta_j = e_j$. $\psi_{1,b}$ then leads to $a_{1,b}e_1\beta_r\alpha_r \ldots \beta_2\alpha_2$, which is well-phased, but not well-formed due to the $\beta_j, \alpha_j$, at shift 2. □

The two following corollaries then express the overall correctness of the reduction over a all path in the CCG.

**Corollary 33.** *Assume starting in a well-formed FIFO state, corresponding to state $\sigma$, then any path through the gadget encoding an assignment or a guard*

- *either leads to a well-formed FIFO state, corresponding to the state $\sigma'$ obtained by executing the assignment, or $\sigma' = \sigma$ for a valid guard;*
- *or leads to a well-phased but not well-formed state.*

**Corollary 34.** *Assume starting in a well-phased but not well-formed FIFO state, then any path through the gadget encoding an assignment or a guard leads to a well-phased but not well-formed FIFO state.*

### Prologue

Using these assignment gadget, we can define a starting sequence in the CCG that initializes the cache to a well-formed cache state representing the registers $v_1 = \mathbf{f}, v_2 = \mathbf{f}, \ldots, v_r = \mathbf{f}$.

**Definition 35.** *From the cache analysis initial vertex $I_f$ to the register machine former initial vertex $I_r$ there is a prologue, a sequence of accesses $a_{1,\mathbf{f}}e_1 \ldots a_{r-1,\mathbf{f}}e_{r-1}a_{r,\mathbf{f}}$. This first sequence of accesses aims at initializing the FIFO cache to a state representing a register machine having all registers set to false.*

The correctness of this initial sequence is a direct consequence of the freshness of blocks $a_{i,v_i}$ and $e_i$.

So far, our reduction transforms a boolean register machine into a CCG such that any path in the machine is associated to a path in the CCG. This path either leads to valid cache state if all guard are fulfilled along the path in the machine, or a well-phased but not well-formed state otherwise.

### Epilogue

Finally, to complete the reduction, we propose a final sequence that turns well-formed cache states into a state where $a_{r,\mathbf{f}}$ is cached and other well-phased caches into a cache not containing $a_{r,\mathbf{f}}$. By doing so, there is a path leading to cache containing $a_{r,\mathbf{f}}$ if and only if the final path of the boolean register machine is reachable.

**Definition 36.** *From the register machine former final vertex $F_r$, we append a sequence of accesses $\psi_{1,\mathbf{f}} \ldots \psi_{r,\mathbf{f}}$ constituting the first part of the epilogue. We note $F_a$ the final vertex of this sequence. From $F_a$, we add another sequence of accesses $a_{1,\mathbf{f}} g_1 e_2 f_2 a_{2,\mathbf{f}} g_2 \ldots e_{r-1} f_{r-1} a_{r-1,\mathbf{f}} g_{r-1} e_r f_r$, constituting the second part of the epilogue. We note $F_h$ the final vertex of this second sequence.*

**Lemma 37.** *The path from $F_r$ to $F_a$:*

- *transforms a well-phased but not well-formed FIFO state into a well-phased but not well-formed FIFO state*
- *transforms any well-formed FIFO state into a well-formed FIFO state $w_0$ corresponding to the initial register state (all registers zero).*

*Proof.* This simply results from the correctness of the $\psi_{i,b}$ sequences. □

We now finish the reduction by showing that the second part of the epilogue leads a cache containing $a_{r,\mathbf{f}}$ if and only if the cache state reaching this sequence is well-formed.

**Lemma 38.** *The path $a_{1,\mathbf{f}} g_1 e_2 f_2 a_{2,\mathbf{f}} g_2 \ldots e_{r-1} f_{r-1} a_{r-1,\mathbf{f}} g_{r-1} e_r f_r$ (from $F_a$ to $F_h$):*

- *transforms $w_0$ into $f_r g_{r-1} f_{r-1} \ldots g_2 f_2 g_1 a_{r,\mathbf{f}}$*
- *transforms any other word $w$ consisting of $a$'s and $e$'s into a word not containing $a_{r,\mathbf{f}}$.*

*Proof.* The first point directly results from the update of $w_0$ according to the given sequence. $a_{1,\mathbf{f}}$ results in a hit and is evicted right after when accessing $g_1$. Then $e_2$ is accessed, leading again to a hit, and is evicted when accessing $f_2$. This process continues until $e_r$ is evicted, leaving $a_{r,\mathbf{f}}$ in the cache.

We shall now prove that it is necessary for the input word to be exactly $w_0$ in order for the final word to contain $a_{r,\mathbf{f}}$. In order for that, there must have been at most $2r-2$ misses along the path. The accesses to $g_1, f_2, g_2, \ldots, f_{r-1}, g_{r-1}, f_r$ are always misses because they are fresh blocks. As there are $2r-2$ of them, there must have been exactly those misses and no others. This implies that $a_{r,\mathbf{f}}$ was in the first position in $w$.

When $e_r$ is processed, similarly there were exactly $2r-3$ misses, and $e_r$ must be a hit. This implies that $e_r$ was in the first or the second position in $w$, but since the first position was occupied by $a_{r,\mathbf{f}}$, $e_r$ must have been in the second position.

The same reasoning holds for all preceding locations, down to the first one, and thus the lemma holds. □

From all these lemmas, the main result follows:

**Corollary 39.** *There is an execution of the FIFO cache from $I_f$ to $F_f$ such that $a_{r,\mathbf{f}}$ is in the final cache state if and only if there is an execution of the Boolean register machine from $I_r$ to $F_r$.*

**Theorem 40.** *The exist-hit problem for FIFO caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

*Proof.* As seen above, a register machine reachability problem can be reduced in polynomial time to a exist-hit FIFO problem, preserving acyclicity if applicable. □

**Remark 41.** *We have described a reduction from a $r$-register machine to a FIFO cache problem with an odd $2r-1$ number of ways. This reduction may be altered to yield an even number of ways as follows. Two special padding letters $p$ and $p'$ are added. A well-formed state is now $a_{r,v_r} e_r \ldots a_{2,v_2} e_2 a_{1,v_1} p$; The definition of well-phased states is similarly modified. Each gadget $G$ for assignment or guard is replaced by $p'GpG$. The first $p'$ turns padding $p$ into $p'$, $G$ is applied. The second $p'$ turns $p'$ into $p$ and $G$ is applied again.*

*This remark also applies to the exist-miss problem studied below.*

#### Reduction to Exist-Miss

We modify the reduction for exist-hit in order to exhibit a miss on $a_{r,\mathbf{f}}$ later on if and only if it is in the cache at the end of the graph defined above.

**Definition 42.** *We transform the register machine graph into a cache analysis graph as in exist-hit case, with the following modification: in between $F_h$ and $F_f$ we insert a sequence $e_r a_{r-1,\mathbf{f}} e_r \dots a_{1,\mathbf{f}} e_1 a_{r,\mathbf{f}}$, constituting the third part of the epilogue.*

**Lemma 43.** *The path from $F_h$ to $F_f$ transforms $f_r g_{r-1} f_{r-1} \dots g_2 f_2 g_1 a_{r,\mathbf{f}}$ into a word not containing $a_{r,\mathbf{f}}$. It transforms any word composed of $f$'s and $g$'s only into a word containing $a_{r,\mathbf{f}}$.*

**Theorem 44.** *The exist-miss problem for FIFO caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

#### Extension to arbitrary starting cache

**Lemma 45.** *The exist-hit and exist-miss problems for an empty starting FIFO cache state are reduced, in linear time, to the same kind of problem for an arbitrary starting cache state, with the same associativity.*

*Proof.* Noting $Blocks$ be the alphabet of blocks in the problem and $k$ its associativity, Let $e_1, \dots, e_{2k-1}$ be new blocks not in $Blocks$; after accessing them in sequence, the cache contains only elements from these accesses [RGBW07a, Th. 1]. Prepend this sequence as a prologue to the cache problem; then the rest of the execution of the cache problem will behave as though it started from an empty cache. □

**Corollary 46.** *The exist-hit and exist-miss problems for FIFO caches with arbitrary starting state is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

#### Extension to Pseudo-RR caches

Recall how a FIFO cache with multiple cache sets — the usual approach in hardware caches — operates. A memory block of address $x$ is stored in the cache set number $H(x)$ where $H$ is a suitable function, normally a simple combination of the bits of $x$. In typical situations, this is as though the address $x$ were specified as a pair $(s, a)$ where $s$ is the number of the cache set and $a$ is the block name to be used by the FIFO in cache set number $s$.

In a FIFO cache, each cache set, being a FIFO, can be implemented as a circular buffer: an array of cache blocks and a "next to be evicted" index. In contrast, in a pseudo-RR cache, the "next to be evicted" index is global to all cache sets.

A FIFO cache exist-hit or exist-miss problem with cache block labels $a_1, \dots, a_n$ can be turned into an equivalent pseudo-RR problem simply by using $(s, a_1), \dots, (s, a_n)$ as addresses for a constant distinguished cache set $s$. Thus, both exist-hit and exist-miss are NP-hard for acyclic control-flow graphs on pseudo-RR caches, and PSPACE-hard for general control-flow graphs.

The same simulation arguments used for FIFO (see Section 3.2.3) hold for establishing membership in NP and PSPACE respectively.

### 3.2.4 PLRU

Because LRU caches were considered too difficult to implement efficiently in hardware, various schemes for heuristically approximating the behavior of a LRU cache (keeping the most recently

used data) have been proposed. By "heuristically approximating" we mean that these schemes are assumed, on "typical" workloads, to perform close to LRU, even though worst-case performance may be different.[8] Some authors lump all such schemes as "pseudo-LRU" or "PLRU", and call the scheme in the present section "tree-based PLRU" or "PLRU-t" [AMM04], while some others [Rei09, p. 26] call "PLRU" only the scheme discussed here.

### PLRU caches

Here is a reminder about the behavior of PLRU caches, see Section 2.1 for more details. The cache lines of a PLRU cache, which may contain cached blocks, are arranged as the leaves of a full binary tree — thus the number of ways $k$ is a power of 2, often $4$ or $8$. Two lines may not contain the same block. Each internal node of the tree has a tag bit, which is represented as an arrow pointing to the left or right branch. The internal state of the cache is thus the content of the lines and the $k-1$ tag bits.

There is always a unique line such that there is a sequence of arrows from the root of the tree to the line; this is the line *pointed at by the tags.* Tags are said to be *adjusted away* from a line as follows: on the path from the root of the tree to the line, tag bits are adjusted so that the arrows all point away from that path.

When a block $a$ is accessed:

- If the block is already in the cache, tags are adjusted away from this line.
- If the block is not already in the cache and one or more cache lines are empty, the leftmost empty line is filled with $a$, and tags are adjusted away from this block.
- If the block is not already in the cache and no cache line is empty, the block pointed at by the tags is evicted and replaced with $a$, and tags are adjusted away from this block.

In this section, we prove that the *exist-hit* and *exist-miss* problems are both *PSPACE-complete* for the PLRU replacement policy.

### Exist-Hit Problem

As for the FIFO policy, we prove the complexity of *exist-hit* by reducing the reachability problem of a boolean register machine to this problem. More precisely, we reduce the reachability problem of a Boolean $r$-register machine to the PLRU exist-hit problem for a $(2r+2)$-way cache — without loss of generality, we can always add useless registers so that $2r+2$ is a power of two. The alphabet of cache blocks we use in this reduction is $\{(a_{i,b})_{1\leq i\leq r, b\in\{\mathbf{f},\mathbf{t}\}}\} \cup \{(e_i)_{0\leq i\leq r}\} \cup \{c\}$.

### Encoding registers

Similarly to the FIFO case, we store blocks $a_{i,b}$ in the cache to represent a state of the boolean register machine where $v_i = b$. The blocks $e_i$ are always in the cache and are used to adjust the

[8] Experimentally, on typical workloads, the tree-based PLRU scheme described in this section is said to produce 5% more misses on a level-1 data cache compared to LRU [AMM04]. However, that scheme may, under specific concocted workloads, indefinitely keep data that are actually never used except once— a misperformance that cannot occur with LRU [HLTW03a]. This can produce *domino effects*: the cache behavior of a loop body may be indefinitely affected by the cache contents before the loop [Ber06].

Because of the difficulties in obtaining justifiable bounds on the worst-case execution times of programs running on a PLRU cache, some designers of safety-critical real-time systems lock all cache ways except for two, exploiting the fact that a 2-way PLRU cache is the same as a 2-way LRU cache and thus recovering predictability [Ber06, §3].

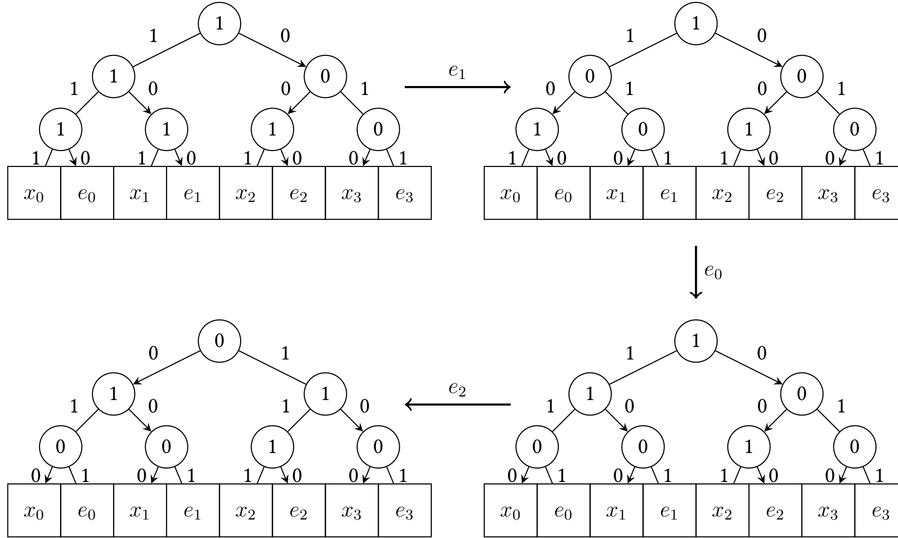



Figure 3.9 – Sequence $\pi_1 = e_1e_0e_2$ makes tags point at $x_1$ without changing cache content

block pointed at by the tags. Finally, the block $c$ is used as a witness of valid execution, i.e. a path in the CCG maintains $c$ in the cache as long as the guards are satisifed. By doing so, the final cache state obtained contains $c$ if and only if the final vertex of the boolean register machine is reachable.

**Definition 47.** *We say that a PLRU cache state is* well-formed *and* corresponds *to a Boolean state* $(v_i)_{1\leq i\leq r}$ *if its leaves are, from left to right:* $c, e_0, a_{1,v_1}, e_1, \ldots, a_{r,v_r}, e_r$.

**Definition 48.** *We say that a PLRU cache state is* well-phased *if its leaves are, from left to right:* $x_0, e_0, a_{1,v_1}, e_1, \ldots, a_{r,v_r}, e_r$ *where* $x_0$ *can be* $c$ *or any* $a_{i,b}$*. The former case corresponds to a* well-formed *cache state, whereas the latter correspond to an invalid execution. In this case, there exists* $i$ *such that* $1 \leq i \leq r$ *and that both* $a_{i,\mathbf{t}}$ *and* $a_{i,\mathbf{f}}$ *are in the cache.*

**Encoding guards and assignement**

To manipulate the blocks representing the value of registers, we need to manipulate the tags to make the arrows points to the different $a_{i,v_i}$ stored. Our reduction is based on the following lemma, which states we can make the tags point at any cache line not containing an $e_i$ block.

**Lemma 49.** *Let* $0 \leq i \leq r$*, there exists a sequence* $\pi_i$ *of accesses, of length logarithmic in* $r$*, such that, when run on a well-phased cache state* $x_0, e_0, x_1 \ldots, x_r, e_r$*, that sequence makes tags point at* $x_i$ *without changing the contents of the cache lines.*

*Proof.* Consider the “tree path” from the cache line $x_i$ to the root tree. We want to set all tags on this path such that the arrows point at $x_i$. We do so level by level, starting from the cache line, by accessing a block $e_i$ that is not in the same subtree than $x_i$. Figure 3.9 shows an example of sequence $\pi_1$ that can be used to make the cache state point at $x_1$. The first step is to make the last segment of the path from the tree root to $x_1$ point to $x_i$ instead of $e_1$. This is done by accessing $e_1$. Then, we want the arrows to point toward the $(x_1, e_1)$ subtree instead of the $(x_0, e_0)$ subtree. We thus access $e_0$ to do so. Finally, we want the root tag to point to the left, and we thus access one of the $e_i$ blocks of the right subtree: $e_2$ is a possible choice. □

Let $1 \leq i \leq r$, $b \in \{\mathbf{t}, \mathbf{f}\}$. The guard gadget we use is $\Phi_{i,b} = \pi_0 a_{i,b}$, where $\pi_0$ is any sequence that makes the tags point at the cache line 0. The intuition is that we make the tags points at

the cache line 0, which contains the block $c$, such that $c$ is evicted if the access to $a_{i,b}$ results in a miss. The idea is that any missed guard irremediably removes $c$ from the cache. In addition, we use the assignment gadget $\Psi_{i,b} = \pi_i a_{i,b}$. This second gadget works by making the tags point at the register value to modifiy, and accessing the block $a_{i,b}$ associated to the register new value.

The following lemmas, stating the correctness of the gadgets, are easily proved by symbolically simulating the execution of the gadgets over the cache states:

**Lemma 50.** *$\Phi_{i,b}$ and $\Psi_{i,b}$ map any well-phased but not well-formed state to a well-phased but not well-formed state.*

**Lemma 51.** *$\Psi_{i,b}$ maps a well-formed state to a well-formed state corresponding to the same Boolean state where register $i$ has been replaced by $b$.*

**Lemma 52.** *$\Phi_{i,b}$ maps a well-formed state corresponding to a Boolean state $(v_i)_{1\leq i\leq r}$ to*

- *if $v_i = b$, a well-formed state corresponding to the same Boolean state;*
- *otherwise, a well-phased but not well-formed state.*

We can then build the complete CCG associated to the given boolean register machine as follows:

**Definition 53.**
- *From the CCG initial vertex $I_p$ to the register machine former initial vertex $I_r$ there is a sequence of accesses $c, e_0, a_{1,\mathbf{f}}, e_1, \ldots, a_{r,\mathbf{f}}, e_r$.*
- *Each guard edge $v_i = b$ is replaced by the sequence $\Phi_{i,b}$, and each assignment edge $v_i := b$ by the sequence $\Psi_{i,b}$.*
- *The cache final vertex $F_p$ is the same as the register machine final vertex $F_r$.*

Finally, the correctness of the whole reduction is trivially deduced from the preceding lemmas.

**Corollary 54.** *There is an execution of the PLRU cache from $I_p$ to $F_p$ such that $c$ is in the final cache state if and only if there is an execution of the Boolean register machine from $I_r$ to $F_r$.*

*Proof.* One simply applies the correctness lemmas associated to the assignment and guard gadgets. The final cache state is well-formed if and only if the execution path in the CFG corresponds to a valid execution of the Boolean register machine. A well-phased state is well-formed if and only if it contains $c$. ☐

**Theorem 55.** *The exist-hit problem for PLRU caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

### Exist-Miss Problem

Similarly to the FIFO replacement policy, we build a reduction for the *exist-miss* problem from the reduction of the *exist-hit* problem. We use an extra blocks $d$ to turn well-formed states into well-phased but not well-formed ones, and conversely.

**Definition 56.** *Let $Z$ be the sequence $\pi_r c \pi_0 d$.*

**Lemma 57.** *$Z$ turns any well-formed state into a state not containing $c$. $Z$ turns any well-phased but not well-formed state into a state containing $c$.*

*Proof.* Consider a well-formed state $ce_0a_{1,v_1}e_1 \dots a_{r,v_r}e_r$. $\pi_r$ only changes the tags value; then the access to $c$ does not change the line contents since $c$ is in the cache, and $c$ is replaced by $d$.

Consider a well-phased but not well-formed state $x_0e_0a_{1,v_1}e_1 \dots a_{r,v_r}e_r$ where $x_0 \neq c$. $Z$ replaces $a_{r,v_r}$ by $c$; then $x_0$ is replaced by $d$. □

The CCG used in the reduction is then identical to the *exist-hit* case, except that $Z$ is appended as epilogue to turn hits to $c$ into misses and conversely.

**Lemma 58.** *There is an execution of the PLRU cache from $I_p$ to $F_p$ such that $c$ is not in the final cache state if and only if there is an execution of the Boolean register machine from $I_r$ to $F_r$.*

**Theorem 59.** *The exist-miss problem for PLRU caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

#### Extension to an arbitrary starting cache

**Lemma 60.** *The exist-hit and exist-miss problems for an empty starting PLRU cache state are reduced, in linear time, to the same kind of problem for an arbitrary starting cache state, with the same associativity.*

*Proof.* Same proof as 45, except we need a sequence of $\frac{k}{2}\log_2 k + 1$ new blocks [RGBW07a, Th. 12]. □

**Corollary 61.** *The exist-hit and exist-miss problems for FIFO caches with arbitrary starting state is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

### 3.2.5 NMRU

Other forms of "pseudo-LRU" schemes have been proposed than the one discussed in Section 3.2.4. One of them, due to [MPH94] is based on the use of "most recently used" bits. It is thus sometimes referred to as the "not most recently used" (NMRU) policy, or "PLRU-m" [AMM04]. Confusingly, some literature [Rei09] also refers to this policy as "MRU" despite the fact that in this policy, it is *not* the most recently used data block that is evicted first. In this thesis we refer to this policy based on "most recently used" bits as NMRU.

#### NMRU caches

This section is a reminder about the NMRU replacement policy (see Section 2.1 for details).

**Definition 62.** *The internal state of an $k$-way NMRU cache is a sequence of at most $k$ memory blocks $\alpha_i$, each tagged by a $0/1$ "MRU-bit" $r_i$ saying whether the associated block is to be considered not recently used (0) or recently used (1), denoted by $\alpha_1^{r_1} \dots \alpha_k^{r_k}$.*

*An access to a block in the cache, a* hit*, results in the associated MRU-bit being set to $1$. If there were already $k-1$ MRU-bits equal to $1$, then all the other MRU-bits are set to $0$.*

*An access to a block $b$ not in the cache, a* miss*, results in:*

- *if the cache is not full (number of blocks less than $k$), then $b^1$ is appended to the sequence*
- *if the cache is full (number of blocks equal to $k$), then the leftmost (least index $i$) block with associated MRU-bit $0$ is replaced by $b^1$. If there were already $k-1$ MRU-bits equal to $1$, then all the other MRU-bits are set to $0$.*

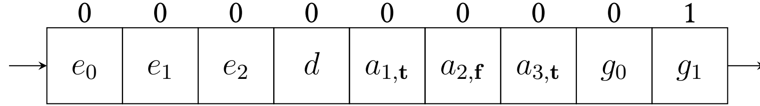



Figure 3.10 – Encoding the word $w = (\mathbf{t}, \mathbf{f}, \mathbf{t})$ as NMRU cache state

**Remark 63.** *This definition is correct because the following invariant is maintained: either the cache is not full, or it is full but at least one MRU-bit is zero.*

**Example 64.** *Assume $k = 4$. If the cache contains $a^0b^0c^0$, then an access to $d$ yields $a^0b^0c^0d^1$ since the cache was not full. If $a$ is then accessed, the state becomes $a^1b^0c^0d^1$. If $e$ is then accessed, the state becomes $a^1e^1c^0d^1$ since $b$ was the leftmost block with a zero MRU-bit. If $f$ is then accessed, then the state becomes $a^0e^0f^1d^0$.*

### Reduction to Exist-Hit

As for the FIFO and PLRU replacement policies, we reduce the reachability problem for boolean-register machines to the exist-hit problem for the NMRU cache. Similarly, we do so by providing an encoding of boolean registers as cache states, and gadgets for guards and assignments.

### Encoding boolean registers

The associativity of the cache is chosen as $k = 2r + 3$. The alphabet of the cache blocks is $\{(a_{i,b})_{1\leq i\leq r, b\in\{\mathbf{f},\mathbf{t}\}}\} \cup \{(e_i)_{1\leq i\leq r}\} \cup \{(c_i)_{1\leq i\leq r}\} \cup \{d\} \cup \{g_0, g_1\}$.

As for the previous replacement policies, the $a_{i,b}$ blocks are used to encode the value of registers, and the $e_i$ blocks are used (together with $d$) to distinguish valid states from invalid ones. The $c_i$ blocks will be used in the epilogue, whereas $g_0$ and $g_1$ are used to keep control over the global-flips.

The internal state $w = v_1, \ldots, v_r$ of the register machine is to be encoded as the NMRU state

$$e_1^0 \ldots e_r^0 d^0 a_{1,v_1}^0 \ldots a_{r,v_r}^0 g_0^0 g_1^1 \tag{3.6}$$

where the exponent (0 or 1) is the MRU-bit associated with the block.

For instance, a boolean register machine with 3 registers containing the values $\mathbf{t}, \mathbf{f}, \mathbf{t}$ is encoded by the NMRU cache state represented on Figure 3.10.

All valid states have the same form, with $e_i$ blocks on the left of $d$ and $a_{i,b}$ blocks on the right. Conversely, if one of the $a_{i,b}$ is on the left of $d$, the cache state is invalid. More formally, we define *well-formed* and *well-phased* cache states as follows.

**Definition 65.** *We say that an NMRU state is well-formed at step $s \in \{0, 1\}$ if it is of the form*

$$\beta_1^0 \ldots \beta_r^0 d^0 \alpha_1^0 \ldots \alpha_r^0 g_0^s g_1^{1-s} \tag{3.7}$$

*where $\forall i, 1 \leq i \leq r, \alpha_i \in \{a_{\sigma(i),\mathbf{f}}, a_{\sigma(i),\mathbf{t}}\}$, $\beta_i = e_{\sigma'(i)}$ and $\sigma$ and $\sigma'$ are two permutations of $[1, r]$. In other words, a well-formed state contains $r$ distinct blocks $e_i$ placed before $d$, and $r$ blocks $a_{i,b}$, with distinct $i$'s, placed between $d$ and $g_0$. We say "well-formed" for short if $s = 0$.*

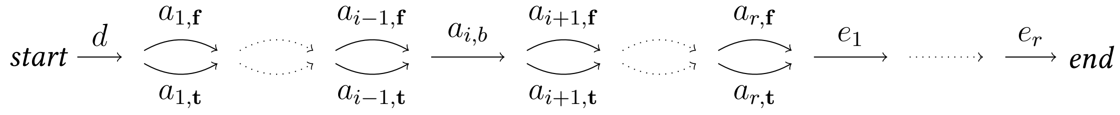
**Definition 66.** *We say that an NMRU state is well-phased at step $s \in \{0, 1\}$ if it is of the form*

$$\gamma_{\sigma(1)}^0 \ldots \gamma_{\sigma(r)}^0 d^0 \gamma_{\sigma(r+1)}^0 \ldots \gamma_{\sigma(2r)}^0 g_0^s g_1^{1-s} \tag{3.8}$$

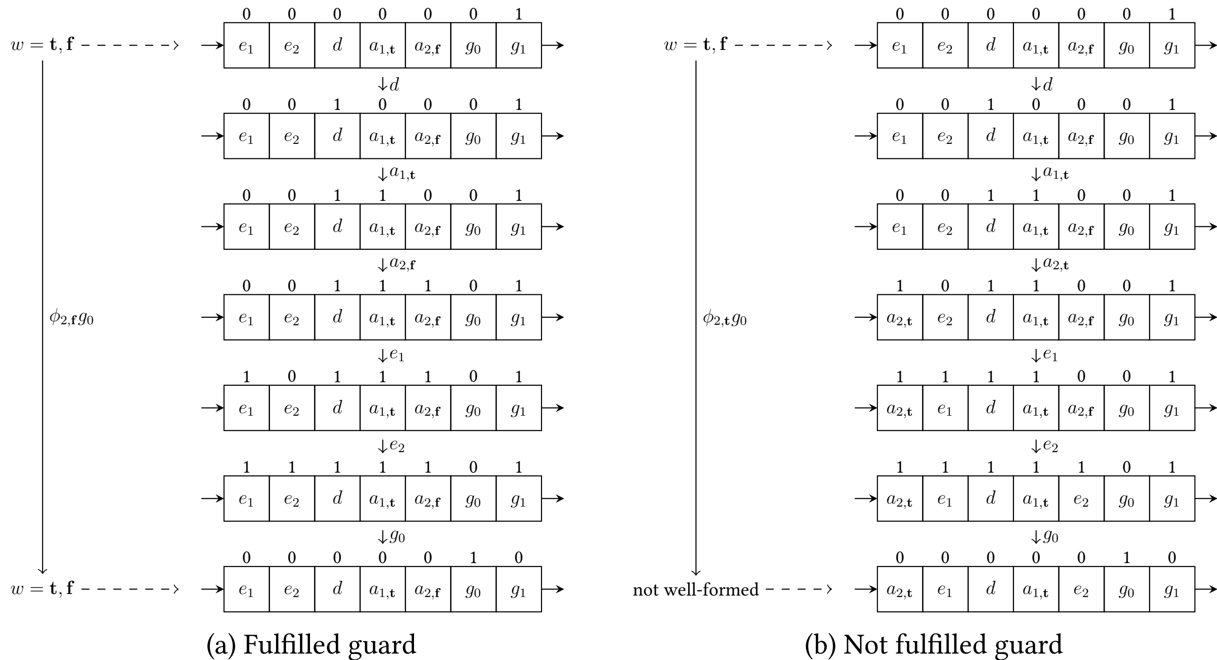
*where $\gamma_1 = e_1, \ldots, \gamma_r = e_r, \gamma_{r+1} \in \{a_{1,\mathbf{f}}, a_{1,\mathbf{t}}\}, \ldots, \gamma_{2r} \in \{a_{r,\mathbf{f}}, a_{r,\mathbf{t}}\}$ and $\sigma$ is a permutation of $[1, 2r]$. In other words, a "well-phrased" state is "well-formed" state without the constraint that blocks $e_i$ and $a_{i,v_i}$ must be respectively on the left and right of $d$. We say "well-phased" for short if $s = 0$.*

**Encoding guards** To verify that a cache state represents a boolean-register machine that satisfies a given guard, we propose the following encoding of guards.

**Definition 67.** *Each guard edge $v_i = b$ is replaced by the gadget $\Phi_{i,b} = \phi_{i,b} g_0 \phi_{i,b} g_1$, where $\phi_{i,b}$ is*




Consider the sequence $\phi_{i,b} g_0$ and a well-formed state. After the first access to $d$, a sequence of blocks $a_{j,b}$ are accessed. If one of them results in a miss, one of the $e_j$ on the left of $d$ is evicted, leading to a state which is not well-formed. The sequence of $e_j$ then replaces the $a_{j,v_j}$ that were not accessed. Conversely, if all the $a_{j,b}$ result in hits, the sequence of $e_j$ accesses also lead to hits and the order of blocks in the cache state is left unchanged. Figure 3.11 shows an example of (half-)guard execution on a cache state representing the registers $v_1 = \mathbf{t}, v_2 = \mathbf{f}$. On the left example (Figure 3.11a), the guard ($v_2 = \mathbf{f}$) is satisfied. As a result, the final cache state is well-formed at step 1 and represent the same cache state. To obtain the same state well-formed at shift 0, one simply execute again the same sequence by replaceing $g_0$ by $g_1$. Hence the definition $\Phi_{2,\mathbf{f}} = \phi_{2,\mathbf{f}} g_0 \phi_{2,\mathbf{t}} g_1$. On the right (Figure 3.11a), the guard executed is $v_2 = \mathbf{t}$ and is thus not satisfied. As a result, the final cache state is well-phased, but not well-formed. Repeating the same sequence with $g_1$ instead of $g_0$ does not change the form of the cache state.




(a) Fulfilled guard (b) Not fulfilled guard

Figure 3.11 – Example of guard execution at step 0

The following lemma expresses the correctness of the guard gadget more formally.

**Lemma 68.** *Executing a path through $\phi_{i,b} g_s$ over an NMRU state $w$ well-phased at step $s$ always leads to a state well-phased at step $1 - s$. Furthermore that state*

- *either is not well-formed at step $1 - s$*
- *or is identical to $w$ except for the $g_0$ and $g_1$ blocks, and this may occur only if $a_{i,b}$ belongs to $w$.*

*Proof.* The input state $w$ is $x_1^0, \ldots, x_r^0, d^0, x_{r+1}^0, \ldots, x_{2r}^0, g_0^0, g_1^1$ where the $(x_i)_{1 \le i \le 2r}$ are a permutation of $\{e_i \mid 1 \le i \le r\} \cup \{a_{i,\beta_i} \mid 1 \le i \le r\}$ for some sequence of Booleans $(\beta_i)_{1 \le i \le r}$.

Consider a path through $\phi_{i,b}$: it consists of $d$, followed by a sequence of $r$ $a$'s, then $r$ $e$'s. Each of these accesses either freshens, or overwrites, one of the $x$ positions. After the sequence of $a$'s, there are either no $a$'s to the left of $d$, or at least one. The former case is possible only if all $a$'s are hits, freshening positions to the right of $d$ — this means all these positions are left untouched except that their MRU bits are flipped to 1. Then the sequence of $e$'s just flips to 1 the MRU-bits of the $e$'s, all located to the left of $d$. The resulting state is thus identical to $w$ except that all MRU-bits to the left of $g_0$ have been flipped to 1; thus after accessing $g_0$, the state is identical to the initial state except that it ends with $g_0^{1-s} g_1^s$ instead of $g_0^s g_1^{1-s}$.

Now consider the latter case: after the sequence of $a$'s there is at least one position of the form $a_{j,\beta}^1$ to the left of $d$. This position cannot be overwritten by the $e$'s. After the path through $\phi_{i,b}$, the state is thus of the form $x_1^1, \ldots, x_r^1, d^1, x_{r+1}^1, \ldots, x_{2r}^1, g_0^0, g_1^1$, and one of the $x_j$ for $1 \le j \le r$ is an $a$. The access to $g_0$ yields $x_1^0, \ldots, x_r^0, d^0, x_{r+1}^0, \ldots, x_{2r}^0, g_0^{1-s}, g_1^s$. This state is well-phased but not well-formed. □

**Encoding assignments** The gadget used to encode assignments is similar to the guard gadget. The idea is to access all the $e_i$ blocks and $a_{i,v_i}$ blocks except the one we want to force the value to $b$. By doing so, all the blocks on the left of $g_0$ except $a_{i,v_i}$ have a MRU-bit set to 1. By accessing $a_{i,b}$, we thus modify the value of $v_i$.

**Definition 69.** *More precisely, the assignment gadget associated to $v_i := b$ is $\Psi_{i,b} = \psi_{i,b} g_0 \psi_{i,b} g_1$, where $\psi_{i,b}$ is*

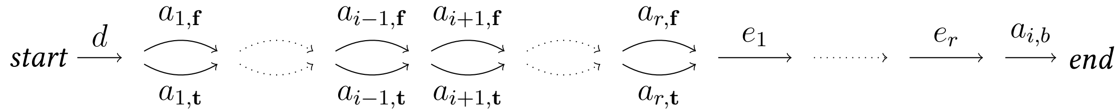


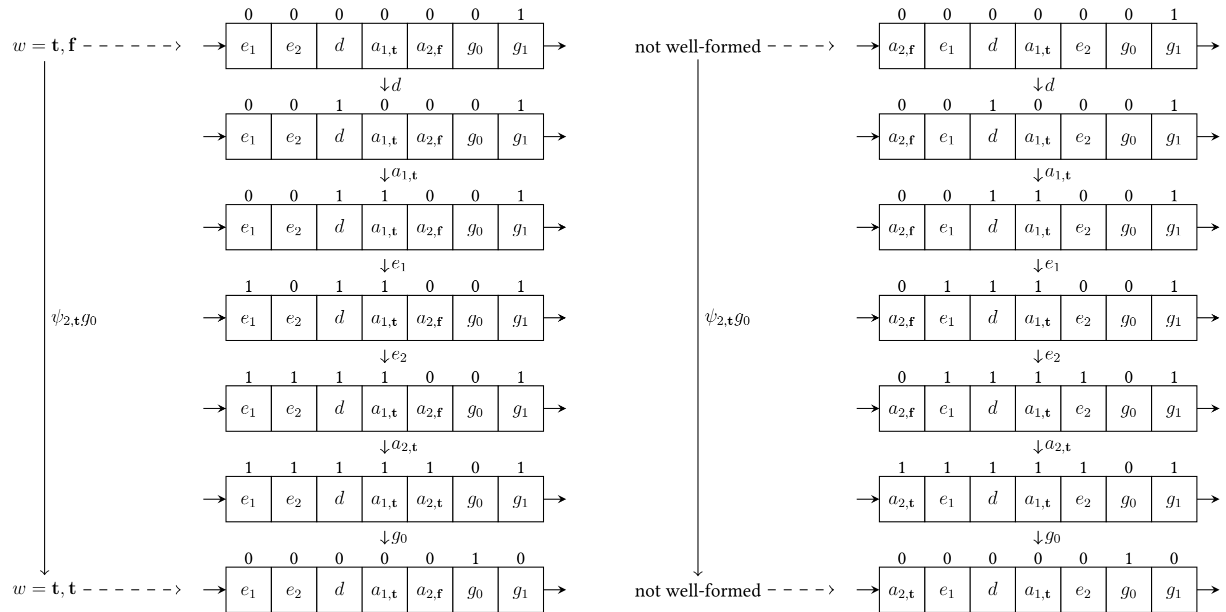

Figure 3.12 shows an example of such assignment gadget. On a well-formed input cache state (see Figure 3.12a), the sequence of $a_{j,v_j}$ and $e_j$ blocks only result in hits, leaving $a_{i,v_i}$ as the next block to evict. The final cache state is thus still well-formed and represents the new value of registers. Conversely, in case of a well-phased but not well-formed cache state (see Figure 3.12b), the gadget can not result in a well-formed cache state.

**Lemma 70.** *Executing a path through $\psi_{i,b} g_s$ over an NMRU state well-phased at step $s$ always leads to a state well-phased at step $1-s$. Furthermore that state*

- *either is not well-formed at step $1-s$*
- *or is identical to the initial state except for the $g_0$ and $g_1$ blocks, and, possibly, the $a_{i,\beta_i}$ block replaced by $a_{i,b}$.*

*Proof.* Again, the initial state is $x_1^0, \ldots, x_r^0, d^0, x_{r+1}^0, \ldots, x_{2r}^0, g_0^0, g_1^1$ where the $(x_i)_{1 \le i \le 2r}$ are a permutation of $\{e_i \mid 1 \le i \le r\} \cup \{a_{i,\beta_i} \mid 1 \le i \le r\}$ for some sequence of Booleans $(\beta_i)_{1 \le i \le r}$.

Consider a path through $\psi_{i,b}$: it consists of $d$, followed by a sequence of $r-1$ $a$'s, then $r$ $e$'s, then $a_{i,b}$. Each of these accesses either freshens, either overwrites, one of the $x$ positions. After the sequence of $a$'s, there are either no $a$'s to the left of $d$, or at least one. The former case is possible only if all these $a$'s are hits, freshening positions to the right of $d$. Then the sequence of $e$'s freshens the $e$'s to the left of $d$. There is one remaining $x$ position with a zero MRU-bit: it is to the right of $d$ and carries a block $a_{i,\beta_i}$. This block is then updated or freshened by the $a_{i,b}$ access.




(a) Assignement when the input state is well-formed (b) Assignement when the input is not well-formed

Figure 3.12 – Example of assignment execution

Then the access to $g_0$ flips all MRU-bits to $0$ except the one for $g_0$, which is flipped to $1$. Since all of the accesses before the $a_{i,b}$ access were hits, the permutation of the positions has not changed: the state is the same as the initial state except that $a^0_{i,\beta_i}$ is replaced by $a^0_{i,b}$ and $g_0^s g_1^{1-s}$ is replaced by $g_0^{1-s} g_1^s$.

Now consider the latter case: after the sequence of $a$'s there is at least one position of the form $a^1_{j,\beta}$ to the left of $d$. Then, as in the proof of the previous lemma, there is still $a^0_{j,\beta}$ to the left of $d$ at the end of the path through $\psi_{i,b} g_s$. Thus the final state cannot be well-formed. □

**Corollary 71.** *Assume starting in a well-formed NMRU state, corresponding to Boolean state $\sigma$, then any path through the gadget encoding an assignment or a guard*

- *either leads to a well-formed NMRU state, corresponding to the state $\sigma'$ obtained by executing the assignment, or $\sigma' = \sigma$ for a valid guard;*
- *or leads to a well-phased but not well-formed state.*

**Prologue**

As for the PLRU and FIFO replacement policies, we define a prologue that turns the empty cache state to a cache state representing the set of boolean registers with value false.

**Definition 72.** *From the cache analysis initial vertex $I_n$ to the register machine former initial vertex $I_r$ there is the prologue: the sequence of accesses $e_1 \ldots e_r d a_{1,\mathbf{f}} \ldots a_{r,\mathbf{f}} g_0 g_1$*

This sequence simply accesses the blocks we want to load in the cache, in the order we want to load them. After accessing $g_1$, all MRU-bit as reset, leading to cache state well-formed at step 0.

**Epilogue**

Finally, we propose to perform the final check on the block $d$. We thus need to turn well-formed cache states into cache states that contain $d$ (i.e. we should not evict it), and well-phased but not well-formed cache states into states that do not contain $d$ (i.e. it should be evicted).

**Definition 73.** *From the register machine former final vertex $F_r$ to a vertex $F_a$ there is a sequence of gadgets for the assignments $v_1 := 0 \dots v_r := 0$, the first part of the epilogue. From $F_a$ to a vertex $F_h$ there is a sequence of accesses $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}} c_1 \dots c_r$, the second part of the epilogue. The final vertex is $F_n = F_h$.*

**Lemma 74.** *Executing the sequence $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}} c_1 \dots c_r$ from $F_a$ to $F_h$ over a well-formed NMRU state corresponding to a zero Boolean state leads to a state containing $d$ — more specifically, a state of the form $c_1^1, \dots, c_r^1, d^0, a^1_{\pi(1),\mathbf{f}}, \dots, a^1_{\pi(r),\mathbf{f}} g_0^0 g_1^1$ where $\pi$ is a permutation.*

*Proof.* The $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}}$ just freshen the corresponding blocks (MRU-bit set to 1), and then the $c_1 \dots c_r$ overwrite the $e$'s. □

**Lemma 75.** *Executing the sequence $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}} c_1 \dots c_r$ from $F_a$ to $F_h$ over a well-phased but not well-formed NMRU state leads to a state not containing $d$ — where the $2r$ first MRU bits are set to $1$, the next one to $0$, and then $g_0^0 g_1^1$.*

*Proof.* The well-phased but not well-formed NMRU state contains at least one $a$ to the right of $d$. When applying $a_{1,\mathbf{f}} \dots a_{r,\mathbf{f}}$, at least one of the $a$'s must thus freshen or replace a letter to the left of $d$. Then when applying $c_1 \dots c_r$, $d$ gets evicted. □

**Corollary 76.** *There is an execution sequence from $I_n$, with empty cache, to $F_n$, such that the final cache contains $d$ if and only if there is an execution trace from $I_r$ to $F_r$.*

**Theorem 77.** *The exist-hit problem for NMRU caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

**Reduction to Exist-Miss**

**Definition 78.** *We modify the reduction of the exist-hit case as follows. Between $F_h$ and $F_n$ we insert the sequence $d g_0 c_1 \dots c_r a_{1,\mathbf{t}}$, as the third part of the epilogue.*

**Lemma 79.** *Executing $d g_0 c_1 \dots c_r a_{1,\mathbf{t}}$ over a state of the form $c_1^1, \dots, c_r^1, d^0, a^1_{\pi(1),\mathbf{f}}, \dots, a^1_{\pi(r),\mathbf{f}} g_0^0 g_1^1$, where $\pi$ is a permutation, leads to a state without $d$.*

*Proof.* $d$ gets freshened, then $g_0$ is the sole block with a zero MRU-bit. Thus, when it is freshened, all other MRU bits are set to zero. Then $c_1 \dots c_r$ freshen the first $r$ blocks, and $a_{1,\mathbf{t}}$ erases $d$. □

**Lemma 80.** *Executing $d g_0 c_1 \dots c_r a_{1,\mathbf{t}}$ over a state not containing $d$, where the $2r$ first MRU bits are set to $1$, the next one to $0$, and then $g_0^0 g_1^1$ leads to a state containing $d$.*

*Proof.* $d$ overwrites the $2r+1$-th position, then $g_0$ is the sole block with a zero MRU-bit. Thus, when it is freshened, all other MRU bits are set to zero. Then possibly some blocks get overwritten among the $r+1$ first blocks, and $d$ is still in the cache. □

**Corollary 81.** *There is an execution sequence from $I_n$, with empty cache, to $F_n$, such that the final cache does not contain $d$ if and only if there is an execution of the Boolean register machine from $I_r$ to $F_r$.*

**Theorem 82.** *The exist-miss problem for NMRU caches is NP-complete for acyclic graphs and PSPACE-complete for general graphs.*

Note that contrary to the FIFO and PLRU cases, the prologue we use heavily relies on the assumption of a cache initially empty. Thus, we have no results for reductions to cache analysis problems with arbitrary starting state. The proof method that we used for FIFO and PLRU — prepend a sufficiently long sequence of accesses that will bring the cache to a sufficiently known state — does not seem to easily carry over to NMRU caches. Even though it is known that $2k-2$ pairwise distinct accesses are sufficient to remove all previous content from an NMRU cache [RGBW07a, Th. 4], it can be shown that there is no sequence guaranteed to yield a completely known cache state [RGBW07a, Th. 5].

# Chapter 4

# Exact Cache Analysis

An ideal cache analysis would statically classify every memory access at every machine-code instruction in a program into one of three cases: i) the access is a cache hit in all possible executions of the program ii) the access is a cache miss in all possible executions of the program iii) in some executions the access is a hit and in others it is a miss. However, no cache analysis can perfectly classify all accesses into these three categories.

One first reason is that perfect cache analysis would involve testing the reachability of individual program statements, which is undecidable.[1] A simplifying assumption often used, including in this thesis, is that all program paths are feasible—this is safe, since it overapproximates possible program behaviors. Even with this assumption, analysis is usually performed using sound but incomplete abstractions (see Chapter 2.3) that can safely determine that some accesses always hit ("∀Hit" in Figure 4.1) or always miss ("∀Miss" in Fig. 4.1). The corresponding analyses are called *may* and *must* analysis and referred to as "classical AI" in Fig. 4.1. Due to incompleteness the status of other accesses however remains "unknown" (Fig. 4.1).

In this chapter, we propose an approach to eliminate this uncertainty in the case of LRU caches. Our first contribution (colored red in Figure 4.1), is a novel abstract interpretation, called *Definitely Unknown analysis*, that safely concludes that certain accesses are hits in some executions ("∃Hit"), misses in some executions ("∃Miss"), or hits in some and misses in other executions ("∃Hit ∧ ∃Miss" in Fig. 4.1). Using this analysis and prior must- and may- cache analyses, most accesses are precisely classified.

Our second contribution consists in two approaches able to refine the remaining accesses (accesses that are not classified due to a lack of precision of the previous analyses). The first approach consists in encoding the classification problem in a model checker and takes advantage of the small number of unclassified accesses to analyze them one by one. This allows us to use two method called *block focusing*, that abstract the cache behavior relatively to the analyzed cache block. The second approach also uses the *block focusing* abstraction, but perform additional simplifications and relies on Zero-Suppressed Decision Diagrams for an efficient implementation of these simplifications. Similarly to the model checking approach, this second approach (formalized as an abstract interpretation analysis) provides an exact classification (colored green in Figure 4.1) of the memory accesses.

Our analysis steps are summarized in Figure 4.2. Based on the control-flow graph and on an initial cache configuration, the abstract-interpretation phase classifies some of the accesses as "always hit", "always miss" and "definitely unknown". Those accesses are already precisely classified and thus do not require a refinement step. The AI phase thus reduces the number of

[1] One may object that given that we consider machine-level aspects, memory is bounded and thus properties are decidable. The time and space complexity is however prohibitive.

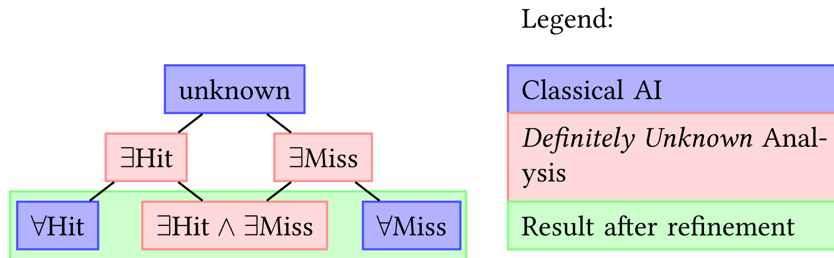



Figure 4.1 – Possible classifications of classical abstract-interpretation-based cache analysis, our new abstract interpretation, and after refinement by model checking.

accesses to be refined. In addition, the results of the AI phase are used to simplify the refinement step. For instance, if the refinement step is performed using a model-checker, a simplified CCG can be used, as discussed in detail in Section 4.2.

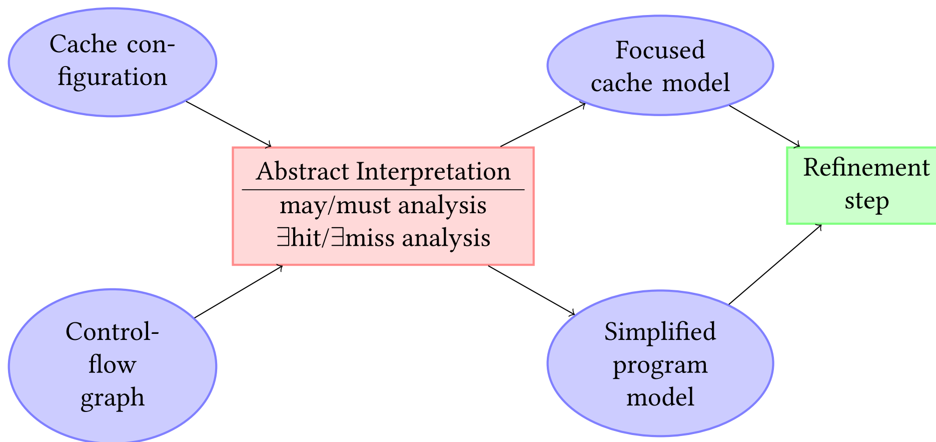



Figure 4.2 – Overall analysis flow.

## 4.1 Approximating the set of Definitely Unknown Accesses

Together, may and must analysis can classify accesses as “always hit”, “always miss” or “unknown” (see 2.3). An access classified as “unknown” may still be “always hit” or “always miss” but not detected as such due to the imprecision of the abstract analysis; otherwise it is “definitely unknown”. This section proposes an abstract analysis that safely establishes that some blocks are “definitely unknown” under LRU replacement.

### 4.1.1 Reminder: Caches and Static Cache Analysis

This chapter focuses on the analysis of LRU instruction caches. As mentioned in Chapter 2.1, the state of an LRU cache can be modeled by a mapping that assigns to each memory block its age, where ages are truncated at $k$, i.e., we do not distinguish ages of uncached blocks. Similarly, all paths in that graph are considered feasible, even if, taking into account the instruction semantics, they are not—e.g. a path including the tests $x \leq 4$ and $x \geq 5$ in immediate succession is considered feasible even though the two tests are mutually exclusive. All our claims of completeness are relative to this model.

The effect of an access to memory block $b \in \mathit{Blocks}$ under LRU replacement is then formalized as follows[2]:

$$
\begin{aligned}
\mathit{update} : D_{LRU} \times \mathit{Blocks} &\to D_{LRU} \\
(q, b) &\mapsto q' \\
\text{where } \forall b' \in \mathit{Blocks}, q'(b') &= \begin{cases} 0 & \text{if } b' = b \\ q(b') & \text{if } b \neq b' \wedge q(b') > q(b) \\ q(b') + 1 & \text{if } b \neq b' \wedge q(b') \leq q(b) \wedge q(b') < k \\ k & \text{otherwise} \end{cases}
\end{aligned}
$$

As usual, we abstract the program under analysis by its cache conflict graph.

### Collecting Semantics

In order to classify memory accesses as "always hit" or "always miss", cache analysis needs to characterize for each control location in a program *all* cache states that may reach that location in any execution of the program. This is commonly called the *collecting semantics*.

Given a control-flow graph $G = (V, E, v_0)$, the *collecting semantics* is defined as the least solution to the following set of equations, where $F : V \to D_{LRU}$ denotes the set of reachable concrete cache configurations at each program location, and $F_0(v)$ denotes the set of possible initial cache configurations:

$$
\forall v' \in V : F(v') = F_0(v') \cup \bigcup_{(v,v') \in E} \mathit{update}(F(v), \mathit{blocks}(b)), \tag{4.1}
$$

where $\mathit{blocks}$ gives the sequence of memory blocks forming a basic block and $\mathit{update}$ has been lifted pointwise from $D_{LRU}$ to $\mathcal{P}(D_{LRU})$ and from single memory block to sequence of blocks.

Explicitly computing the collecting semantics is practically infeasible. For a tractable analysis, it is necessary to operate in an abstract domain whose elements compactly represent large sets of concrete cache states.

### Classical Abstract Interpretation of LRU Caches

To this end, the classical abstract interpretation of LRU caches [AFMW96] assigns to every memory block at every program location an interval of ages enclosing the possible ages of the block during any program execution. These two analyses are detailed in Section 2.3. The following is only a reminder about the notation used. The analysis for upper bounds, or *must analysis*, can prove that a block must be in the cache; conversely, the one for lower bounds, or *may analysis*, can prove that a block may not be in the cache. For instance, on Figure 4.3, the Must analysis ensures that the last access to $f$ always results in a hit and the May analysis guarantees that the first access to each block is always a miss.

The domains for abstract cache states under may and must analysis are $D_{May} = D_{Must} = D_{LRU} = \mathit{Blocks} \to \{0, ..., k\}$, where ages greater than or equal to the cache's associativity $k$ are truncated at $k$ as in the concrete domain. The set of concrete cache states represented by abstract cache states is given by the concretization function:

$$
\begin{aligned}
\gamma_{May}(\hat{q}_{May}) &= \{q \in D_{LRU}, \forall b \in \mathit{Blocks} : \hat{q}_{May}(b) \leq q(b)\} \\
\gamma_{Must}(\hat{q}_{Must}) &= \{q \in D_{LRU}, \forall b \in \mathit{Blocks} : q(b) \leq \hat{q}_{Must}(b)\}
\end{aligned}
$$

[2]Assuming for simplicity that all cache blocks map to the same cache set.

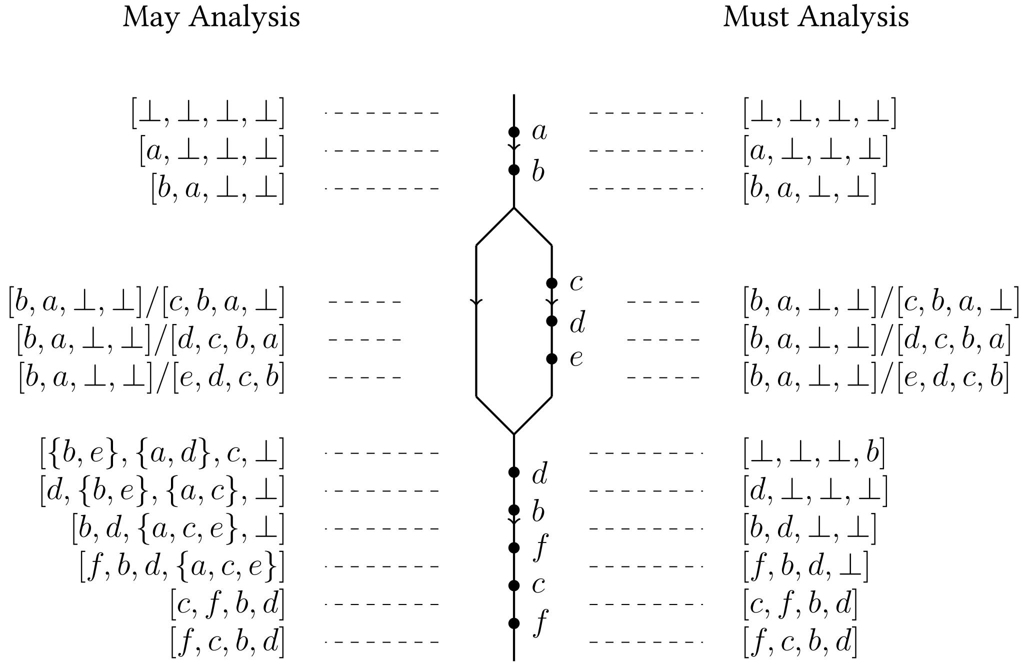



Figure 4.3 – Example of May and Must analyses

Abstract cache states can be joined by taking their pointwise extrema:

$$\hat{q}_{Must,1} \sqcup_{Must} \hat{q}_{Must,2} = \hat{q}_{Must3}, \text{ where } \forall b \in \mathit{Blocks}, \hat{q}_{Must3}(b) = \max\{\hat{q}_{Must,1}(b), \hat{q}_{Must,2}(b)\}$$
$$\hat{q}_{May,1} \sqcup_{May} \hat{q}_{May,2} = \hat{q}_{May3}, \text{ where } \forall b \in \mathit{Blocks}, \hat{q}_{May3}(b) = \min\{\hat{q}_{May,1}(b), \hat{q}_{May,2}(b)\}$$

Suitably defined abstract semantics $F^{\#}_{Must}$ and $F^{\#}_{May}$ can be shown to overapproximate their concrete counterpart:

**Theorem 83** (Analysis Soundness [AFMW96])**.** *The may and the must abstract semantics are safe approximations of the collecting semantics:*

$$\forall v \in V, F(v) \subseteq \gamma_{Must}(F^{\#}_{must}(v)) \wedge F(v) \subseteq \gamma_{May}(F^{\#}_{May}(v)). \tag{4.2}$$

The may and must abstract transformers are in fact the best abstract transformers defined by the abstraction/concretization pairs [Rei09].

### 4.1.2 Abstract Interpretation for Definitely Unknown

An access is "definitely unknown" if there is a concrete execution in which the access misses and another in which it hits. The aim of our analysis is to prove the existence of such executions to classify an access as "definitely unknown". Note the difference with classical may/must analysis and most other abstract interpretations, which compute properties that hold *for all executions*, while here we seek to prove that *there exist* two executions with suitable properties.

An access to a block $a$ results in a hit if $a$ has been accessed recently, i.e., $a$'s age is low. Thus we would like to determine the minimal age that $a$ may have in a reachable cache state immediately prior to this access. The access can be a hit if and only if this minimal age is lower than the cache's associativity. Because we cannot efficiently compute exact minimal ages, we devise an *Exists Hit* (EH) analysis to compute safe upper bounds on minimal ages. Similarly, to be sure there is an execution in which accessing $a$ results in a miss, we compute a safe lower bound on the maximal age of $a$ using the *Exists Miss* (EM) analysis.

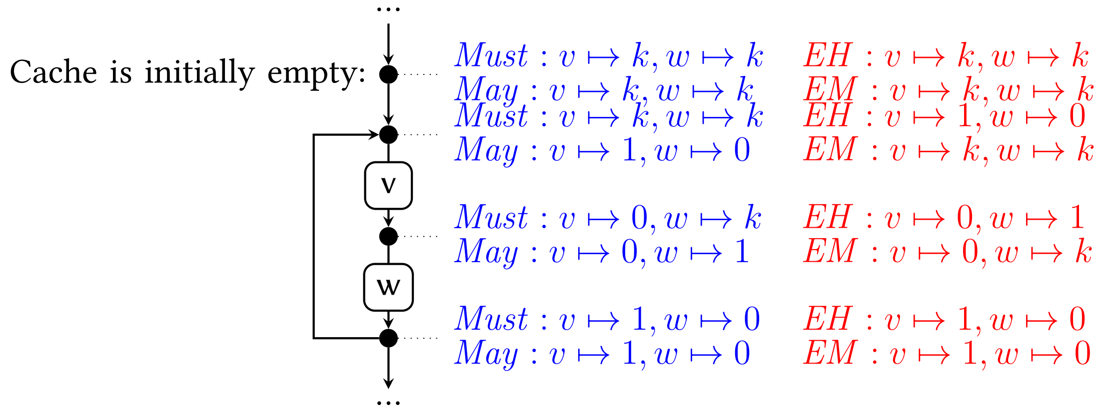



Figure 4.4 – Example of two accesses in a loop that are definitely unknown. May/Must and EH/EM analysis results are given next to the respective control locations.

**Example.** Let us now consider a small example. In Figure 4.4, we see a small control-flow graph corresponding to a loop that repeatedly accesses memory blocks $v$ and $w$. Assume the cache is empty before entering the loop. Then, the accesses to $v$ and $w$ are definitely unknown in fully-associative caches of associativity 2 or greater: they both miss in the first loop iteration, while they hit in all subsequent iterations. Applying standard may and must analysis, both accesses are soundly classified as “unknown”. On the other hand, applying the EH analysis, we can determine that there are cases where $v$ and $w$ hit. Similarly, the EM analysis derives that there exist executions in which they miss. Combining those two results, the two accesses can safely be classified as definitely unknown. Note that this example could be solved by unrolling the first iteration of the loop. Indeed, accesses in the first iteration always results in a miss, whereas further iterations always lead to hits. This method is however not applicable if the definitely unknown access is not in a loop and is not always successful (see Section 4.1.4).

We will now define these *Exist-Hit* and *Exist-Miss* analyses and their underlying domains more formally. The EH analysis maintains upper bounds on the minimal ages of blocks. Consider for instance the Exist-Hit abstract cache $[a, b, c, d]$. This abstract cache gives the information that $a$ has age $0$ for at least one path, i.e. there is at least one path on which $a$ is the least recently used block. Similarly, there is at least one path to the current location where $b$ has an age less than or equal to 1. This path might be the same than the one where $a$ has age $0$, but it might be a different one. Now consider an access to block $a$. The new value associated to $a$ will be 0, because it still has age 0 on at least one path (in fact, it now has age 0 for any paths). The main question is what should the new upper-bound associated to $b$ be? We know that $b$ has an age less than or equal to 1 on at least one path. On this path, accessing $a$ can increase the age of $b$ of at most 1. It is thus safe to consider that there is now at least one path where $b$ has an age less than or equal to 2. In other words, while $a$ has age 0 in some state and $b$ has age 1 in some state, these are not necessarily the same state. Thus, accessing $a$ forces us to consider that $b$ might now have age 2.

In order to give more precise information about $b$, i.e. to claim that $b$ has age less than or equal to 1 on at least one path, one should guarantee that accessing $a$ did not increase the age of $b$ on the path we consider. More precisely, one should ensure that $a$ was already accessed after $b$ on the same path, and this information is in general not given by the exist hit abstract state (the information it gives about $a$ can concern another path). One should thus look at the must abstract cache state. If the must analysis ensures that $a$ has age at most 0, then one would know that $a$ was accessed more recently than $b$ on the path where it has age $1$, and accessing $a$ again would not

increase the age of $b$. Thus, our EH analysis includes a must analysis to obtain upper bounds on all possible ages of blocks, which are required for precise updates. Then, the domain for abstract cache states under the EH analysis is $D_{EH} = (Blocks \to \{0,\ldots,k\}) \times D_{Must}$. Similarly, the EM analysis maintains lower bounds on the maximal ages of blocks and includes a regular may analysis: $D_{EM} = (Blocks \to \{0,\ldots,k\}) \times D_{May}$.

The properties we wish to establish, i.e. bounds on minimal and maximal ages, are actually *hyperproperties* [CS08]: they are not properties of individual reachable states but rather of the entire *set* of reachable states. Indeed, the EH analysis is used to make claims of the form "for the given block $a$, there *exists* (at least) one reachable state among the *set* of reachable states where $a$ has age greater than $i$" (for some arbitrary $i$). Thus, the conventional approach in which abstract states concretize to sets of concrete states that are a superset of the actual set of reachable states is not applicable. Instead, we express the meaning, $\gamma_{EH}$, of abstract states by *sets of sets* of concrete states. A set of states $Q$ is represented by an abstract EH state $(\hat{q}_{EH}, \hat{q}_{Must})$, if for each block $b$, $\hat{q}_{EH}(b)$ is an upper bound on $b$'s minimal age in $Q$, $\min_{q\in Q} q(b)$:

$$\begin{aligned} &\gamma_{EH} : D_{EH} \to \mathcal{P}(D_{LRU}) \\ &(\hat{q}_{EH}, \hat{q}_{Must}) \mapsto \left\{Q \subseteq \gamma_{Must}(\hat{q}_{Must}), \forall b \in Blocks : \min_{q\in Q} q(b) \leq \hat{q}_{EH}(b)\right\} \end{aligned}$$

Conversely, the Exist-Miss uses the following concretization function:

$$\begin{aligned} &\gamma_{EM} : D_{EM} \to \mathcal{P}(D_{LRU}) \\ &(\hat{q}_{EM}, \hat{q}_{May}) \mapsto \left\{Q \subseteq \gamma_{May}(\hat{q}_{May}), \forall b \in Blocks : \hat{q}_{EM}(b) \leq \max_{q\in Q} q(b)\right\} \end{aligned}$$

The actual set of reachable states is an element rather than a subset of this concretization. The concretizations for the may and must analyses, $\gamma_{May}$ and $\gamma_{Must}$, are simply lifted to this setting. Also note that—possibly contrary to initial intuition—our abstraction cannot be expressed as an underapproximation, as different blocks' minimal ages may be attained in different concrete states.

The abstract transformer $update_{EH}((\hat{q}_{EH}, \hat{q}_{Must}), b)$ corresponding to an access to block $b$ is the pair $(\hat{q}'_{EH}, update_{Must}(\hat{q}_{Must}, b))$, where

$$\forall b' \in Blocks, \hat{q}'_{EH}(b') = \begin{cases} 0 & \text{if } b' = b \\ \hat{q}_{EH}(b') & \text{if } \hat{q}_{Must}(b) \leq \hat{q}_{EH}(b') \\ \hat{q}_{EH}(b') + 1 & \text{if } \hat{q}_{Must}(b) > \hat{q}_{EH}(b') \wedge \hat{q}_{EH}(b') < k \\ k & \text{if } \hat{q}_{Must}(b) > \hat{q}_{EH}(b') \wedge \hat{q}_{EH}(b') = k \end{cases}$$

Let us explain the four cases in the transformer above. After an access to $b$, $b$'s age is 0 in all possible executions. Thus, 0 is also a safe upper bound on its minimal age (case 1). The access to $b$ may only increase the ages of younger blocks (because of the LRU replacement policy). In the cache state in which $b'$ attains its minimal age, it is either younger or older than $b$. If it is younger, then the access to $b$ may increase $b'$'s actual minimal age, but not beyond $\hat{q}_{Must}(b)$, which is a bound on $b$'s age in every cache state, and in particular in the one where $b'$ attains its minimal age. Otherwise, if $b'$ is older, its minimal age remains the same and so may its bound. This explains why the bound on $b'$'s minimal age does not increase in case 2. Otherwise, for safe upper bounds, in cases 3 and 4, the bound needs to be increased by one, unless it has already reached $k$. For instance, the abstract state $([\bot, b, a, \bot]_{Must}, [\{b, c\}, a, \bot, \bot]_{EH})$ would be updated to $([b, \bot, a, \bot]_{Must}, [b, c, a, \bot]_{EH})$ on an access to $b$.

similarly, the abstract transformer $update_{EM}((\hat{q}_{EM}, \hat{q}_{May}), b)$ corresponding to an access to block $b$ is the pair $(\hat{q}'_{EM}, update_{May}(\hat{q}_{May}, b))$, where

$$\forall b' \in Blocks, \hat{q}'_{EM}(b') = \begin{cases} 0 & \text{if } b' = b \\ \hat{q}_{EM}(b') & \text{if } \hat{q}_{May}(b) < \hat{q}_{EM}(b') \\ \hat{q}_{EM}(b') + 1 & \text{if } \hat{q}_{May}(b) \geq \hat{q}_{EM}(b') \wedge \hat{q}_{EM}(b') < k \\ k & \text{if } \hat{q}_{May}(b) \geq \hat{q}_{EM}(b') \wedge \hat{q}_{EM}(b') = k \end{cases}$$

The followings theorem express the consistency of our abstract transformers:

**Lemma 84** (EH Local Consistency)**.** *The abstract transformer* $update_{EH}$ *soundly approximates its concrete counterpart* $update$*:*

$$\forall (\hat{q}_{EH}, \hat{q}_{Must}) \in D_{EH}, \forall b \in Blocks, \forall Q \in \gamma_{EH}(\hat{q}_{EH}, \hat{q}_{Must}) : \\ update(Q, b) \in \gamma_{EH}(update_{EH}((\hat{q}_{EH}, \hat{q}_{Must}), b)). \quad (4.3)$$

**Lemma 85** (EM Local Consistency)**.** *The abstract transformer* $update_{EM}$ *soundly approximates its concrete counterpart* $update$*:*

$$\forall (\hat{q}_{EM}, \hat{q}_{May}) \in D_{EM}, \forall b \in Blocks, \forall Q \in \gamma_{EM}(\hat{q}_{EM}, \hat{q}_{May}) : \\ update(Q, b) \in \gamma_{EM}(update_{EM}((\hat{q}_{EM}, \hat{q}_{May}), b)). \quad (4.4)$$

Concerning the merging of EH (respectively EM) states at control-flow joins, the standard must (respectively may) join can be applied for the must (respectively may) analysis component. In the concrete, the union of the states reachable along all incoming control paths is reachable after the join. It is thus safe to take the *minimum* (respectively *maximum*) of the upper (respectively lower) bounds on minimal (respectively maximal) ages for the EH (respectively EM) part:

$$(\hat{q}_{EH,1}, \hat{q}_{Must,1}) \sqcup_{EH} (\hat{q}_{EH,2}, \hat{q}_{Must,2}) = (\hat{q}_{EH3}, \hat{q}_{Must,1} \sqcup_{Must} \hat{q}_{Must,2}) \\ \text{where } \forall b \in Blocks, \hat{q}_{EH3}(b) = \min(\hat{q}_{EH,1}(b), \hat{q}_{EH,2}(b))$$

$$(\hat{q}_{EM,1}, \hat{q}_{May,1}) \sqcup_{EM} (\hat{q}_{EM,2}, \hat{q}_{May,2}) = (\hat{q}_{EM3}, \hat{q}_{May,1} \sqcup_{May} \hat{q}_{May,2}) \\ \text{where } \forall b \in Blocks, \hat{q}_{EM3}(b) = \max(\hat{q}_{EM,1}(b), \hat{q}_{EM,2}(b))$$

One can then formulate the correctness of the join operators as follows:

**Lemma 86** (EH Join Consistency)**.** *The join operator* $\sqcup_{EH}$ *is correct:*

$$\forall ((\hat{q}_{EH,1}, \hat{q}_{Must,1}), (\hat{q}_{EH,2}, \hat{q}_{Must,2})) \in D^2_{EH}, \forall Q_1 \in \gamma_{EH}(\hat{q}_{EH,1}, \hat{q}_{Must,1}), \forall Q_2 \in \gamma_{EH}(\hat{q}_{EH,2}, \hat{q}_{Must,2}) : \\ Q_1 \cup Q_2 \in \gamma_{EH}((\hat{q}_{EH,1}, \hat{q}_{Must,1}) \sqcup_{EH} (\hat{q}_{EH,2}, \hat{q}_{Must,2})). \quad (4.5)$$

**Lemma 87** (EM Join Consistency)**.** *The join operator* $\sqcup_{EM}$ *is correct:*

$$\forall ((\hat{q}_{EM,1}, \hat{q}_{May,1}), (\hat{q}_{EM,2}, \hat{q}_{May,2})) \in D^2_{EM}, \forall Q_1 \in \gamma_{EM}(\hat{q}_{EM,1}, \hat{q}_{May,1}), Q_2 \in \gamma_{EM}(\hat{q}_{EM,2}, \hat{q}_{May,2}) : \\ Q_1 \cup Q_2 \in \gamma_{EM}((\hat{q}_{EM,1}, \hat{q}_{May,1}) \sqcup_{EM} (\hat{q}_{EM,2}, \hat{q}_{May,2})). \quad (4.6)$$

Given a control-flow graph $G = (V, E, v_0)$, the *abstract EH semantics* is defined as the least solution to the following set of equations, where $F^{\#}_{EH} : V \to D_{EH}$ denotes the abstract cache configuration associated with each program location, and $F^{\#}_{EH0}(v) \in \gamma_{EH}(F_{EH0}(v))$ denotes the initial abstract cache configuration:

$$\forall v' \in V : F^{\#}_{EH}(v') = F^{\#}_{EH0}(v') \sqcup_{EH} \bigsqcup_{(v,v')\in E} update_{EH}(F^{\#}_{EH}(v), blocks(v)). \tag{4.7}$$

The exist miss counterpart of the exist hit abstract semantics is defined by the same fixpoint equation, where all EH are replaced by EM.

It follows from Lemmas 84 and 86 that the abstract EH semantics includes the actual set of reachable concrete states:

**Theorem 88** (Analysis Soundness)**.** *The collecting semantics is a member of both the EH and EM abstract semantics:*

$$\forall v \in V : F(v) \in \gamma_{EH}(F^{\#}_{EH}(v)) \cap \gamma_{EM}(F^{\#}_{EM}(v))$$

We can use the results of the EH analysis to determine that an access results in a hit in at least some of all possible executions. This is the case if the minimum age of the block prior to the access is guaranteed to be less than the cache's associativity. Similarly, the EM analysis can be used to determine that an access results in a miss in at least some of the possible executions.

Combining the results of the two analyses, some accesses can be classified as *Definitely Unknown*. Then, further refinement is provably impossible under the hypothesis that all paths are feasible. Classifications as *Exist-Hit* or *Exist-Miss*, which occur if either the EH or the EM analysis is successful but not both, are also useful to reduce further refinement efforts. Indeed, in case of *Exist-Hit* one knows for sure that the access can not be refined as *Always-Miss*. It suffices to determine by any method that a miss is possible to fully classify the access.

### 4.1.3 Definitely Unknown Proofs

As mentioned earlier, the soundness of the EH analysis comes from the consistency of the abstract transformers with respect to the concrete transformers. Here are the detailed proofs for the EH analysis.

**Lemma 84** (EH Local Consistency)**.** *The abstract transformer* $update_{EH}$ *soundly approximates its concrete counterpart* $update$*:*

$$\forall(\hat{q}_{EH}, \hat{q}_{Must}) \in D_{EH}, \forall b \in Blocks, \forall Q \in \gamma_{EH}(\hat{q}_{EH}, \hat{q}_{Must}) : \\ update(Q, b) \in \gamma_{EH}(update_{EH}((\hat{q}_{EH}, \hat{q}_{Must}), b)). \tag{4.3}$$

*Proof.* Consistency of EH Analysis

Let $(\hat{q}_{EH,0}, \hat{q}_{Must,0}) \in D_{EH}$ and $b \in Blocks$. We use the following additional notations:

- $(\hat{q}_{EH,1}, \hat{q}_{Must,1}) = update_{EH}((\hat{q}_{EH,0}, \hat{q}_{Must,0}), b)$
- $\mathcal{Q}_1 = \gamma_{EH}(\hat{q}_{EH,1}, \hat{q}_{Must,1})$
- $\mathcal{Q}_0 = \gamma_{EH}(\hat{q}_{EH,0}, \hat{q}_{Must,0})$
- $\mathcal{Q}_2 = \{update(Q, b), Q \in \mathcal{Q}_0\}$

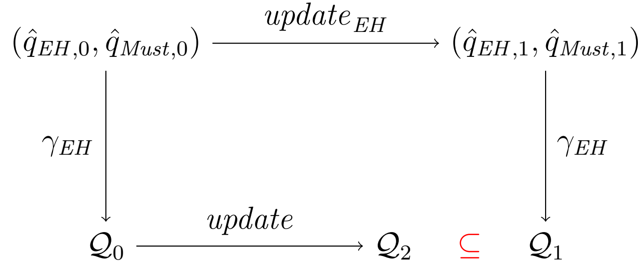



Figure 4.5 – Notations used

These notations are summarized on Figure 4.5. We want to prove the inclusion $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$, as shown in red on the figure. To prove that this inclusion holds, we prove that any $\widetilde{Q}_2$ in $\mathcal{Q}_2$ belongs to $\mathcal{Q}_1$. Let $\widetilde{Q}_2 \in \mathcal{Q}_2$ and $\widetilde{Q}_0 \in \mathcal{Q}_0$ such that $\widetilde{Q}_2 = update(\widetilde{Q}_0, b)$. We first prove that $\widetilde{Q}_2$ is covered by the must concretization of $\hat{q}_{Must,0}$. First, we have:

$$\forall \widetilde{q}_0 \in \widetilde{Q}_0, update(\widetilde{q}_0, b) \in \{update(q, b), q \in \gamma_{Must}(\hat{q}_{Must,0})\}$$

Thus, by the consistency of the must analysis, we have:

$$\forall \widetilde{q}_0 \in \widetilde{Q}_0, update(\widetilde{q}_0, b) \subseteq \gamma_{Must}(\{update_{Must}(\hat{q}_{Must,0}, b)\}) = \gamma_{Must}(\hat{q}_{Must,1})$$

Then: $\widetilde{Q}_2 \subseteq \gamma_{Must}(\hat{q}_{Must,1})$. To complete the proof that $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$, it remains to prove that:

$$\forall b' \in Blocks, \exists q \in \widetilde{Q}_2, \text{ such that: } q(b') \leq \hat{q}_{EH,1}(b')$$

Let $b' \in Blocks$. We have:

$$\hat{q}_{EH,1}(b') = \begin{cases} 0 & \text{if } b = b' \\ \hat{q}_{EH,0}(b') & \text{if } \hat{q}_{Must,0}(b) \leq \hat{q}_{EH,0}(b') \\ \hat{q}_{EH,0}(b') + 1 & \text{if } \hat{q}_{Must,0}(b) > \hat{q}_{EH,0}(b') \wedge \hat{q}_{EH,0}(b') < k \\ k & \text{if } \hat{q}_{Must,0}(b) > \hat{q}_{EH,0}(b') \wedge \hat{q}_{EH,0}(b') = k \end{cases}$$

Let $\widetilde{q}_0 \in \widetilde{Q}_0$ such that $\widetilde{q}_0(b') \leq \hat{q}_{EH,0}(b')$. Let $\widetilde{q}_2 = update(\widetilde{q}_0, b) \in \widetilde{Q}_2$. We show that $\widetilde{q}_2$ is a good candidate, i.e. $\widetilde{q}_2(b') \leq \hat{q}_{EH,1}(b')$.

$$\widetilde{q}_2(b') = \begin{cases} 0 & \text{if } b = b' \\ \widetilde{q}_0(b') & \text{if } \widetilde{q}_0(b') \geq \widetilde{q}_0(b) \\ \widetilde{q}_0(b') + 1 & \text{if } \widetilde{q}_0(b') < \widetilde{q}_0(b) \wedge \widetilde{q}_0(b') < k \\ k & \text{if } \widetilde{q}_0(b') < \widetilde{q}_0(b) \wedge \widetilde{q}_0(b') = k \end{cases}$$

We then proceed by case disjunction:

- If $b = b'$: $\widetilde{q}_2(b') = 0 = \hat{q}_{EH,1}(b')$.
- If $b \neq b' \wedge \hat{q}_{Must,0}(b) \leq \hat{q}_{EH,0}(b')$, then $\hat{q}_{EH,1}(b') = \hat{q}_{EH,0}(b')$.
  - If $\widetilde{q}_0(b') \geq \hat{q}_{Must,0}(b)$, then: $\widetilde{q}_0(b') \geq \widetilde{q}_0(b)$.
    Thus: $\widetilde{q}_2(b') = \widetilde{q}_0(b')$.
    Finally: $\hat{q}_{EH,1}(b') = \hat{q}_{EH,0}(b') \geq \widetilde{q}_0(b') = \widetilde{q}_2(b')$

- Otherwise, $\widetilde{q}_0(b') < \hat{q}_{Must,0}(b)$ and thus: $\widetilde{q}_0(b') < \hat{q}_{EH,0}(b')$.
  - if $\widetilde{q}_0(b') \geq \widetilde{q}_0(b)$, then: $\widetilde{q}_2(b') = \widetilde{q}_0(b') < \hat{q}_0(b') \leq \hat{q}_{EH,1}(b')$
  - if $\widetilde{q}_0(b') < \widetilde{q}_0(b)$, then: $\widetilde{q}_0(b') < k$ and thus: $\widetilde{q}_2(b') = \widetilde{q}_0(b') + 1 \leq \hat{q}_{EH,0}(b') \leq \hat{q}_{EH,1}(b')$

- If $b \neq b' \wedge \hat{q}_{Must,0}(b) > \hat{q}_{EH,0}(b') \wedge \hat{q}_{EH,0}(b') < k$, then $\hat{q}_{EH,1}(b') = \hat{q}_{EH,0}(b') + 1$. Moreover, $\widetilde{q}_0(b') \leq \hat{q}_{EH,0}(b') < k$.

  Thus: $\widetilde{q}_2(b') \leq \widetilde{q}_0(b') + 1 \leq \hat{q}_{EH,0}(b') + 1 = \hat{q}_{EH,1}(b')$

- Otherwise, $b \neq b' \wedge \hat{q}_{Must,0}(b) > \hat{q}_{EH,0}(b') \wedge \hat{q}_{EH,0}(b') = k$. Then: $\hat{q}_{EH,1}(b') = l$ and trivially: $\widetilde{q}_2(b') \leq \hat{q}_{EH,1}(b')$

In all cases, we have: $\widetilde{q}_2(b') \leq \hat{q}_{EH,1}(b')$. Thus, $\widetilde{Q}_2 \in \mathcal{Q}_1$, proving that $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$ □

The proof of the consistency of the EM update transformer is the dual of the proof the EH consistency.

**Lemma 85** (EM Local Consistency)**.** *The abstract transformer* $update_{EM}$ *soundly approximates its concrete counterpart* $update$*:*

$$\forall (\hat{q}_{EM}, \hat{q}_{May}) \in D_{EM}, \forall b \in Blocks, \forall Q \in \gamma_{EM}(\hat{q}_{EM}, \hat{q}_{May}) : \\ update(Q, b) \in \gamma_{EM}(update_{EM}((\hat{q}_{EM}, \hat{q}_{May}), b)). \quad (4.4)$$

*Proof.* Consistency of EM Analysis

Let $(\hat{q}_{EM,0}, \hat{q}_{May,0}) \in D_{EM}$ and $b \in Blocks$. We use the additional notations:

- $(\hat{q}_{EM,1}, \hat{q}_{May,1}) = update_{EM}((\hat{q}_{EM,0}, \hat{q}_{May,0}), b)$
- $\mathcal{Q}_1 = \gamma_{EM}(\hat{q}_{EM,1}, \hat{q}_{May,1})$
- $\mathcal{Q}_0 = \gamma_{EM}(\hat{q}_{EM,0}, \hat{q}_{May,0})$
- $\mathcal{Q}_2 = \{update(Q, b), Q \in \mathcal{Q}_0\}$

These notations are summarized on Figure 4.6. We want to prove the inclusion $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$, as shown

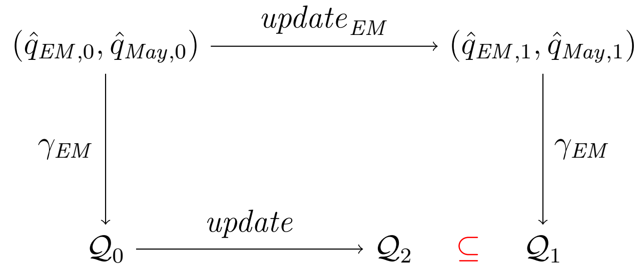



Figure 4.6 – Notations used

in red on the figure. To prove that this inclusion holds, we prove that any $\widetilde{Q}_2$ in $\mathcal{Q}_2$ belongs to $\mathcal{Q}_1$. Let $\widetilde{Q}_2 \in \mathcal{Q}_2$ and $\widetilde{Q}_0 \in \mathcal{Q}_0$ such that $\widetilde{Q}_2 = update(\widetilde{Q}_0, b)$. We first prove that $\widetilde{Q}_2$ is covered by the must concretization of $\hat{q}_{Must,0}$. First, we have:

$$\forall \widetilde{q}_0 \in \widetilde{Q}_0, update(\widetilde{q}_0, b) \in \{update(q, b), q \in \gamma_{May}(\hat{q}_{May,0})\}$$

Thus, by the consistency of the may analysis, we have:

$$\forall \widetilde{q}_0 \in \widetilde{Q}_0, \mathit{update}(\widetilde{q}_0, b) \subseteq \gamma_{May}(\{\mathit{update}_{May}(\hat{q}_{May,0}, b)\}) = \gamma_{May}(\hat{q}_{May,1})$$

Then: $\widetilde{Q}_2 \subseteq \gamma_{May}(\hat{q}_{May,1})$.

To complete the proof that $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$, it remains to prove that:

$$\forall b' \in \mathit{Blocks}, \exists q \in \widetilde{Q}_2, \text{ such that: } q(b') \leq \hat{q}_{EM,1}(b')$$

Let $b' \in \mathit{Blocks}$. We have:

$$\hat{q}_{EM,1}(b') = \begin{cases} 0 & \text{if } b = b' \\ \hat{q}_{EM,0}(b') & \text{if } \hat{q}_{May,0}(b) < \hat{q}_{EM,0}(b') \\ \hat{q}_{EM,0}(b') + 1 & \text{if } \hat{q}_{May,0}(b) \geq \hat{q}_{EM,0}(b') \wedge \hat{q}_{EM,0}(b') < k \\ k & \text{if } \hat{q}_{May,0}(b) \geq \hat{q}_{EM,0}(b') \wedge \hat{q}_{EM,0}(b') = k \end{cases}$$

Let $\widetilde{q}_0 \in \widetilde{Q}_0$ such that $\widetilde{q}_0(b') \geq \hat{q}_{EM,0}(b')$. Let $\widetilde{q}_2 = \mathit{update}(\widetilde{q}_0, b) \in \widetilde{Q}_2$. We show that $\widetilde{q}_2$ is a good candidate, i.e. $\widetilde{q}_2(b') \geq \hat{q}_{EM,1}(b')$.

$$\widetilde{q}_2(b') = \begin{cases} 0 & \text{if } b = b' \\ \widetilde{q}_0(b') & \text{if } \widetilde{q}_0(b') \geq \widetilde{q}_0(b) \\ \widetilde{q}_0(b') + 1 & \text{if } \widetilde{q}_0(b') < \widetilde{q}_0(b) \wedge \widetilde{q}_0(b') < k \\ k & \text{if } \widetilde{q}_0(b') < \widetilde{q}_0(b) \wedge \widetilde{q}_0(b') = k \end{cases}$$

We then proceed by case disjunction:

- If $b = b'$: $\widetilde{q}_2(b') = 0 = \hat{q}_{EM,1}(b')$.
- If $b \neq b' \wedge \hat{q}_{May,0}(b) < \hat{q}_{EM,0}(b')$, then $\hat{q}_{EM,1}(b') = \hat{q}_{EM,0}(b')$.
  Moreover, $b \neq b' \Rightarrow \widetilde{q}_2(b') \geq \widetilde{q}_0(b') \geq \hat{q}_{EM,0}(b') = \hat{q}_{EM,1}(b')$.
- If $b \neq b' \wedge \hat{q}_{May,0}(b) \geq \hat{q}_{EM,0}(b') \wedge \hat{q}_{EM,0}(b') < k$, then $\hat{q}_{EM,1}(b') = \hat{q}_{EM,0}(b') + 1$.
  - If $\widetilde{q}_0(b') \leq \hat{q}_{May,0}(b)$, then: $\widetilde{q}_0(b') \leq \widetilde{q}_0(b)$.
    Moreover, $b \neq b' \Rightarrow \widetilde{q}_0(b') < \widetilde{q}_0(b)$, and thus: $\widetilde{q}_2(b') \geq \hat{q}_{EM,0}(b') + 1 = \hat{q}_{EM,1}(b')$.
  - Otherwise, $\widetilde{q}_0(b') > \hat{q}_{May,0}(b)$ and thus: $\widetilde{q}_0(b') > \hat{q}_{EM,0}(b')$.
    * if $\widetilde{q}_0(b') \geq \widetilde{q}_0(b)$, then:
      $\widetilde{q}_2(b') = \widetilde{q}_0(b') \geq \hat{q}_{EM,0}(b') + 1 = \hat{q}_{EM,1}(b')$
    * if $\widetilde{q}_0(b') < \widetilde{q}_0(b) \wedge \widetilde{q}_0(b') < k$, then:
      $\widetilde{q}_2(b') = \widetilde{q}_0(b') + 1 > \hat{q}_{EM,0}(b') + 1 = \hat{q}_{EM,1}(b')$
    * if $\widetilde{q}_0(b') < \widetilde{q}_0(b) \wedge \widetilde{q}_0(b') = k$, then:
      $\widetilde{q}_2(b') = k \geq \hat{q}_{EM,1}(b')$
- If $b \neq b' \wedge \hat{q}_{May,0}(b) \geq \hat{q}_{EM,0}(b') \wedge \hat{q}_{EM,0}(b') = k$, then $\hat{q}_{EM,1}(b') = k$.
  Thus: $b \neq b' \Rightarrow \widetilde{q}_2(b') \geq \widetilde{q}_0(b') \geq \hat{q}_{EM,0}(b') = k \geq \hat{q}_{EM,1}(b')$

In every case, we have: $\widetilde{q}_2(b') \geq \hat{q}_{EM,1}(b')$.

Thus, $\widetilde{Q}_2 \in \mathcal{Q}_1$, proving that $\mathcal{Q}_2 \subseteq \mathcal{Q}_1$ $\square$

Similarly to the $update$ and $update_{EH}$, one can show the consistency of the join operator $\sqcup_{EH}$ relatively to its concrete counterpart.

**Lemma 86** (EH Join Consistency). *The join operator* $\sqcup_{EH}$ *is correct:*

$$\forall((\hat{q}_{EH,1}, \hat{q}_{Must,1}), (\hat{q}_{EH,2}, \hat{q}_{Must,2})) \in D_{EH}^2, \forall Q_1 \in \gamma_{EH}(\hat{q}_{EH,1}, \hat{q}_{Must,1}), \forall Q_2 \in \gamma_{EH}(\hat{q}_{EH,2}, \hat{q}_{Must,2}) : \\ Q_1 \cup Q_2 \in \gamma_{EH}((\hat{q}_{EH,1}, \hat{q}_{Must,1}) \sqcup_{EH} (\hat{q}_{EH,2}, \hat{q}_{Must,2})). \quad (4.5)$$

*Proof.* Let $((\hat{q}_{EH,1}, \hat{q}_{Must,1}), (\hat{q}_{EH,2}, \hat{q}_{Must,2})) \in D_{EH}^2, Q_1 \in \gamma_{EH}(\hat{q}_{EH,1}, \hat{q}_{Must,1}), Q_2 \in \gamma_{EH}(\hat{q}_{EH,2}, \hat{q}_{Must,2})$. We use the additional notation:

- $Q_3 = Q_1 \cup Q_2$
- $(\hat{q}_{EH,3}, \hat{q}_{Must,3}) = (\hat{q}_{EH,1}, \hat{q}_{Must,1}) \sqcup_{EH} (\hat{q}_{EH,2}, \hat{q}_{Must,2})$

We want to prove that: $Q_3 \in \gamma_{EH}(\hat{q}_{EH,3}, \hat{q}_{Must,3})$.

Let $b \in Blocks$, $\min_{q \in Q_3} q(b) \leq \min_{q \in Q_1} q(b) \leq \hat{q}_{EH,1}(b)$. Similarly, $\min_{q \in Q_3} q(b) \leq \hat{q}_{EH,2}(b)$. Thus, $\min_{q \in Q_3} q(b) \leq \hat{q}_{EH,3}(b)$.

Then, using the consistency of the must join, we have: $Q_3 \in \gamma_{EH}(\hat{q}_{EH,3}, \hat{q}_{Must,3})$ □

As for the $update$, the consistency proof for the EM join is a very similar to the EH join operator.

**Lemma 87** (EM Join Consistency). *The join operator* $\sqcup_{EM}$ *is correct:*

$$\forall((\hat{q}_{EM,1}, \hat{q}_{May,1}), (\hat{q}_{EM,2}, \hat{q}_{May,2})) \in D_{EM}^2, \forall Q_1 \in \gamma_{EM}(\hat{q}_{EM,1}, \hat{q}_{May,1}), Q_2 \in \gamma_{EM}(\hat{q}_{EM,2}, \hat{q}_{May,2}) : \\ Q_1 \cup Q_2 \in \gamma_{EM}((\hat{q}_{EM,1}, \hat{q}_{May,1}) \sqcup_{EM} (\hat{q}_{EM,2}, \hat{q}_{May,2})). \quad (4.6)$$

*Proof.* Let $((\hat{q}_{EM,1}, \hat{q}_{May,1}), (\hat{q}_{EM,2}, \hat{q}_{May,2})) \in D_{EM}^2, Q_1 \in \gamma_{EM}(\hat{q}_{EM,1}, \hat{q}_{May,1}), Q_2 \in \gamma_{EM}(\hat{q}_{EM,2}, \hat{q}_{May,2})$. We use the additional notation:

- $Q_3 = Q_1 \cup Q_2$
- $(\hat{q}_{EM,3}, \hat{q}_{May,3}) = (\hat{q}_{EM,1}, \hat{q}_{May,1}) \sqcup_{EM} (\hat{q}_{EM,2}, \hat{q}_{May,2})$

We want to prove that: $Q_3 \in \gamma_{EM}(\hat{q}_{EM,3}, \hat{q}_{May,3})$.

Let $b \in Blocks$, $\max_{q \in Q_3} q(b) \geq \max_{q \in Q_1} q(b) \geq \hat{q}_{EM,1}(b)$. Similarly, $\max_{q \in Q_3} q(b) \geq \hat{q}_{EM,2}(b)$. Thus, $\max_{q \in Q_3} q(b) \geq \hat{q}_{EM,3}(b)$.

Then, using the consistency of the may join, we have: $Q_3 \in \gamma_{EM}(\hat{q}_{EM,3}, \hat{q}_{May,3})$ □

Using the previous results on the local consistency of all the operators involved in our analyses, one can show the consistency as whole. This will guarantee that if the initial abstract value correctly abstracts the real initial cache states, then the analysis is correct at any point in the program.

**Theorem 88** (Analysis Soundness). *The collecting semantics is a member of both the EH and EM abstract semantics:*

$$\forall v \in V : F(v) \in \gamma_{EH}(F_{EH}^{\#}(v)) \cap \gamma_{EM}(F_{EM}^{\#}(v))$$

*Proof.* Both the collecting semantics and the abstract EH semantics are defined as least solutions to sets of equations, i.e., least fixed points of functions corresponding to these equations. The two domains are both finite for a given program, as the number of memory blocks is finite. Thus, both domains have finite ascending chains, and so the least fixed points can be obtained in a finite number of Kleene iterations.

Let $F_i : V \to D_{LRU}$ and $F^{\#}_{EH,i} : V \to D_{EH}$ denote the values reached in the $i^{th}$ Kleene iteration:

$$\forall v' \in V : F_{i+1}(v') = F_0(v') \cup \bigcup_{(v,v')\in E} update(F_i(v), blocks(v)), \tag{4.8}$$

$$\forall v' \in V : F^{\#}_{EH,i+1}(v') = F^{\#}_{EH,0}(v') \sqcup_{EH} \bigsqcup_{(v,v')\in E} update_{EH}(F^{\#}_{EH,i}(v), blocks(v)). \tag{4.9}$$

We prove by induction that for all $i \in \mathbb{N}$, we have

$$\forall v' \in V : F_i(v') \in \gamma_{EH}(F^{\#}_{EH,i}(v')).$$

This then implies the theorem, as due to finite ascending chains, there is a $j \in \mathbb{N}$, such that the least solutions $F$ and $F^{\#}_{EH}$ are $F_j$ and $F^{\#}_{EH,j}$.

- Induction base ($i = 0$):

  This follows immediately from to the assumption that $F_0(v) \in \gamma_{EH}(F^{\#}_{EH,0}(v))$.

- Induction step ($i \to i + 1$):

  Let $v' \in V$ be arbitrary. By induction hypothesis, we have $F_i(v) \in \gamma_{EH}(F^{\#}_{EH,i}(v))$ for all $v \in V$ s.t. $(v, v') \in E$. By Lemma 84 (local consistency) this implies $update(F_i(v), blocks(v)) \in \gamma_{EH}(update_{EH}(F^{\#}_{EH,i}(v), blocks(v)))$ for all $v \in V$ s.t. $(v, v') \in E$. Applying Lemma 86 (join consistency) this in turn implies:
  $\bigcup_{(v,v')\in E} update(F_i(v), blocks(v)) \in \gamma_{EH}(\bigsqcup_{(v,v')\in E)} update_{EH}(F^{\#}_{EH,i}(v), blocks(v))$.
  Applying Lemma 86 again, as by assumption $F_0(v') \in \gamma_{EH}(F^{\#}_{EH,0}(v'))$, yields:

$$\begin{aligned} F_0(v') \cup \bigcup_{(v,v')\in E} & update(F_i(v), blocks(v)) \\ & \in \gamma_{EH}(F^{\#}_{EH,0}(v') \sqcup_{EH} \bigsqcup_{(v,v')\in E} update_{EH}(F^{\#}_{EH,i}(v), blocks(v))), \end{aligned} \tag{4.10}$$

  which, by (4.8) and (4.9), is equivalent to $F_{i+1}(v') \in \gamma_{EH}(F^{\#}_{EH,i+1}(v'))$.

□

### 4.1.4 Experimental Evaluation

#### Experimental settings

To illustrate the behavior of the *Definitely Unknown* analysis, we implemented it in a tool called OTAWA [BCRS10]. OTAWA is framework for WCET estimation, and provide several useful tools to integrate our cache analyses. In our case, we benefit from CFG reconstruction and decoration

with memory accesses. We also reuse the abstract interpretation engine that comes with the May/Must analysis.

We analyze 50 benchmarks from the TACLeBench [FAH+16] benchmark collection, which is the successor of the Mälardalen collection and is widely used in the WCET community. These benchmarks are compiled into an ARM executable, and have sizes that range from 18 cache blocks for the smallest to more than 5000 blocks for the biggest. A majority of benchmarks (33 over 50) have a size between 100 and 1000 memory blocks. Figure 4.7 shows the size of the benchmarks using a logarithmic scale. The cache configuration used in our experiment is 8 ways cache of 32 sets with blocks of 16 bytes for a total of 4KB.

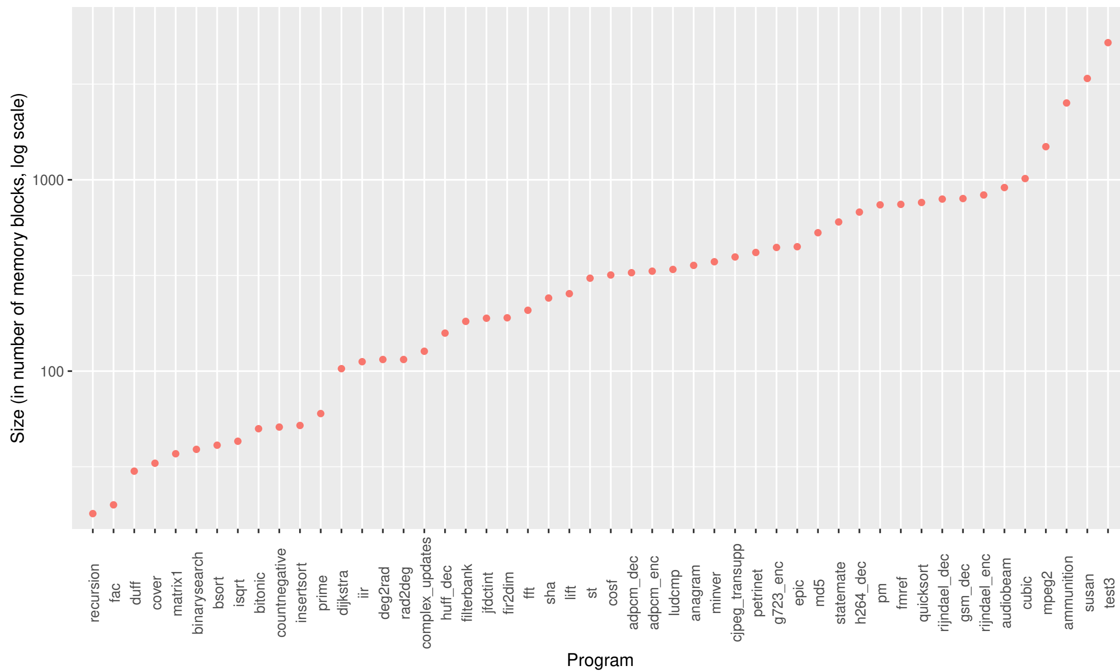



Figure 4.7 – Size of the benchmarks in number of cache blocks

Using these settings we evaluate the performance of the *Definitely Unknown* analysis by measuring two values:

- First, we look at the number of accesses that are classified as *Definitely Unknown* and compare it to the number of accesses that were left unclassified by the may and must analyses. By doing so, we estimate the reduction of uncertainty due to the *Definitely Unknown* analysis.

- Second, we measure the execution time of analyses, to evaluate the efficiency of the *Definitely Unknown* analysis in comparison of the may and must analyses.

Figure 4.8 shows for each benchmark how many accesses where classified by the *Definitely Unknown* analysis among the accesses left unclassified by the may and must analyses. The total height of a bar is the number block that the may and must analyses were not able to classify (NC stands for “not classified”). These bars are split in two parts: i) a red part which shows the number of blocks that the *Definitely Unknown* analysis was able to classified ii) a blue part (at the bottom of the red part) that shows the potential accesses that are still candidates for a refinement after the *Definitely Unknown* analysis has been performed.

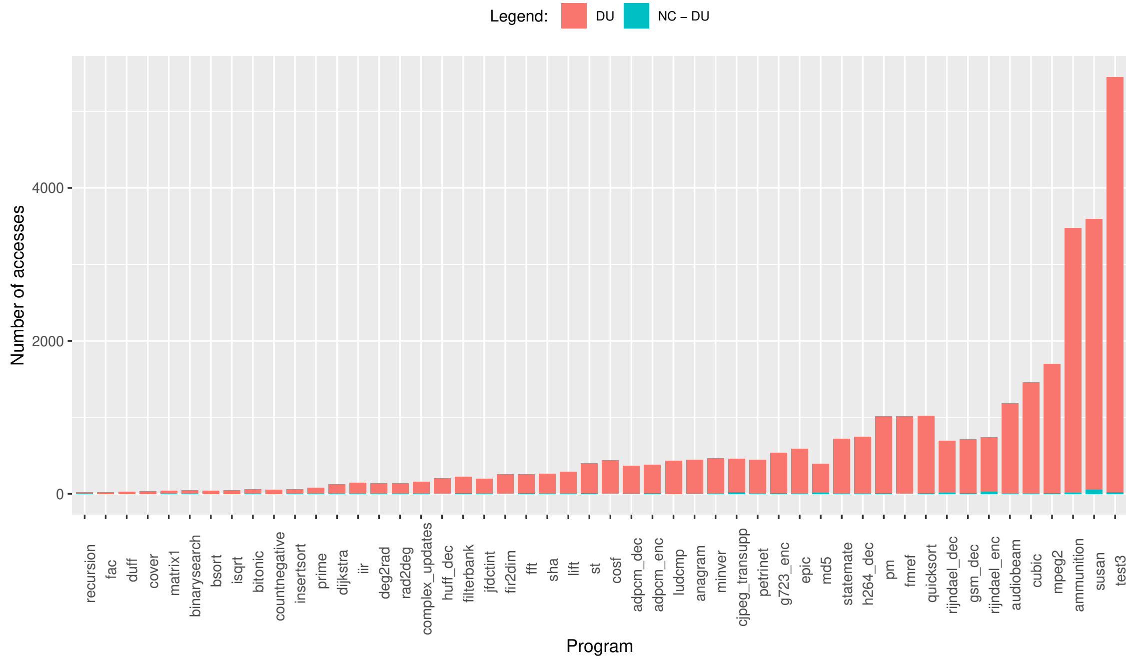



Figure 4.8 – Number of blocks refined from *Unknown* to *Definitely Unknown*

This experiment shows that the vast majority (more than 98.4% in geometric average) of the accesses left unclassified are in fact accesses that can both lead to a hit and a miss. By performing this analysis we thus drastically reduce the number of accesses to refine. For instance, the maximum number of accesses reamining unclassified after this analysis is 56 for the benchmark called susan (the total number of accesses is 3595). This also shows that the May and Must analyses are very efficient in practice. Among all the accesses they left unclassified, less than 2% could be real *Always-Hit* or *Always-Miss* in the worst case.

Figure 4.9 also shows the classification of accesses, but the first iteration of loops is unrolled. In all this thesis, loop unrolling refers to a transformation of the CFG manipulated by the analyzer, as explained in Section 2.3.4, and not to a compile time optimization. Indeed, the typical behavior of small loops is to produce cache misses during the first iteration, and cache hits afterward. One can thus argue that these definitely unknown blocks could have been easily refined by unrolling the loops or performing a persistence analysis. However, our experiment validates our approach, with more than 98.4% of the accesses classified by the *Definitely Unknown* analysis. Note that by unrolling the first iteration of loops the number of accesses in some benchmarks highly increases. This phenomenon is due to nested loops. Indeed, by unrolling the first iteration of a loop, we duplicate the loop body. Accesses that are nested in $n$ loops are duplicated $2^n$ times. However, the number of accessse classified as *Unknown* or *Definitely Unknown* remains approximately the same. For instance, in the case of the susan benchmark, the number of accesses goes from 4361 to 13950 due to several nested loop levels. However the number of *Unknown* accesses goes from 3595 to 3589, and the number of *Definitely Unknown* accesses goes from 3539 to 3541.

The second set of experiments aims at measuring the cost of the *Definitely Unknown* analysis relatively to the May and Must analyses. Figure 4.10 shows the cost of both analyses in logarithmic scale. In geometric average, the definitely unknown analysis is only 2.72 times slower than the May and Must analyses.

Figure 4.11 shows the same comparison of costs when unrolling the first iteration of loops. This time, our analysis complete in less than 3.25 times the cost of the May and Must analyses.

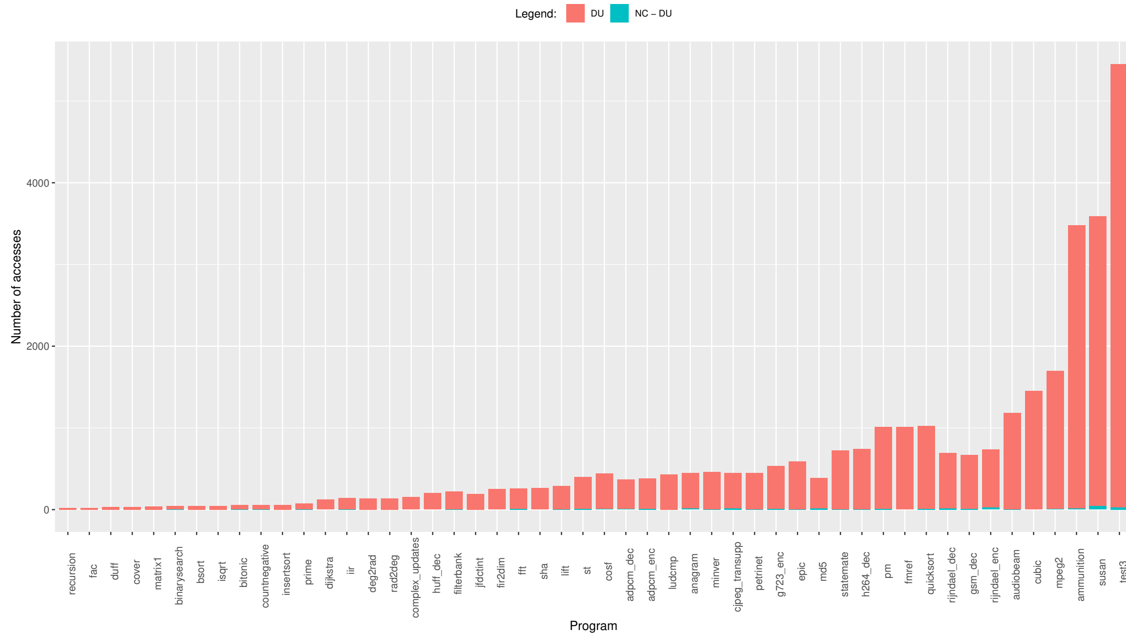



Figure 4.9 – Number of blocks refined from *Unknown* to *Definitely Unknown* when unrolling loops

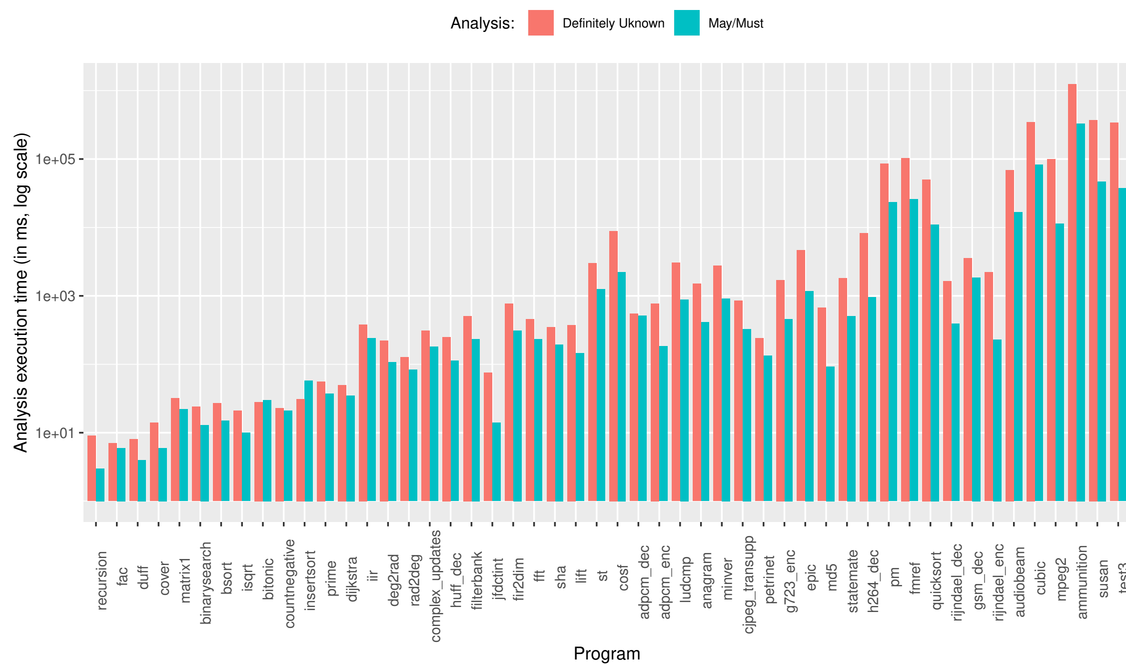



Figure 4.10 – Comparison of the May/Must and Definitely Unknown analysis

However, note that the execution time of both analyses (May/Must and Definitely Unknown) highly increases for some benchmarks with deep nested loops.

## Conclusion

We have demonstrated that the set of accesses classified as *Unknown* by the usual may and must analyses can be considerably reduced by performing the *Exist-Hit* and *Exist-Miss* analyses. In particular, the accesses classified as *Definitely Unknown* by these analyses are optimally classified at a

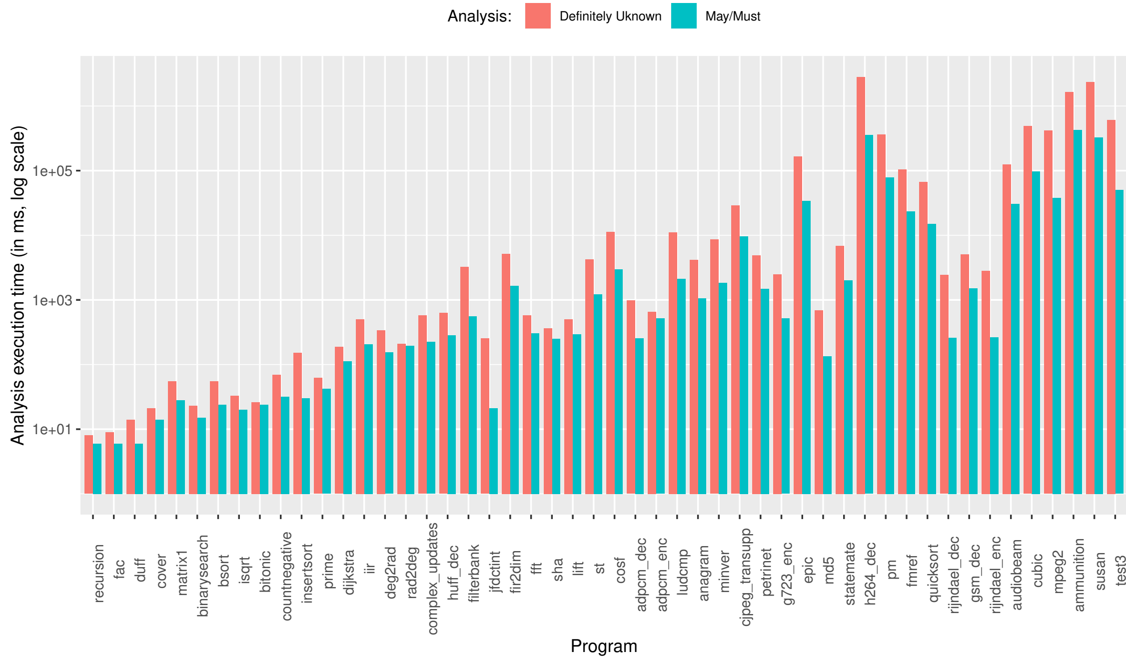



Figure 4.11 – Comparison of the May/Must and Definitely Unknown analysis when unrolling loops

reasonable cost, avoiding more costly analyses. Moreover, the accesses not classified as *Definitely Unknown* usually belong to one of the *Exist-Hit* or *Exist-Miss* category, and this information can be used by further analysis. In the remaining of this chapter, we propose different approaches to classify the remaining *Unknown* accesses.

## 4.2 Exact Analysis of LRU Cache by Model Checking

One of the question raised by the complexity results proved in Section 3 is the possibility of designing an optimally precise cache analysis efficient in practice. This section proposes an approach that rely on a model-checker and an exact abstraction to achieve this goal.

Because the model-checking phase is based on an exact abstraction of the cache replacement policy, our method, overall, is *optimally precise*: it answers precisely whether a given access is always a hit, always a miss, or a hit in some executions and a miss in others[3] (see “Result after refinement” in Fig. 4.1). This precision improvement in access classifications can be beneficial for tools built on top of the cache analysis, as demonstrated in Section 5.

The model-checking phase would be sufficient to resolve all accesses, but our experiments show this does not scale; it is necessary to combine it with the abstract-interpretation phase for tractability, thereby reducing (a) the number of model-checker calls, and (b) the size of each model-checking problem.

[3]This completeness is relative to an execution model where all control paths are feasible, disregarding the functional semantics of the edges.

### 4.2.1 Block Focusing

Using the *Definitely Unknown* analysis described in the previous section, one can reduce the set of accesses that can be refined into *Always-Hit* or *Always-Miss*. We shall now see how to classify these remaining blocks using model checking. Not only is the model-checking phase *sound*, i.e. its classifications are correct, it is also *complete* relative to our program model, i.e. there remain no unclassified accesses: each access is classified as "always hit", "always miss" or "definitely unknown". Remember that our analysis is based on the assumption that each path in the CFG is semantically feasible.

In order to classify the remaining unclassified accesses, we feed the model checker a finite-state machine modeling the cache behavior of the program, composed of i) a model of the program, yielding the possible sequences of memory accesses ii) a model of the cache. In this section, we introduce a new cache model, focusing on the state of a particular memory block to be classified.

It would be possible to directly encode the control-flow graph of the program, adorned with memory accesses, as one big finite-state system. A first step is obviously to slice that system per cache set to make it smaller. Here we take this approach further by defining a model sound and complete with respect to a given memory block $a$: parts of the model that have no impact on the caching status of $a$ are discarded, which greatly reduces the model's size. For each unclassified access, the analysis constructs a model focused on the memory block accessed, and queries the model checker. Both the simplified program model and the focused cache model are derived automatically, and do not require any manual interaction.

The *focused cache model* is based on the following simple property of LRU: a memory block is cached if and only if its age is less than the associativity $k$, or in other words, if there are less than $k$ younger blocks. In the following, without loss of generality, let $a \in Blocks$ be the memory block we want to focus the cache model on. If we are only interested in whether $a$ is cached or not, it suffices to track the set of blocks younger than $a$. Without any loss in precision concerning $a$, we can abstract from the relative ages of the blocks younger than $a$ and of those older than $a$.

Thus, the domain of the focused cache model is $D_{\odot} = \mathcal{P}(Blocks) \cup \{\dashv\}$. Here, $\dashv$ is used to represent those cache states in which $a$ is not cached. If $a$ is cached, the analysis tracks the set of blocks younger than $a$. We can relate the focused cache model to the concrete cache model defined in Section 4.1.1 using an abstraction function mapping concrete cache states to focused ones:

$$\begin{aligned} \alpha_{\odot} &: D_{LRU} \to D_{\odot} \\ q &\mapsto \begin{cases} \dashv & \text{if } q(a) = k \\ \{b \in Blocks, q(b) < q(a)\} & \text{if } q(a) < k \end{cases} \end{aligned} \tag{4.11}$$

The focused cache update $update_{\odot}$ models a memory access as follows:

$$\begin{aligned} update_{\odot} &: D_{\odot} \times Blocks \to D_{\odot} \\ (\widehat{Q}, b) &\mapsto \begin{cases} \emptyset & \text{if } b = a \\ \dashv & \text{if } b \neq a \wedge \widehat{Q} = \dashv \\ \widehat{Q} \cup \{b\} & \text{if } b \neq a \wedge \widehat{Q} \neq \dashv \wedge |\widehat{Q} \cup \{b\}| < k \\ \dashv & \text{if } b \neq a \wedge \widehat{Q} \neq \dashv \wedge |\widehat{Q} \cup \{b\}| = k \end{cases} \end{aligned} \tag{4.12}$$

Let us briefly explain the four cases above. If $b = a$ (case 1), $a$ becomes the most-recently-used block and thus no other blocks are younger. If $a$ is not in the cache and it is not accessed (case 2), then $a$ remains outside of the cache. If another block is accessed, it is added to $a$'s younger set (case 3) unless the access causes $a$'s eviction, because it is the $k^{th}$ distinct younger block (case 4).

**Example.** Figure 4.12 depicts a sequence of memory accesses and the resulting concrete and focused cache states (with a focus on block $a$) starting from an empty cache of associativity 2. We represent concrete cache states by showing the two blocks of age 0 and 1. The example illustrates that many concrete cache states may collapse to the same focused one. At the same time, the focused cache model does not lose any information about the caching status of the focused block, which is captured by the following lemma and theorem (see Section 4.2.2 for the proofs).

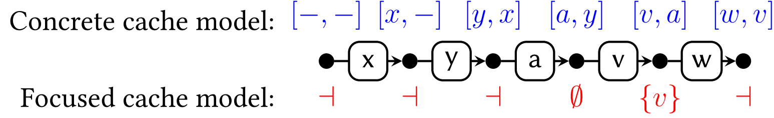



Figure 4.12 – Example: concrete vs. focused cache model.

**Lemma 89** (Local Soundness and Completeness)**.** *The focused cache update abstracts the concrete cache update exactly:*

$$\forall q \in D_{LRU}, \forall b \in Blocks : \alpha_{\odot}(update(q, b)) = update_{\odot}(\alpha_{\odot}(q), b) \tag{4.13}$$

The *focused collecting semantics* is defined analogously to the *collecting semantics* as the least solution to the following set of equations, where $F_{\odot}(v)$ denotes the set of reachable focused cache configurations at each program location, and $F_{\odot,0}(v) = \alpha_{\odot}(F_0(v))$ for all $v \in V$:

$$\forall v' \in V : F_{\odot}(v') = F_{\odot,0}(v') \cup \bigcup_{(v,v')\in E} update_{\odot}(F_{\odot}(v), blocks(v)), \tag{4.14}$$

where $update_{\odot}$ denotes the focused cache update function lifted to sets of focused cache states, i.e., $update_{\odot}(Q, b) = \{update_{\odot}(q, b), q \in Q\}$, and $\alpha_{\odot}$ is lifted to sets of states, i.e., $\alpha_{\odot}(Q) = \{\alpha_{\odot}(q), q \in Q\}$.

**Theorem 90** (Analysis Soundness and Completeness)**.** *The focused collecting semantics is exactly the abstraction of the collecting semantics:*

$$\forall v \in V, \alpha_{\odot}(F(v)) = F_{\odot}(v). \tag{4.15}$$

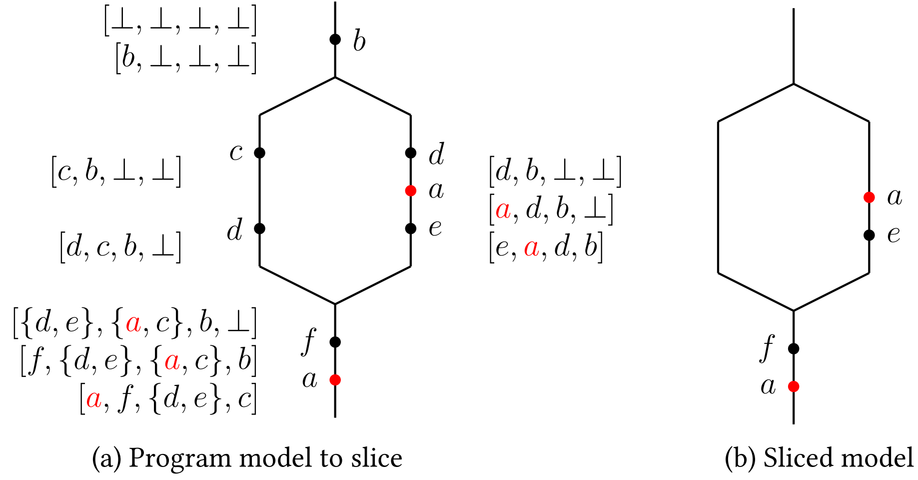
*Proof.* From Lemma 89 it immediately follows that the lifted focused update $update_{\odot}$ exactly corresponds to the lifted concrete cache update $update$.

Since the concrete domain is finite, the least fixed point of the system of equations defining the concrete collecting semantics (Def. 4.1) is reached after a bounded number of Kleene iterations. One then just applies the consistency lemmas in an induction proof. □

Thus we can employ the focused cache model in place of the concrete cache model without any loss in precision to classify accesses to the focused block as "always hit", "always miss", or "definitely unknown".

For the program model, we simplify the CCG without affecting the correctness nor the precision of the analysis: i) If we know, from may analysis, that in a given program block $a$ is never in the cache, then this block cannot affect $a$'s eviction: thus we simplify the program model by not including this block. ii) When we encode the set of blocks younger than $a$ as a bit vector, we do not include blocks that the may analysis proved not to be in the cache at that location: these bits would anyway always be 0.

An example of the first simplification is shown on Figure 4.13. The Figure 4.13a shows the complete CCG, on which the May analysis has been performed. From the result, on can notice that the beginning of the graph and the left branch are not relevant when classifying $a$. They can thus be removed from the program model.



(a) Program model to slice

(b) Sliced model

Figure 4.13 – Slicing a program model according to l-block $a$

### 4.2.2 Proof of Block focusing correctness

**Lemma 89** (Local Soundness and Completeness)**.** *The focused cache update abstracts the concrete cache update exactly:*

$$\forall q \in D_{LRU}, \forall b \in Blocks : \alpha_{\odot}(update(q, b)) = update_{\odot}(\alpha_{\odot}(q), b) \tag{4.13}$$

*Proof.* Let $q = [b_0, ..., b_{k-1}] \in D_{LRU}$ a reachable cache state and $b \in Blocks$ a memory block.

We prove consistency by inspection of different possible cases. First half of the proof deals with cache states that do not contain the interesting block $a$ (i.e. $\forall i \in \{0, ..., k-1\}, b_i \neq a$). Second half deals with cache states that contain it (i.e. $\exists i \in \{0, ..., k-1\}, b_i = a$). In both part, we treat the cases where the block accessed $b$ is $a$ or not. Moreover, to treat $a$ eviction, the second half of the proof adds sub-cases for distinction of states containing $a$ "at the end" of the cache (near eviction).

- if $\forall i \in \{0, ..., k-1\}, b_i \neq a$ (i.e. $a$ is not in $q$), then we can distinguish two cases:
  - if $b = a$ then:

$$\begin{aligned}
\alpha_{\odot}(update(q, b)) &= \alpha_{\odot}(update([\overset{\neq a}{b_0}, \overset{\neq a}{b_1}, ..., \overset{\neq a}{b_{k-1}}], a)) \\
&= \alpha_{\odot}([a, b_0, ..., b_{k-2}]) \\
&= \{\} \\
update_{\odot}(\alpha_{\odot}(q), b) &= update_{\odot}(\alpha_{\odot}([\overset{\neq a}{b_0}, \overset{\neq a}{b_1}, ..., \overset{\neq a}{b_{k-1}}]), a) \\
&= update_{\odot}(\dashv, a) \\
&= \{\}
\end{aligned}$$

So consistency holds.

– if $b \neq a$ then:

$$\begin{aligned}
\alpha_{\odot}(\mathit{update}(q, b)) &= \alpha_{\odot}(\mathit{update}([\overset{\neq a}{b_0}, \overset{\neq a}{b_1}, ..., \overset{\neq a}{b_{k-1}}], \overset{\neq a}{b})) \\
&= \alpha_{\odot}([\overset{\neq a}{b'_0}, \overset{\neq a}{b'_1}, ..., \overset{\neq a}{b'_{k-1}}]) \text{ where } \forall i \in \{0, ..., k-1\}, b'_i \in \{b_0, ..., b_{k-1}, b\} \\
&= \dashv \\
\mathit{update}_{\odot}(\alpha_{\odot}(q), b) &= \mathit{update}_{\odot}(\alpha_{\odot}([\overset{\neq a}{b_0}, \overset{\neq a}{b_1}, ..., \overset{\neq a}{b_{k-1}}]), b) \\
&= \mathit{update}_{\odot}(\dashv, \overset{\neq a}{b}) \\
&= \dashv
\end{aligned}$$

So consistency holds.

- if $\exists i \in \{0, ..., k-1\}$ such that $b_i = a$ (i.e. $a$ is in the cache), we also distinguish between the cases $b = a$ and $b \neq a$:

– if $b = a$ then:

$$\begin{aligned}
\alpha_{\odot}(\mathit{update}(q, b)) &= \alpha_{\odot}(\mathit{update}([\overset{\neq a}{b_0}, ..., \overset{\neq a}{b_{i-1}}, a, b_{i+1}, ..., b_{k-1}], a)) \\
&= \alpha_{\odot}([a, b_0, ..., b_{i-1}, b_{i+1}, ..., b_{k-1}]) \\
&= \{\} \\
\mathit{update}_{\odot}(\alpha_{\odot}(q), b) &= \mathit{update}_{\odot}(\alpha_{\odot}([\overset{\neq a}{b_0}, ..., \overset{\neq a}{b_{i-1}}, a, b_{i+1}, ..., b_{k-1}]), a) \\
&= \mathit{update}_{\odot}(\{b_0, ..., b_{i-1}\}, a) \\
&= \{\}
\end{aligned}$$

So consistency holds.

– if $b \neq a$, there are different cases depending on whether $b$ is in the cache before or after $a$ and depending on whether $a$ is the least recently used block:

* if there exists $j < i$ such that $b_j = b$ (i.e. $b$ is in the cache and is younger than $a$):

$$\begin{aligned}
&\alpha_{\odot}(\mathit{update}(q, b)) \\
&= \alpha_{\odot}(\mathit{update}([b_0, ..., b_{j-1}, b, b_{j+1}, ..., b_{i-1}, a, b_{i+1}, ..., b_{k-1}], b)) \\
&= \alpha_{\odot}([b, b_0, ..., b_{j-1}, b_{j+1}, ..., b_{i-1}, a, b_{i+1}, ..., b_{k-1}]) \\
&= \{b, b_0, ..., b_{j-1}, b_{j+1}, ..., b_{i-1}\} \\
&= \{b_0, ..., b_{i-1}\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{update}_{\odot}(\alpha_{\odot}(q), b) \\
&= \mathit{update}_{\odot}(\alpha_{\odot}([b_0, ..., b_{j-1}, b, b_{j+1}, ..., b_{i-1}, a, b_{i+1}, ..., b_{k-1}]), b) \\
&= \mathit{update}_{\odot}(\{b_0, ..., b_{j-1}, b, b_{j+1}, ..., b_{i-1}\}), b) \\
&= \mathit{update}_{\odot}(\{b_0, ..., b_{i-1}\}, b) \\
&= \{b_0, ..., b_{i-1}\}
\end{aligned}$$

So consistency holds.

* if there exists $j > i$ such that $b_j = b$ (i.e. $b$ is in the cache and is older than $a$):

$$\begin{aligned}
&\alpha_\odot(update(q, b)) \\
&= \alpha_\odot(update([b_0, ..., b_{i-1}, a, b_{i+1}, ..., b_{j-1}, b, b_{j+1}, ..., b_{k-1}], b)) \\
&= \alpha_\odot([b, b_0, ..., b_{i-1}, a, b_{i+1}, ..., b_{j-1}, b_{j+1}, ..., b_{k-1}], b) \\
&= \{b, b_0, ..., b_{i-1}\}
\end{aligned}$$

$$\begin{aligned}
&update_\odot(\alpha_\odot(q), b) \\
&= update_\odot(\alpha_\odot([b_0, ..., b_{i-1}, a, b_{i+1}, ..., b_{j-1}, b, b_{j+1}, ..., b_{k-1}]), b) \\
&= update_\odot(\{b_0, ..., b_{i-1}\}, b) \\
&= \{b, b_0, ...b_{i-1}\}
\end{aligned}$$

  So consistency holds.

* if $\forall j, b_j \neq b$ and $i \neq k-1$ (i.e. $b$ is not in the cache and $a$ is not the least recently used block):

$$\begin{aligned}
\alpha_\odot(update(q, b)) &= \alpha_\odot(update([b_0, ..., b_{i-1}, a, b_{i+1}, ..., b_{k-1}], b)) \\
&= \alpha_\odot([b, b_0, ..., b_{i-1}, a, b_{i+1}, ..., b_{k-2}]) \\
&= \{b, b_0, ..., b_{i-1}\} \\
update_\odot(\alpha_\odot(q), b) &= update_\odot(\alpha_\odot([b_0, ..., b_{i-1}, a, b_{i+1}, ..., b_{k-1}]), b) \\
&= update_\odot(\{b_0, ..., b_{i-1}, a\}, b) \\
&= \{b, b_0, ..., b_{i-1}\}
\end{aligned}$$

  So consistency holds.

* if $\forall j, b_j \neq b$ and $i = k-1$ (i.e. $b$ is not in the cache and $a$ is the least recently used block):

$$\begin{aligned}
\alpha_\odot(update(q, b)) &= \alpha_\odot(update([b_0, ..., b_{k-2}, a], b)) \\
&= \alpha_\odot([b, b_0, ..., b_{k-2}]) \\
&= \dashv \\
update_\odot(\alpha_\odot(q), b) &= update_\odot(\alpha_\odot([b_0, ..., b_{k-2}, a]), b) \\
&= update_\odot(\{b_0, ..., b_{k-2}\}, b) \\
&= \dashv
\end{aligned}$$

  So consistency holds.

In all cases, consistency holds. □

## 4.3 Exact Analysis of LRU Cache by Abstract Interpretation

In this section, we develop an analysis based on abstract interpretation that comes close to the efficiency of the classical approach [AFMW96] while achieving exact classification of all memory accesses as the model-checking approach described in Section 4.2. In other terms, we introduce

an exact and scalable analysis by carefully refining the abstraction and using suitable algorithms and data structures.

Our main contribution in this Chapter is the introduction of a new exact abstraction for LRU caches that is based on a partial order of cache states. To classify cache misses (cache hits), it is sufficient to only keep minimal (maximal) elements w.r.t. this partial order. As a consequence, the abstraction may be exponentially more succinct than the model-checking approach.

We improve the focused semantics presented in Section 4.2 by removing subsumed elements with upward and downward closures. This form of convergence acceleration preserves the precision of the final classification.

We discuss a suitable data structure for this abstraction based on zero-suppressed binary decision diagrams (ZDDs), and an implementation on top of OTAWA [BCRS10] and CUDD [Som01]. Our experimental evaluation shows an analysis speedup of up to 950 compared with the prior exact approach 4.2. The geometric mean of the speedup across all studied benchmarks is at least 9[4]. On the other hand, compared with the imprecise age-based analysis [AFMW96], we observe an average slowdown across all benchmarks of only 4.12.

### Collecting Semantics vs Focused Semantics vs Antichain

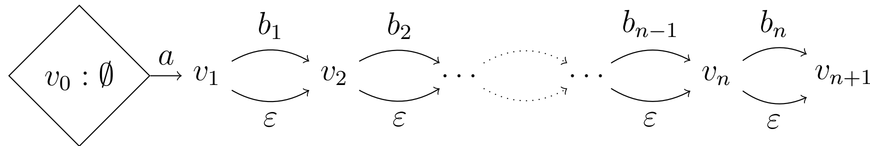



Figure 4.14 – Example where collecting and focused semantics are unnecessarily detailed and inefficient. $\varepsilon$ means "no access".

Consider the CCG of 4.14, with an associativity $k > n$, and an empty initial cache. At the last control location $v_{n+1}$, the possible cache states are all subsequences of $b_n, \ldots, b_1$ followed by $a$ and possibly empty lines, e.g. $(b_5, b_3, b_1, a, \bot, \bot)$. There are therefore $2^n$ reachable cache states at $v_{n+1}$, all of which appear in the collecting semantics of the program composed with the cache.

The "block-focused" abstraction, which was also applied in the model checking approach 4.2 when encoding cache problems into model-checking reachability problems, records only the set of blocks younger than the block of interest. Here, if our block of interest is $a$, this abstraction thus yields at $v_{n+1}$ the set of subsets of $\{b_1, \ldots, b_n\}$. Of course, symbolic set representation techniques may have a compact representation for such a set, but the main issue is that this set keeps too much information.

If our goal is to prove the existence of a "hit" on an access to memory block $a$ further down the execution, then it is sufficient to keep, in this set, $\emptyset$, corresponding to a path composed of the access to $a$ followed by a sequence of no accesses $\varepsilon$. More generally, it is sufficient to keep only the minimal elements (with respect to the inclusion ordering) from this set, which can be exponentially more succinct, as in this example. Similarly, if our goal is to prove the existence of a "miss" on $a$ further down the execution, then it is sufficient to keep, in this set, $\{b_1, \ldots, b_n\}$, corresponding to a path composed of the access to $a$ followed by a sequence of accesses to $b_1$, ..., $b_n$. More generally, it is sufficient to keep only the maximal elements from this set.

[4] On a number of benchmarks the prior exact approach timed out at 12 hours. For these benchmarks, we conservatively assume an analysis time of 12 hours, and may thus underestimate the actual speedup, had the analysis been run to completion.

This is the main difference between our analysis and the "focused" model that we fed into the model checker 4.2: the "focused" model contains unnecessary information (non-minimal elements for the Always-miss analysis, non-maximal elements for the Always-hit analysis), which increases model-checking times. We discard these in our analysis.

The following section formally defines our new *Always-Hit* and *Always-Miss* analyses. The Always-hit analysis (respectively Always-miss) computes the antichain of maximal elements (respectively minimal elements) i.e. the downward (respectively upward) closure of reachable states.

### 4.3.1 Exact Analyses as Fixed-Point Problems

#### Always-Hit Analysis

To get a better idea of what the Always-hit analysis computes let us first recall the definitions of antichain and upper set, and illustrate this analysis with an example.

**Definition 91.** *An* antichain *is a subset of an ordered set such that no two distinct elements of that subset are comparable. An* upper set *(respectively* lower set*) is a set such that if an element is in this set, then all elements larger (respectively, smaller) than it are also in the set.*

**Example 92.** *Assume that a basic block $BB_i$ may be reached with cache states*

$$F(BB_i) = \{[c, b, e, a], [b, c, d, a], [b, a, \bot, \bot]\}$$

*The possible a-focused states are: $F_{\odot}(BB_i) = \{\{b, c, e\}, \{b, c, d\}, \{b\}\}$. Since $\{b\}$ is strictly included in $\{b, c, e\}$ (and in $\{b, c, d\}$), it may not contribute to cache misses that would not also occur following $\{b, c, e\}$ (and $\{b, c, d\}$) and can be removed without affecting soundness; the antichain of the maximal elements of $F_{\odot}(BB_i)$ is $\{\{b, c, e\}, \{b, c, d\}\}$, which will be called $F_{\odot}^{\max}(BB_i)$.*

In all that follows, the ordering will be the inclusion ordering $\subset$.

Recall that for any $S \in F_{\odot}(BB_i), S \neq \dashv$ means that at entry of basic block $BB_i$, there is a reachable cache state of the form $(b_0, \dots, b_{|S|-1}, a, \dots)$ where $S = \{b_0, \dots, b_{|S|-1}\}$. An access to block $a$ at entry of $BB_i$ "always hits" if and only if it "may not miss", that is, if there is no execution trace leading to a miss at this location, i.e. $\dashv \notin F_{\odot}(BB_i)$.

**Definition 93.** *Let $\widehat{Q} \xrightarrow{b} \widehat{Q}'$ denote the transition "upon an access to block $b$, $b \neq a$, the cache may move from an $a$-focused state $\widehat{Q}$ to an $a$-focused state $\widehat{Q}'$". Recall that $\widehat{Q}$ may be $\dashv$ ($a$ is not in the cache) or a subset of cache blocks, not containing $a$, of cardinality at most $k-1$. This deterministic transition relation is defined as follows:*

- $\dashv \xrightarrow{b} \dashv$;
- $\widehat{Q} \xrightarrow{b} \widehat{Q} \cup \{b\}$ *for* $|\widehat{Q} \cup \{b\}| < k$;
- $\widehat{Q} \xrightarrow{b} \dashv$ *for* $|\widehat{Q} \cup \{b\}| = k$.

**Definition 94.** *For $\widehat{Q}, \widehat{Q}' \in D_{\odot}$, let $\widehat{Q}, \xrightarrow{\downarrow} \widehat{Q}'$ denote a downward closure step:*

- $\dashv \xrightarrow{\downarrow} y$ *for any* $y$;
- $x \xrightarrow{\downarrow} y$ *for any* $x, y \neq \dashv, y \subseteq x$.

**Example 95.** $\{b,c,e\} \xrightarrow{\downarrow} \widehat{Q}'$ *for any* $\widehat{Q}' \in \{\emptyset, \{b\}, \{c\}, \{e\}, \{b,c\}, \{b,e\}, \{c,e\}, \{b,c,e\}\}$

**Lemma 96.** *Assume there are* $\widehat{Q}_1, \widehat{Q}_2, \widehat{Q}_3$, *and* $b$ *such that* $\widehat{Q}_1 \xrightarrow{\downarrow} \widehat{Q}_2 \xrightarrow{b} \widehat{Q}_3$. *Then there exists* $\widehat{Q}'_2$ *such that* $\widehat{Q}_1 \xrightarrow{b} \widehat{Q}'_2 \xrightarrow{\downarrow} \widehat{Q}_3$.

*Proof.* As the transition relation is deterministic, $\widehat{Q}'_2$ is uniquely determined by $\widehat{Q}_1$ and $b$.

We distinguish two cases based on the value of $\widehat{Q}'_2$:

1. If $\widehat{Q}'_2 = \dashv$, then the results follows immediately, as $\dashv \xrightarrow{\downarrow} \widehat{Q}_3$ for any $\widehat{Q}_3$.
2. If $\widehat{Q}'_2 \neq \dashv$, then $\widehat{Q}'_2 = \widehat{Q}_1 \cup \{b\}$ with $|\widehat{Q}'_2| < k$. Then $\widehat{Q}_1 \neq \dashv$ and $\widehat{Q}_2 \subseteq \widehat{Q}_1$, and so $\widehat{Q}_3 = \widehat{Q}_2 \cup \{b\} \subseteq \widehat{Q}_1 \cup \{b\} = \widehat{Q}'_2$.

□

**Corollary 97.** *There exists a sequence of the form*

$$\widehat{Q}_0 \xrightarrow{b_0} \widehat{Q}'_0 \xrightarrow{\downarrow} \widehat{Q}_1 \xrightarrow{b_1} \widehat{Q}'_1 \xrightarrow{\downarrow} \dots \widehat{Q}_n \xrightarrow{b_n} \widehat{Q}'_n \xrightarrow{\downarrow} \widehat{Q}_{n+1}$$

*if and only if there exists a sequence of the form*

$$\widehat{Q}_0 \xrightarrow{b_0} \widehat{Q}''_1 \xrightarrow{b_1} \widehat{Q}''_2 \dots \xrightarrow{b_n} \widehat{Q}''_n \xrightarrow{\downarrow} \widehat{Q}_{n+1}$$

It is thus equivalent to compute the reachable states of the $a$-focused semantics (for transitions different from $a$), then apply downward closure, and to apply downward closure at every step during the computation of reachable states. In addition, it is obvious that $\dashv$ is in a set if and only if it is in its downward closure. It is thus equivalent to test for a "may miss" on the reachable states of the $a$-focused semantics and on their downward closure.

This suggests two possible (and equivalent, in a sense) simplifications to the focused semantics if our goal is to find places where an access to $a$ may be a miss:

**Closure** Replace $F_\odot(BB_i)$ by its down-closure $F^{\downarrow}_\odot(BB_i)$: $S' \in F^{\downarrow}_\odot(BB_i)$ if and only if there exists $S \in F_\odot(BB_i)$ such that $S' \subseteq S$.

**Subsumption removal** Replace $F_\odot(BB_i)$ by the antichain of its maximal elements: $S \in F^{\max}_\odot(BB_i)$ if and only if $S \in F_\odot(BB_i)$ and there is no $S' \in F_\odot(BB_i)$ such that $S \subsetneq S'$.

Note that $F^{\downarrow}_\odot(BB_i)$ is the down-closure of $F^{\max}_\odot(BB_i)$, and that $F^{\max}_\odot(BB_i)$ is the antichain of maximal elements of $F^{\downarrow}_\odot(BB_i)$; thus $F^{\max}_\odot(BB_i)$ is just an alternative representation for $F^{\downarrow}_\odot(BB_i)$. Our idea is to directly compute $F^{\max}_\odot(BB_i)$.

### Always-Miss Analysis

This subsection presents the Always-miss analysis which is the dual of the Always-hit analysis of Section 4.3.1. A control location "always misses" if and only if it "may not hit", that is, if there is no execution trace leading to a hit at this location.

**Definition 98.** *For* $\widehat{Q}, \widehat{Q}' \in D_\odot$, *let* $\widehat{Q} \xrightarrow{\uparrow} \widehat{Q}'$ *denote an upward closure step:*

- $\widehat{Q} \xrightarrow{\uparrow} \dashv$ *for any* $\widehat{Q}$;

- $\widehat{Q} \xrightarrow{\uparrow} \widehat{Q}'$ *for any* $\widehat{Q}, \widehat{Q}' \neq \dashv$, $\widehat{Q} \subseteq \widehat{Q}'$.

**Lemma 99.** *Assume there are* $\widehat{Q}_1, \widehat{Q}_2, \widehat{Q}_3$*, and* $b$ *such that* $\widehat{Q}_1 \xrightarrow{\uparrow} \widehat{Q}_2 \xrightarrow{b} \widehat{Q}_3$*. Then there exists* $\widehat{Q}'_2$ *such that* $\widehat{Q}_1 \xrightarrow{b} \widehat{Q}'_2 \xrightarrow{\uparrow} \widehat{Q}_3$*.*

*Proof.* We distinguish two cases based on the value of $\widehat{Q}_3$:

1. If $\widehat{Q}_3 = \dashv$, then the results follows immediately, as $\widehat{Q}'_2 \xrightarrow{\uparrow} \dashv$ for any $\widehat{Q}'_2$.
2. If $\widehat{Q}_3 \neq \dashv$, then $\widehat{Q}_3 = \widehat{Q}_2 \cup \{b\}$ with $|\widehat{Q}_3| < k$. Then $\widehat{Q}_2 \neq \dashv$ and $\widehat{Q}_1 \subseteq \widehat{Q}_2$, and so $\widehat{Q}'_2 = \widehat{Q}_1 \cup \{b\} \subseteq \widehat{Q}_2 \cup \{b\} = \widehat{Q}_3$.

□

**Corollary 100.** *There exists a sequence of the form*

$$\widehat{Q}_0 \xrightarrow{b_0} \widehat{Q}'_0 \xrightarrow{\uparrow} \widehat{Q}_1 \xrightarrow{b_1} \widehat{Q}'_1 \xrightarrow{\uparrow} \dots \widehat{Q}_n \xrightarrow{b_n} \widehat{Q}'_n \xrightarrow{\uparrow} \widehat{Q}_{n+1}$$

*if and only if there exists a sequence of the form*

$$\widehat{Q}_0 \xrightarrow{b_0} \widehat{Q}''_1 \xrightarrow{b_1} \widehat{Q}''_2 \dots \xrightarrow{b_n} \widehat{Q}''_n \xrightarrow{\uparrow} \widehat{Q}_{n+1}$$

It is thus equivalent to compute the reachable states of the $a$-focused semantics (for transitions different from $a$), then apply upward closure, and to apply upward closure at every step during the computation of reachable states. In addition, it is obvious that there exists $\widehat{Q}$ in a set, $\widehat{Q} \neq \dashv$, if and only if there exists $\widehat{Q}'$ in the upward closure of the same set such that $\widehat{Q}' \neq \dashv$. It is thus equivalent to test for a “may hit” on the reachable states of the $a$-focused semantics and on their upward closure.

This again suggests two possible simplifications to the focused semantics if our goal is to find places where an access to $a$ may be a hit:

**Closure** Replace $F_{\odot}(BB_i)$ by its up-closure $F_{\odot}^{\uparrow}(BB_i)$: $\widehat{Q}' \in F_{\odot}^{\uparrow}(BB_i)$ if and only if there exists $\widehat{Q} \in F_{\odot}(BB_i)$ such that $\widehat{Q} \subseteq \widehat{Q}'$.

**Subsumption removal** Replace $F_{\odot}(BB_i)$ by the antichain of its minimal elements: $\widehat{Q} \in F_{\odot}^{\min}(BB_i)$ if and only if $\widehat{Q} \in F_{\odot}(BB_i)$ and there is no $\widehat{Q}' \in F_{\odot}(BB_i)$ such that $\widehat{Q}' \subsetneq \widehat{Q}$.

Note that $F_{\odot}^{\uparrow}(BB_i)$ is the up-closure of $F_{\odot}^{\min}(BB_i)$, and that $F_{\odot}^{\min}(BB_i)$ is the antichain of minimal elements of $F_{\odot}^{\uparrow}(BB_i)$; thus $F_{\odot}^{\min}(BB_i)$ is just an alternative representation for $F_{\odot}^{\uparrow}(BB_i)$. Our idea is to directly compute $F_{\odot}^{\min}(BB_i)$.

### A Remark on Lattice Height

We replace the focused semantics by its upward or downward closure; this is a form of convergence acceleration, albeit one that preserves the precision of the final result. We shall see in 4.4 that this improves practical performance considerably compared to a version that checks the focused semantics in a model checker. It is however unlikely that this improvement translates to the worst case; let us see why.

The number of iterations of a data-flow or abstract interpretation analysis is bounded by the height of the analysis lattice, that is, the maximal length of a strictly increasing sequence. However, this height does not change by imposing that the sets should be lower (respectively upper) closed: just apply the following lemma to $T$, the set of subsets of blocks of cardinality at most $k-1$ (plus $\dashv$) ordered by inclusion (respectively, reverse inclusion).

**Lemma 101.** *Let $(T, \leq)$ be a partially ordered finite set. The lattice of lower subsets of $T$, ordered by inclusion, has height $|T|$, the same height as the lattice of subsets of $T$.*

*Proof.* Order $T$ topologically: $t_1, \dots, t_{|T|}$, such that $\forall i, j : \ t_i \leq t_j \implies i \leq j$. The sequence $(u_i)_{i=0,\dots,|T|}$, with $u_i = \{t_1, \dots, t_i\}$, is a strictly ascending sequence of lower sets. □

### 4.3.2 Data Structures and Algorithms

In 4.3.1 we defined a collecting semantics for concrete cache states, then, in two steps (1. focused semantics, 2. closures), showed that there is a cache hit (respectively, a cache miss) in the concrete semantics if and only if there is a cache hit (respectively, a cache miss) in an upward-closed (respectively, downward-closed) semantics, and that upward-closed (respectively, downward-closed) sets may be represented by the antichains of their minimal (respectively, maximal) elements.

#### Computation by Abstract Interpretation

The abstracted semantics in upward-closed (or, downward-closed) sets may be computed by a standard data-flow/abstract interpretation algorithm, by upward iterations, as follows.

To each initial control point we initially attach an initialization value (see below). For a semantics focused on accesses to $a$, we consider that each access to the focus block $a$ is reset the current abstract value to the $\emptyset$. Then, we iterate in the usual abstract interpretation fashion: we maintain a "working set", initially containing the initial locations; we take a control point $x$ from the working set, update the abstract values at the end point of edges going out of $x$ (using the union operation on upper or lower sets), and add these end points to the working set if their value has changed (equality testing). The iterations stop when the working set becomes empty. It is a classical result [Cou78, §2.9] that the final result of such iterations does not depend on the iteration ordering, and in fact several elements from the working set may be treated in parallel; the only requirement is that all elements from the working set are eventually treated.

The sequence of updates to the set decorating a given control location is strictly ascending, in a finite lattice; thus its length is bounded by the height $h$ of that lattice. If $Access$ is the set of memory accesses, then the total number of updates is bounded by $|Access| \cdot h$. Recall that the height of the lattice of subsets of a set $X$ is $|X|$.

If we implement the focused semantics directly, then we compute over sets of subsets of size at most $k-1$ of $Blocks \setminus \{a\}$, completed with $\dashv$; the number of such subsets is bounded by $\sum_{k=0}^{k-1}(|Access| - 1)^k$ and thus $h \leq \frac{(|Access|-1)^k}{|Access|-2} + 1$. The cost could thus be exponential in the associativity; Indeed, as seen in 3, a polynomial-time algorithm is unlikely, since the problems are NP-complete.

#### Closed Sets Implementation

We initially attempted adding closure steps to the focused semantics, and running a model checker on the resulting systems. The performance was however disappointing, worse than model-checking the focused semantics itself as described in Section 4.2. The model checker ( nuXmv ) was representing its sets of sets of blocks using state-of-the-art binary decision diagrams; we thus did not expect any gain by going to our own implementation of iterations over the same structure. We thus moved from representing a closed set by its content to representing it by the antichain of its minimal (respectively, maximal) elements. There remains the question of how to store and compute upon the antichains representing those sets.

We then tried storing an antichain simply as a sorted set of subsets of $Blocks$, each subset being represented as the list of its elements. Experimentally, this approach was inefficient; let us explain why, algorithmically. For once, when computing the antichain for the union of two upward or downward closed sets $S$ and $S'$, one takes the antichains $W$ and $W'$ representing $S$ and $S'$ and eliminates redundancies; if such a naive representation is used, one needs to enumerate all pairs of items from $W \times W'$ — there is no way to immediately identify which parts of $W$ and $W'$ are subsumed, or even to identify which parts are identical. Furthermore, there is no sharing of representation between related antichains.

Binary decision diagrams are one well-known data structure for representations of sets of states; as explained in Section 2.2 they share identical subsets, and allow fast equality testing. All operations over such diagrams can be "memoïzed", meaning that when an operation is run twice between identical subparts of existing diagrams, the result may be cached. We store an antichain, a set $S \neq \{\dashv\}$ of sets of blocks, as a *zero-suppressed decision diagram* (ZDD) [Min01, Mis14] [Knu11, §7.1.4, p.249], a variant of binary decision diagrams optimized for representing sets of sparse sets of items.

#### Basic Functions for exact Always-Miss and exact Always-Hit Analyses

We assume that all control states are reachable (unreachable states are easily discarded by a graph traversal). The starting points of the analyses focused on $a$ is the CCG initial control points.

The operations that we need for antichains defining upper sets, for the Always-Miss analysis, are

**Initialization to empty cache** Return $\{\dashv\}$.

**Initialization to undefined cache state** Return $\{\emptyset\}$.

**Initialization to unreachable state** Return $\emptyset$.

**Access** to address $b \neq a$: Return $\{\widehat{Q} \cup \{b\} \mid \widehat{Q} \in S\}$ if $S \neq \{\dashv\}$, $\{\dashv\}$ otherwise.

**Test for eviction** Returns whether there exists $\widehat{Q} \in S$ such that $|\widehat{Q}| \geq k$, in which case $S$ is replaced by $\{\dashv\}$.

**Access to tracked block** $a$: Return $\{\emptyset\}$

**Limitation to associativity** Let $X = \{\widehat{Q} \mid \widehat{Q} \in S \wedge |\widehat{Q}| \leq k-1\}$. Return $\{\dashv\}$ if $X = \emptyset$, and $X$ otherwise [5].

**Union of upper sets** represented by antichain of minimal elements of $S$ and $S'$: Return $\{\dashv\}$ if $S = S' = \{\dashv\}$. Return $S$ if $S' = \{\dashv\}$ and conversely. Return $\{\widehat{Q} \mid \widehat{Q} \in S \wedge \neg\exists \widehat{Q}' \in S'\ \widehat{Q}' \subsetneq \widehat{Q}\} \cup \{\widehat{Q}' \mid \widehat{Q}' \in S' \wedge \neg\exists \widehat{Q} \in S\ \widehat{Q} \subsetneq \widehat{Q}'\}$ otherwise.

**Equality testing** given $S$ and $S'$, return whether $S = S'$.

**Example 102.** *Let $S$ be the upper set generated by the antichain $\{\{a\},\{b,c\}\}$, and $S'$ the upper set generated by the antichain $\{\{b\},\{a,c\},\{d\}\}$. The union of the two upper sets is an up-set generated by the union of these two antichains. However, this union is not an antichain because it contains redundant items: $\{a,c\}$ is subsumed by $\{a\}$, $\{b,c\}$ is subsumed by $\{b\}$. The antichain of minimal elements of $S \cup S'$ is thus $\{\{a\},\{b\},\{d\}\}$.*

[5] This means that execution traces that cannot lead to a "hit" on the next access to $a$ are discarded. When the last trace leading to hit is removed, abstract value is set to $\{\dashv\}$.

The operations that we need for antichains defining lower sets, for the may-miss analysis, are

**Initialization to empty or undefined cache state** Return $\{\dashv\}$.

**Initialization to unreachable state** Return $\emptyset$.

**Accesses** Same as with upper sets.

**Test for eviction** Returns whether there exists $\widehat{Q} \in S$ such that $|\widehat{Q}| \geq k$, in which case $S$ is replaced by $\{\dashv\}$

**Union of lower sets** represented by antichains of maximal elements $S$ and $S'$: Return $\{\dashv\}$ if $S = \{\dashv\}$ or $S' = \{\dashv\}$. Return $\{\widehat{Q} \mid \widehat{Q} \in S \wedge \neg\exists \widehat{Q}' \in S'\ \widehat{Q} \subsetneq \widehat{Q}'\} \cup \{\widehat{Q}' \mid \widehat{Q}' \in S' \wedge \neg\exists \widehat{Q} \in S\ \widehat{Q}' \subsetneq \widehat{Q}\}$ otherwise.

**Equality testing** Same as with upper sets.

The union of antichains with subsumption removal was supported by an extension [Mis14] of the ZDD library that we used. The only operations not supported were the test for eviction and the limitation to associativity. We implemented them by recursive descent over the structure of the ZDD, with an extra parameter for the current depth (number of items already seen in the set), and memoïzation of the results. As in the CUDD library, we call "then" the branch where the top variable is true (i.e. the branch that contains the cache block associated to the current node) and "else" the branch associated to value false (i.e. the branch that does not contain that cache block). As shown in Algorithm 1, the general case (case 3) of the algorithm simply consists in truncating the "then" and "else" branches of the current nodes. When the number of "then" branches taken reaches the associativity (case 2), we remove all further "then" branches (they only lead to sets of cardinality greater than the associativity). Finally, the algorithm may stop exploring a branch for two different reasons: a) either the node treated is a leaf of the ZDD (case 0), or b) the result of the truncate function has already been computed and memoïzed (case 1).

**Algorithm 1** Truncate(zdd, n) as a recursive function

```
 1: function TRUNCATE(zdd, n)
 2:     if zdd = ∅ or zdd = {∅} then
 3:         return zdd                                   ▷ Case 0. Leaf of the ZDD DAG
 4:     end if
 5:     res ← cacheLookup(Truncate, zdd, n)               ▷ Case 1. Already computed
 6:     if res then
 7:         return res
 8:     end if
 9:     if n = 0 then                                     ▷ Case 2. Associativity is reached
10:         return TRUNCATE(zdd.else, 0)                  ▷ Case 2. Else branch recursion
11:     else                                              ▷ Case 3. General case
12:         then ← TRUNCATE(zdd.then, n − 1)              ▷ Case 3. Then branch recursion
13:         else ← TRUNCATE(zdd.else, n)                  ▷ Case 3. Else branch recursion
14:         return ZDD(zdd.var, then, else)
15:     end if
16: end function
```

## 4.4 Experiments

In order to evaluate the efficiency of our exact analyses, we run several experiments that are described in this section. In particular, the three following points are discussed below:

- First, the actual goal of an exact analysis is to classify more accesses. We thus evaluate the precision of our exact analyses by measuring the gain of accesses classified as *Always-Hit* and *Always-Miss* relatively to the May and Must analyses.
- Then, we investigate the behavior of our two exact approaches in term of analysis time, to measure the gain of the antichain approach over the model checking based solution.
- Finally, we compare the efficiency of the best of our two exact analyses to the May Must approach.

### 4.4.1 Refinement of Accesses classification by Exact Analyses

As a first experiment, we look at the improvement on the classification of accesses. Because one is usually more interested in classifying accesses in *Always-Hit* and *Always-Miss*, we measure the number of additional accesses classified as such and compare it to the number of accesses that were already classified as such by the May and Must analyses. Figure 4.15 shows the that

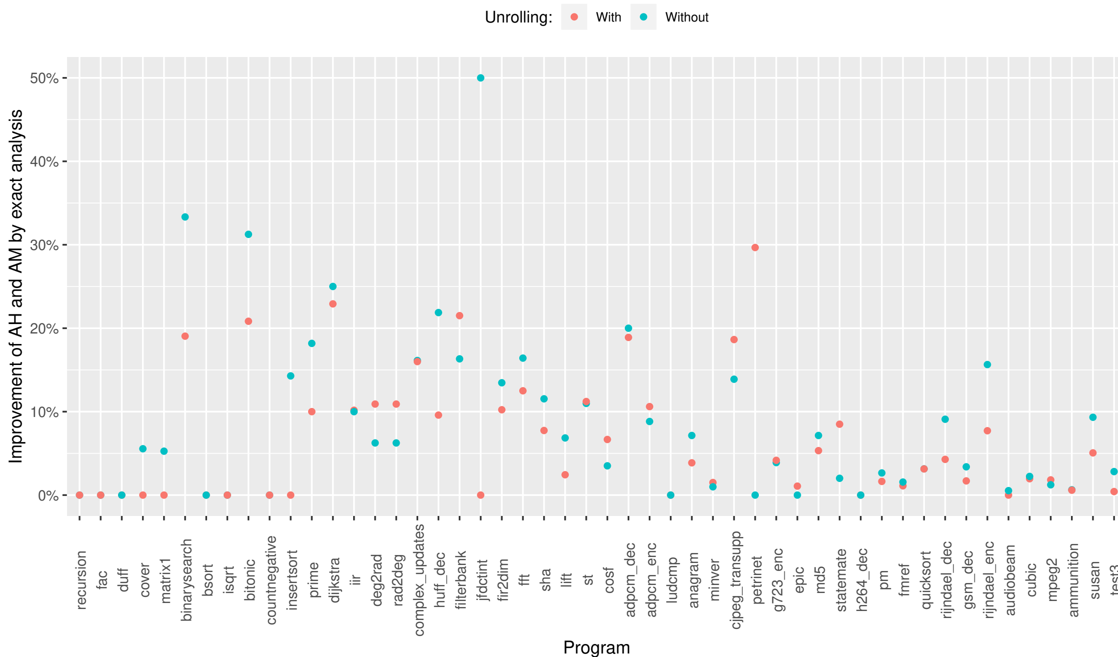



Figure 4.15 – Classification improvement

an exact analysis improves the classification of the May/Must approach in the vast majority of cases. In our settings, performing an exact analysis leads to an average of 18.2% (18.8% when unrolling the loops) of blocks classified as *Always-Hit* or *Always-Miss*, whereas the May and Must analyses only classified 16.8% (17.7% when unrolling loops). This also suggests that performing an exact analysis leads to better classification than unrolling loops, without increasing the size of the program model. Finally, note that both the antichain and the model-checking approaches leads to the same conclusion, because they compute the same classification.

### 4.4.2 Efficiency comparison of Model Checking and ZDD approach

To validate the benefit of pruning subsumed younger sets, we compare the analysis time of both exact analyses[6]. Results are shown on Figure 4.16 (log. scale). In particular, one can notice that the ZDD based approach is faster on all significant benchmarks, and that the observed speed-up often reaches more than 100. More precisely, the average speed-up (geometric mean) is 19.5 in the classical settings, and 12.7 when unrolling loops[7]. The maximum speed-up reached is 297.4 (322.5 when unrolling the first iteration of loops).

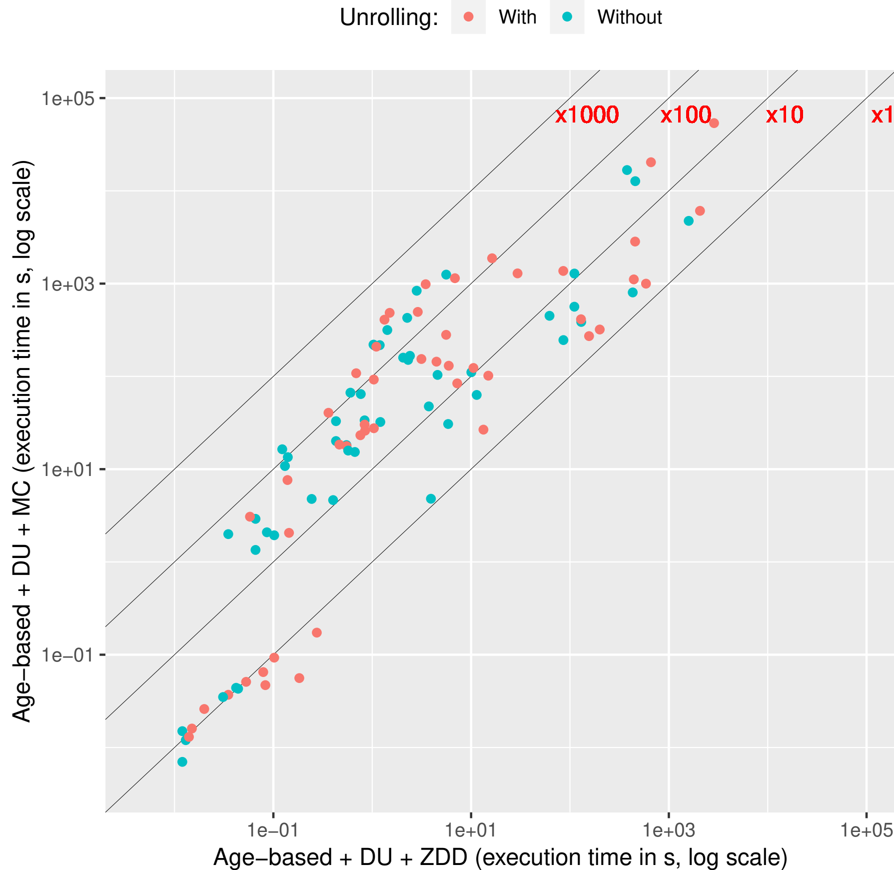



Figure 4.16 – Efficiency comparison in analysis time

### 4.4.3 May/Must and Exact analyses execution time comparison

Finally, to ascertain the efficiency of the exact analysis (May/Must analysis + Definitely Unknown analysis + ZDD approach), we compare it to the May/Must analysis. Figure 4.17 presents the analysis time required by both approaches. For the vast majority of benchmarks, we are able to provide a precise classification in less than 10 times the delay required by the May/Must analysis. In particular, it shows that one can obtain an exact classification in 4.12 times more time in average (5.05 when unrolling loops). Finally, note that some measure for the smallest benchmark are noisy.

[6]Here, we compare the ZDD approach to BDD-based model checker. Our experiments tend to show that using IC3 instead of the BDD-based algorithm does not change the results.

[7]note that this second measure does not take into account two benchmarks for which the model checking approach reached a time out.

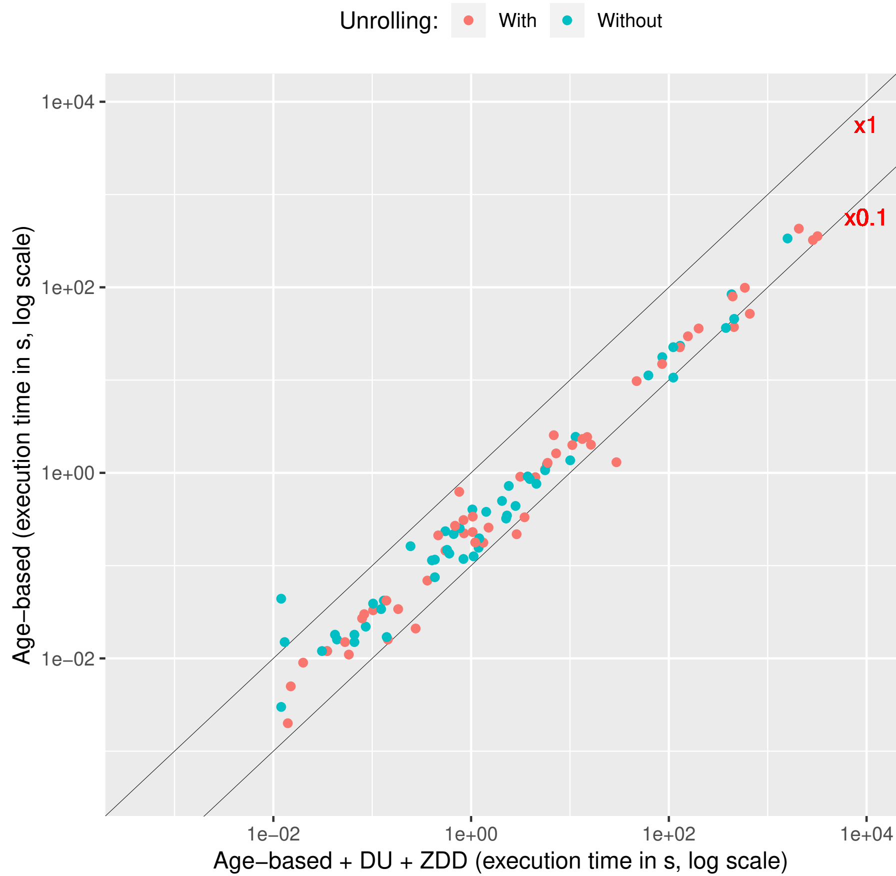



Figure 4.17 – Efficiency comparison in analysis time

Indeed, using our approach, it is impossible to obtain an exact analysis faster than performing the May and Must analysis: our analysis performs a May/Must approximation in the first place. The small variation observed is due to the fact that the two measures (May/Must only, and exact approach) come from different runs of the analyzer.

# Chapter 5

# Applications

## 5.1 WCET Application

Deciding if the WCET of a program is below a given bound, for all execution paths, all inputs, all possible hardware starting configurations, has prohibitive complexity. One thus usually relies on an over-approximation of the WCET instead. The challenge is then to design safe analyses (i.e; that guarantee that the obtained bound is an over-approximation of the WCET) that yield bounds close enough to the actual WCET to be useful. In order to obtain precise bounds on WCET, one must take into account the microarchitecture (pipelines, caches, etc.).

Instruction pipelines provide instruction-level parallelism by splitting the execution of instructions into several stages. An instruction can thus be fetched while another is decoded, a third is executing and a last one is writing to the register file. Thus, instruction executions overlap, and neglecting the pipeline behavior, by assuming that in the worst case instructions execute as though sequentially, would result in a very pessimistic WCET estimation. In some architectures, pipeline models are complex and must take into account, for instance, certain instructions using certain execution units for several cycles and blocking other instructions that might use these units. The number of pipeline states to consider may thus explode.

In addition to pipeline effects, caches must be condidered when bounding the WCET of a program. Indeed, retrieving data from the main memory (*cache miss*) instead of the cache (*cache hit*) can be several orders of magnitude more costly. Naively assuming that every memory access results in a cache miss would thus highly over-estimate the actual WCET of the analyzed program. The purpose of the cache analysis is thus to classify memory accesses into *cache hits* (data is retrieved from the cache) and *cache misses* (data is retrieved from main memory, or from a more distant cache).

Unfortunately, cache memories and pipelines interact, and are thus difficult to analyze in complete isolation. Indeed, the behavior of the pipeline may depend on whether some instruction is in the cache or not. For instance, an out-of-order processor pick the instructions to execute among the instructions already issued (i.e. once the processor knows the instructions dependencies). Depending on instruction cache misses, some of the instructions might not be available. In case the cache analysis is unable to classify a memory access as a hit or a miss, both cases have to be treated to precisely derive the possible pipeline states, with two consequences: the computed WCET bound may become too pessimistic, and the pipeline analysis needs to take more pipeline states into account. This may lead to combinatorial explosion inside the pipeline analysis. Thus, deriving precise information about the cache behavior can reduce the cost of the pipeline analysis in addition to reducing the WCET estimation.

Caches differ in a number of architectural parameters, including their *replacement policy*,

which defines which memory block is to be evicted from the cache to make room for newer data. Analyses are specific to each replacement policy. Chapter 4 proposes precise analyses of instruction caches implementing the *Least Recently Used* (LRU) policy. We show that these analyses are optimally precise under the assumption that all program paths are feasible, meaning they give the most precise sound classification of accesses into hits, misses, and "it depends on the execution". These analyses are costlier than those commonly used for that purpose (which classify into hits, misses, and "I don't know"), typically 4 times slower. In this section we show that despite that, our precise instruction cache analysis can greatly reduce (more than 10 hours) the overall analysis time compared to the classical analysis, because the added precision can greatly reduce the number of cases that the pipeline analysis has to take into account, and thus the cost of the pipeline analysis.

In the following section, we give some background about micro-architectural analyses: in particular how pipeline analyses compute execution times from instruction sequences. Section 5.1 explains how our precise cache analysis speeds up the pipeline analysis used in OTAWA , a WCET estimation framework. Finally, Section 5.2 describes our experiments on the impact of cache analysis on pipeline analysis, and interprets the results.

**Pipeline Analysis**

Pipelined processors split the execution of instructions into several smaller steps performed by different parts (called pipeline stages) of the CPU circuit. Instead of waiting for one instruction to complete before executing the next one, as in older or simpler processors, a pipelined processor thus executes several instructions in parallel. One often distinguishes five stages: fetch, decode, execute, memory, and write-back. The execute step of an instruction and the decode step of the following instruction may then be performed simultaneously. In the ideal case, the delay to issue one instruction is equal to the time needed for an instruction to go through one of the pipeline stages. The ideal pipeline behavior is shown in Table 5.1. In this example, each of the 5 instructions require 5 cycles to be executed. However, the whole sequence of instructions can be executed in only 9 cycles by using all the pipeline stages in parallel.

| | Clock cycle | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Inst | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | FE | DE | EX | MEM | WB | | | | |
| 2 | | FE | DE | EX | MEM | WB | | | |
| 3 | | | FE | DE | EX | MEM | WB | | |
| 4 | | | | FE | DE | EX | MEM | WB | |
| 5 | | | | | FE | DE | EX | MEM | WB |

Table 5.1 – Instruction pipeline

In practice, starting the execution of an instruction stage before the end of the previous one is not always possible. One reason is that some instructions may use one pipeline stage for more than one cycle. For example, some complex floating-point arithmetic operations may use the same functional unit for several cycles in the execution stage; thus other arithmetic instructions entering the pipeline afterwards may be stalled until the operation is complete.

In addition to these resource conflicts, two instructions may exhibit a control dependency or a data dependency. A data dependency occurs when the result of an instruction is used by a following instruction: then, the execution step of the second instruction must occur after the write-back step of the first instruction. In some processors, the old (and incorrect) value read in

the register file would then be used as an operand for the second instruction; more commonly, the second instruction is stalled until its operand is available. Such an extra delay is called a bubble. When a bubble is introduced in a pipeline stage, instructions in the preceding stages are stalled for one cycle, whereas instructions in following stages continue their executions.

Micro-architectural events external to the pipeline, such as cache misses or branch misprediction, can also lead to bubbles: indeed, if an instruction needs as operand a value read by a preceding load instruction, but this value is not in the cache yet, it will be stalled until the value is available. All these sources of pipeline delays might accumulate or partially compensate each other, making the exact execution time of a basic block highly dependent on the execution context (pipeline state, cache state, etc.). In out-of-order processors, that is, processors that can execute instructions not in their semantic order depending on the availability of operands and execution units, things are even more complex.

In the following, we briefly describe the pipeline analysis as performed by OTAWA . A more precise description is available in [RS09]. As a first approximation, the pipeline analysis consists in evaluating the WCET of every basic block, i.e. the WCET of the associated instruction sequence. As mentioned previously, the execution time of such a sequence is influenced by dependencies and resource conflicts between instructions involved. To extract a safe WCET approximation, all hazards that can occur during the basic block execution have to be considered. Among all the possible hazards combinations in the basic block instruction sequence, the one that results in the highest execution time defines the basic block's WCET.

This naive approach is safe, but it might lead to pessimist WCET bounds because it treats basic blocks in isolation. In particular, no assumption can be made about the initial pipeline state to derive safe WCET bounds. An improvement [RS09] is to take the execution context into account when evaluating the WCET of a basic block. Ideally, the context would define the exact set of pipeline states reachable at the basic block entry. This is however not feasible in practice, because of the huge number of program paths to consider. Instead, Otawa, when analyzing a basic block, uses a partial context derived from the execution of the previous basic block [RS09]. More precisely, if the basic block $v$ follows the basic block $u$, the pipeline analysis evaluates the WCET of $v$ by considering the instruction sequence $uv$ as a whole. The WCET of $v$, defined as the delay between the termination of the last instruction of $u$ and the termination of the last instruction of $v$, is then expressed as a function of the pipeline state at the entry of $u$. In case the block $v$ has several predecessors, the process is repeated for each of them, and the maximum of the WCET obtained is used to define the bound on the WCET of $v$.

In addition to the pipeline dynamics, the pipeline analysis has to take into account external events, which are sources of nondeterminism. A memory load results in different behaviors of the pipeline depending on how many cycles it takes to complete, which in turn depends on whether the requested data is in the cache. For a "definitely unknown" cache access, both hit and miss cases must be taken into account by the pipeline analysis. This also applies to accesses classified as "unknown" by an imprecise cache analysis, even though they always result in a hit or in a miss. Thus, lack of precision in the cache analysis may result not only in overestimation of WCET, but also in increases in the number of combinations to be considered in the pipeline analysis and thus in higher analysis times.

Note that it is not always safe to take only the miss case into account if the goal is to obtain a safe upper bound on the WCET. Some architectures indeed exhibit so called *timing anomalies*: a local increase of the execution time (e.g. because of a cache miss) may lead to a global reduction of the execution time [RWT+06].

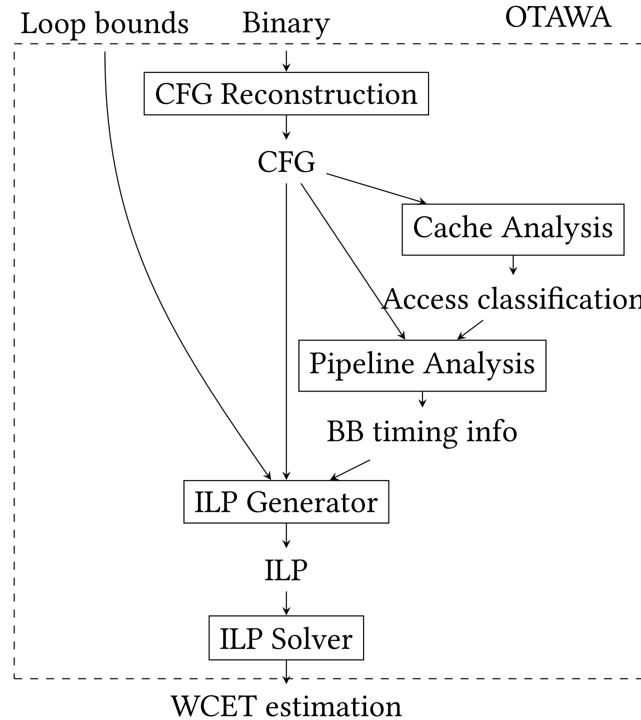



Figure 5.1 – OTAWA Workflow

The precise cache analysis described in Chapter 4 is implemented as a plugin of OTAWA , a framework for WCET estimation. The overall workflow of OTAWA is shown on Figure 5.1. First, the program CFG is reconstructed from the binary code. Then, the cache analysis explores this CFG and classifies every memory access as *Always-Hit*, *Always-Miss* or *Definitely Unknown*. Using this information, the pipeline analysis provides a WCET for each basic block as described above. Timing information is then used to produce an integer linear programming problem with an objective function modeling the execution time, and a set of constraints that encode the CFG structure and the loop bounds[1]. Maximizing the objective function under these constraints gives an upper-bound on the actual program Worst-Case Execution Time.

The default version of OTAWA uses the classical age-based LRU analysis [AFMW96]. We thus compared OTAWA with that imprecise analysis to OTAWA with our exact analysis.

## 5.2 Experiments

We evaluated the impact of our precise cache analysis on the WCET analysis on two aspects:

1. We measured how tighter our precise analysis makes the WCET bounds. Indeed, by running a precise cache analysis some block classified as *Unknown* are refined into *Always-Hit* or *Always-Miss*. This, especially refinements into *Always-Hit*, may result in a tighter overall bound.

2. We measured how much time was needed to perform the cache analysis, the pipeline analysis, and the overall analysis, using the classical and the exact cache analyses.

[1]Unbounded loops obviously lead to unbounded execution time.

To evaluate these two aspects, we analyzed 50 benchmarks from the TACLeBench [FAH+16] benchmark collection. Each benchmark was analyzed twice: first using the classical May/Must cache analysis (see [AFMW96]), then using our precise cache analysis (see Chapter 4).

The instruction cache we consider stores memory blocks of 16 bytes, has associativity 8 and 4 cache sets. This low number of cache sets is not a limitation of our approach, but it is mandatory to get meaningful results despite the relatively small size of the benchmarks. Indeed, when using a higher number of cache sets (64 is a common number for L1 instruction caches), the number of cache blocks mapping each cache set is very low, and lower than the associativity, thus no eviction takes place and analysis is trivial. We assume a delay of 10 cycles when accessing the main memory (i.e. in case of cache miss) and a one cycle delay when accessing the cache (i.e. in case of cache hit).

The pipeline analyzed consists of 4 stages (fetch, decode, execute and commit). Each of this stage has a latency of 1 cycle. In particular, the execute stage has a unique functional unit which is able to perform any instruction in one cycle. In addition, the pipeline introduces a queue between the fetch and decode stage that is able to store up to 2 instructions.

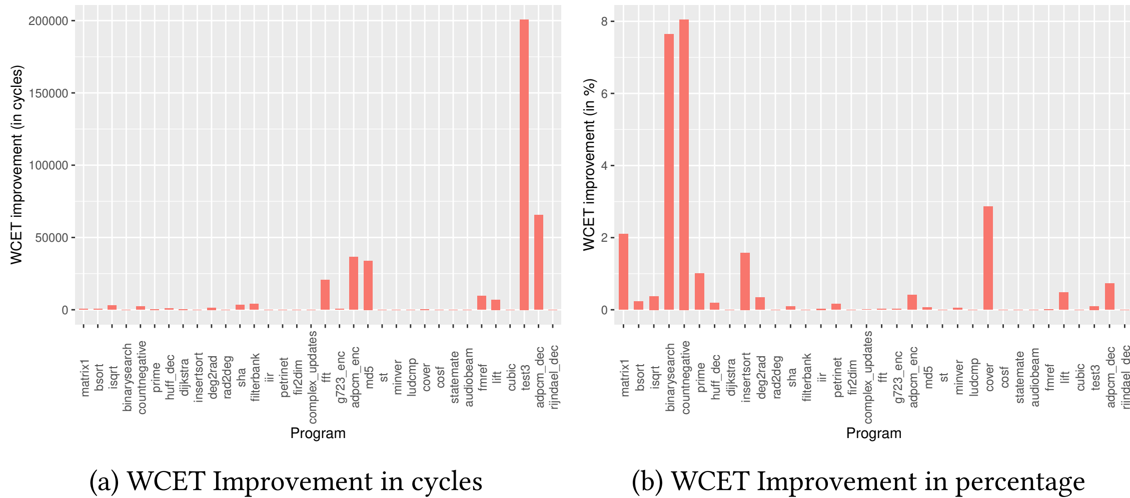
Because of the prohibitive cost of running the complete pipeline analysis on some benchmarks, we limit the total WCET analysis time to 24 hours. This timeout is reached when the program under analysis has some basic blocks with long instruction sequences. For these blocks, the high number of events leads to an intractable number of executions.

Finally, note that we do not unroll the first iteration of loops in these experiments. If OTAWA provides a CFG transformation pass for unrolling loops, the effect on loop bounds is not automatically estimated. For instance, in the case of a loop that can execute at most $n$ times, one can find in the program ILP a constraint of the form $x_{\text{backedge}} \leq n$ where $x_{\text{backedge}}$ is the variable indicating how many times the loop backedge has been executed. When the first iteration of the loop is unrolled, this constraint should be adapted to take into account the unrolled iteration and become $x_{\text{backedge}} \leq n - 1$. Another possibility is to locate in the graph the edge corresponding to the unrolled backedge and to add the constraint $x_{\text{backedge}} + x_{\text{unrolled_backedge}} \leq n$. In case of nested loops, both methods can be used, either adapting the arithmetical expression used to bound the number of execution of all backedges, either bounding the sum of executions of all copies of the same backedge. These modifications of the ILP are not automatically handled by OTAWA , leading to coarse WCET over-estimation when unrolling loops even with more precise pipeline and cache analyses results. Thus, we did not unroll the loops for this set of experiments

### 5.2.1 WCET comparison

As mentioned above, the bound on the WCET is expected to be lower when performing a precise cache analysis. However, the reduction is modest: on average (geometric mean), the WCET estimation was reduced by only 0.8%. Figure 5.2a and Figure 5.2b show the improvement of the WCET in number of cycles and percentage. Note that we only show benchmarks which both WCET analyses terminate in less than 24 hours.

This low reduction can be explained by looking at the details of the cache analysis results. Indeed, among the accesses refined using our precise analysis, most of them are refined from *Unknown* to *Always-Miss* or *Definitely Unknown*, and not from *Unknown* to *Always-Hit*. Refining a block from *Unknown* to *Always-Miss* may reduce the WCET estimation due to timing anomalies, but this is not the case on the simple platform we use. Generally, it is more probable that refining an *Unknown* access to *Always-Hit* leads to a better WCET bounds than refining it to *Always-Miss*.




(a) WCET Improvement in cycles

(b) WCET Improvement in percentage

Figure 5.2 – WCET reduction under exact cache analysis

### 5.2.2 Analysis time comparison

In order to evaluate the benefits of running a precise cache analysis to speed the pipeline analysis up, we look at the cumulative costs of both analyses. In our experimental settings, this total cost is almost equal to the whole WCET analysis cost. The other steps of the WCET analysis, including CFG reconstruction and ILP solving, have minor cost in comparison to cache and pipeline analyses.

Table 5.2 shows the analysis time of all benchmarks (cache analysis, pipeline analysis, and total of both analyses) depending on the cache analysis employed. Differences of total analysis time in bold in the last column enhances the benchmarks where the precise cache analysis leads to shorter WCET analysis. It shows that our approach leads to important reductions of the global analysis time.

Note that for some benchmarks (adpcm_dec, ammunition, duff, epic, fir2dim, fmref, ludcmp, statemate, in italic in Table 5.2), the pipeline analysis is longer when a precise cache analysis is performed. This can be due to some noise in the measurements, but another phenomenon can explain these deviations. Indeed, in addition to derive all the possible execution time of a basic block, the pipeline analysis summarizes these values by two values, one over-approximating the "normal" execution time of the basic block, and one "exceptional" execution time taking into account the penalties due to events. Intuitively, events that are frequents, but have low penalties should be accounted into the "normal" approximation, whereas events that are rare but costly should be accounted in the "exceptional" approximation. The algorithm classifying execution time into these two categories is usually cheap in comparison to the pipeline states exploration. However, it might explain the counter-intuitive phenomenon observed. For instance, assuming one event (like a cache miss) is much more costly than the others, it will be considered as the "exceptional" behavior and treated separately. If this event is removed by the precise cache analysis, then the pipeline analysis will try to split the remaining events into the "normal" ones and "exceptional" ones. Depending on the algorithm used to perform the split, this might lead to a higher pipeline analysis time.

Table 5.2 also shows that for one benchmark (gsm_dec) the pipeline analysis terminates because the number of events was reduced by the precise cache analysis. For this benchmark the pipeline analysis does not finish in 24 hours under the May/Must analysis, whereas it terminates in less than 2 hours under our precise cache analysis. As mentioned previously, this compensa-

| | Analysis time (in s) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Cache Analysis | | | Pipeline Analysis | | | Total | | | |
| program | May/Must | Precise | Difference | May/Must | Precise | Difference | May/Must | Precise | Difference | Ratio |
| adpcm_dec | < 0.1 | 0.2 | 0.2 | *13157.2* | *13160.3* | *3.1* | 13157.2 | 13160.5 | 3.3 | 1.00 |
| adpcm_enc | < 0.1 | 0.2 | 0.2 | 4.2 | 3.8 | -0.4 | 4.2 | 4.0 | **-0.2** | 0.98 |
| ammunition | 733.2 | 3232.9 | 2499.7 | *15.9* | *16.0* | *< 0.1* | 749.1 | 3248.9 | 2499.8 | 4.34 |
| anagram | 0.2 | 0.5 | 0.4 | 1.7 | 1.4 | -0.3 | 1.8 | 1.9 | < 0.1 | 1.05 |
| audiobeam | 18.6 | 109.7 | 91.1 | 15.4 | 13.8 | -1.6 | 34.0 | 123.5 | 89.5 | 3.63 |
| binarysearch | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | 0.96 |
| bitonic | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | 0.95 |
| bsort | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | 1.32 |
| cjpeg_transupp | 0.2 | 0.6 | 0.4 | 1.5 | 1.3 | -0.2 | 1.7 | 1.9 | 0.2 | 1.13 |
| complex_updates | < 0.1 | 0.3 | 0.2 | 1.6 | 1.6 | < 0.1 | 1.6 | 1.9 | 0.2 | 1.15 |
| cosf | 3.1 | 22.5 | 19.3 | 6.7 | 6.2 | -0.5 | 9.8 | 28.6 | 18.8 | 2.92 |
| countnegative | < 0.1 | < 0.1 | < 0.1 | 0.2 | < 0.1 | -0.1 | 0.2 | 0.2 | < 0.1 | 0.72 |
| cover | 2.6 | 13.9 | 11.3 | 0.3 | 0.3 | < 0.1 | 2.9 | 14.2 | 11.3 | 4.87 |
| cubic | 173.7 | 928.4 | 754.7 | 372.4 | 365.2 | -7.2 | 546.1 | 1293.6 | 747.5 | 2.37 |
| deg2rad | < 0.1 | 0.2 | 0.1 | 0.7 | 0.7 | < 0.1 | 0.8 | 0.9 | 0.1 | 1.14 |
| dijkstra | < 0.1 | < 0.1 | < 0.1 | 0.4 | 0.4 | < 0.1 | 0.4 | 0.4 | < 0.1 | 1.01 |
| duff | < 0.1 | < 0.1 | < 0.1 | *< 0.1* | *0.1* | *< 0.1* | < 0.1 | 0.1 | < 0.1 | 1.25 |
| epic | 0.9 | 5.7 | 4.8 | *606.6* | *614.2* | *7.7* | 607.5 | 620.0 | 12.5 | 1.02 |
| fac | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | 1.41 |
| fft | < 0.1 | 0.4 | 0.4 | 2.1 | 2.0 | < 0.1 | 2.2 | 2.5 | 0.3 | 1.14 |
| filterbank | < 0.1 | 0.3 | 0.2 | 1.1 | 1.1 | < 0.1 | 1.1 | 1.3 | 0.2 | 1.19 |
| fir2dim | 0.1 | 0.9 | 0.7 | *1.2* | *1.3* | *< 0.1* | 1.3 | 2.1 | 0.8 | 1.61 |
| fmref | 42.9 | 197.9 | 155.0 | *14.3* | *15.4* | *1.1* | 57.2 | 213.3 | 156.0 | 3.73 |
| g723_enc | < 0.1 | 0.6 | 0.5 | 3.5 | 1.5 | -2.0 | 3.6 | 2.1 | **-1.5** | 0.59 |
| gsm_dec | 0.2 | 0.7 | 0.5 | timeout | 7095.6 | NA | timeout | 7096.3 | NA[i] | NA[ii] |
| h264_dec | < 0.1 | 0.1 | 0.1 | timeout | timeout | NA | timeout | timeout | NA | NA |
| huff_dec | < 0.1 | < 0.1 | < 0.1 | 0.3 | 0.3 | < 0.1 | 0.3 | 0.4 | < 0.1 | 1.10 |
| iir | < 0.1 | 0.2 | 0.2 | 1.2 | 1.2 | < 0.1 | 1.3 | 1.4 | 0.2 | 1.12 |
| insertsort | < 0.1 | < 0.1 | < 0.1 | 0.8 | 0.7 | < 0.1 | 0.8 | 0.7 | < 0.1 | 0.97 |
| isqrt | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | 0.94 |
| jfdctint | < 0.1 | < 0.1 | < 0.1 | timeout | timeout | NA | timeout | timeout | NA | NA |
| lift | < 0.1 | 0.2 | 0.2 | 1112.4 | 282.3 | -830.0 | 1112.4 | 282.6 | **-829.8** | 0.25 |
| ludcmp | 0.6 | 3.0 | 2.4 | *5.0* | *5.1* | *0.1* | 5.6 | 8.1 | 2.5 | 1.45 |
| matrix1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | 1.46 |
| md5 | < 0.1 | 0.2 | 0.2 | 5.1 | 3.2 | -1.9 | 5.2 | 3.4 | **-1.8** | 0.66 |
| minver | 0.5 | 2.3 | 1.8 | 4.9 | 4.1 | -0.8 | 5.4 | 6.3 | 0.9 | 1.17 |
| mpeg2 | 7.1 | 33.4 | 26.4 | timeout | timeout | NA | timeout | timeout | NA | NA |
| petrinet | < 0.1 | 0.4 | 0.4 | 1.4 | 1.1 | -0.4 | 1.4 | 1.5 | < 0.1 | 1.02 |
| pm | 36.8 | 173.4 | 136.6 | timeout | timeout | NA | timeout | timeout | NA | NA |
| prime | < 0.1 | < 0.1 | < 0.1 | 0.2 | 0.2 | < 0.1 | 0.2 | 0.2 | < 0.1 | 1.14 |
| quicksort | 12.4 | 88.2 | 75.8 | 11.7 | 11.1 | -0.6 | 24.1 | 99.3 | 75.2 | 4.13 |
| rad2deg | < 0.1 | 0.2 | 0.2 | 0.8 | 0.7 | < 0.1 | 0.9 | 1.0 | 0.1 | 1.15 |
| recursion | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | < 0.1 | 1.52 |
| rijndael_dec | < 0.1 | 0.4 | 0.4 | 50927.3 | 14924.3 | -36003.0 | 50927.3 | 14924.7 | **-36002.6** | 0.29 |
| rijndael_enc | < 0.1 | 0.4 | 0.4 | timeout | timeout | NA | timeout | timeout | NA | NA |
| sha | < 0.1 | 0.4 | 0.3 | 0.9 | 0.8 | -0.1 | 1.0 | 1.1 | 0.2 | 1.17 |
| st | 0.6 | 3.7 | 3.1 | 3.6 | 3.6 | < 0.1 | 4.3 | 7.3 | 3.1 | 1.73 |
| statemate | 0.2 | 1.0 | 0.8 | *49.2* | *51.4* | *2.2* | 49.4 | 52.4 | 3.0 | 1.06 |
| susan | 50.3 | 305.2 | 254.8 | timeout | timeout | NA | timeout | timeout | NA | NA |
| test3 | 13.2 | 40.0 | 26.8 | 4933.3 | 4799.6 | -133.8 | 4946.5 | 4839.5 | **-106.9** | 0.98 |

Table 5.2 – Execution time of cache and pipeline analyses

tion of the extra cache analysis cost by the pipeline analysis cost is due to the lower number of states to explore. For example, using the precise cache analysis when analyzing rijndael_dec and lift reduces the cumulated number of events from 599 to 499 for rijndael_dec, and from 257 to 175 for lift. As a consequence, the total analysis time for analyzing these benchmarks drops by 70% (10 hours) and 75% (13 minutes) respectively. On the other hand, when the precise cache analysis fails to reduce the total analysis time, the increase of the wcet analysis time is modest.

Figure 5.3 summarizes the impact of precise cache analysis on the WCET analysis. The blue

[i]Real improvement is beyond -79303.7s

[ii]Real ratio is below 0.0821

bars show the amount of time devoted to cache analysis. The other bar shows the time spent in pipeline analysis. It can be green, when performing the precise analysis make the pipeline analysis faster, or red otherwise. Finally, note that no bar is shown in case the pipeline analysis did not terminate.

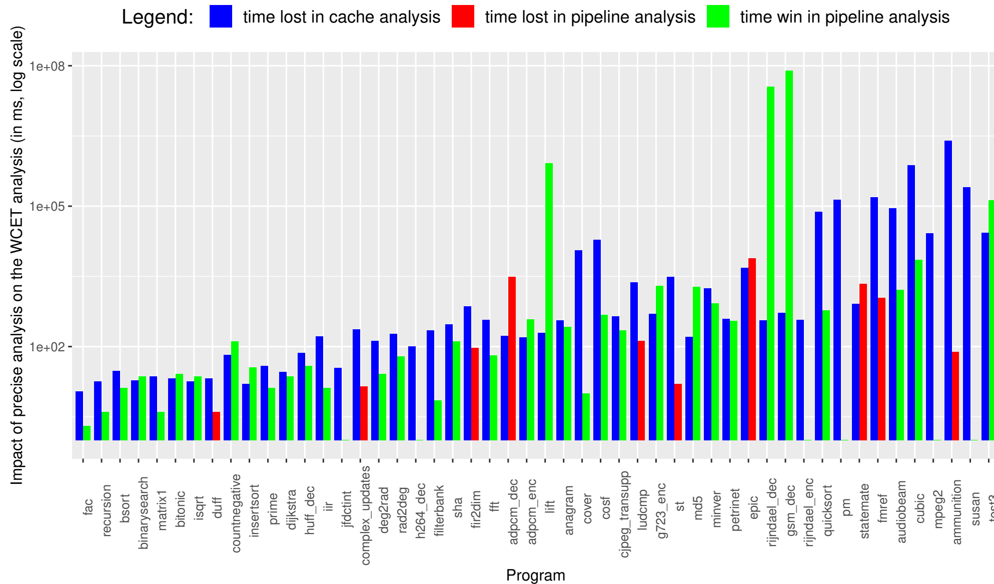



Figure 5.3 – Benefit of the precise cache analysis on overall analysis time

Figure 5.4 compares the number of events (aggregated over all basic blocks) generated by the May/Must cache analysis to the number of events generated by our complete exact cache analysis. This figure only represents benchmarks for which the pipeline analysis terminated. In our experiments, we measure an average reduction of the number of events by 9.2%. Because cache memory is the only source of pipeline events in our setting, this implies that our precise cache analysis classifies 9.2% of the *Unknown* accesses into *Always-Hit* or *Always-Miss* in average.

**Perspectives**

This chapter investigates the relation between cache and pipeline analyses and shows that performing a different cache analysis might impact the performance of the pipeline analysis as well. Moreover, the collaboration between cache and pipeline analyses could be improved. In OTAWA , the pipeline analysis is performed separately in the context of the pipeline states provided by each preceding basic block. However, we compute a single classification of accesses for each basic block. We could instead compute this classification separately for each preceding basic block. This could yield better and faster analysis when certain blocks are classified as "definitely unknown" because they are always a hit when control flows from one preceding basic block, and always a miss when control flows from another basic block.

In addition, our exact cache analysis could be applied on demand, where it seems that improved precision would help most. For example, one could refine accesses occurring in frequently executed basic blocks, based on the provided loop bounds.

Furthermore, it could be interesting to consider the impact of precise cache analysis on other analyses related to caches, e.g. on analyses taking into account task preemption. A task preempting another one performs memory accesses and thus pollutes the content of the cache of the preempted task. When this happens, the preempted task usually suffers additional cache misses

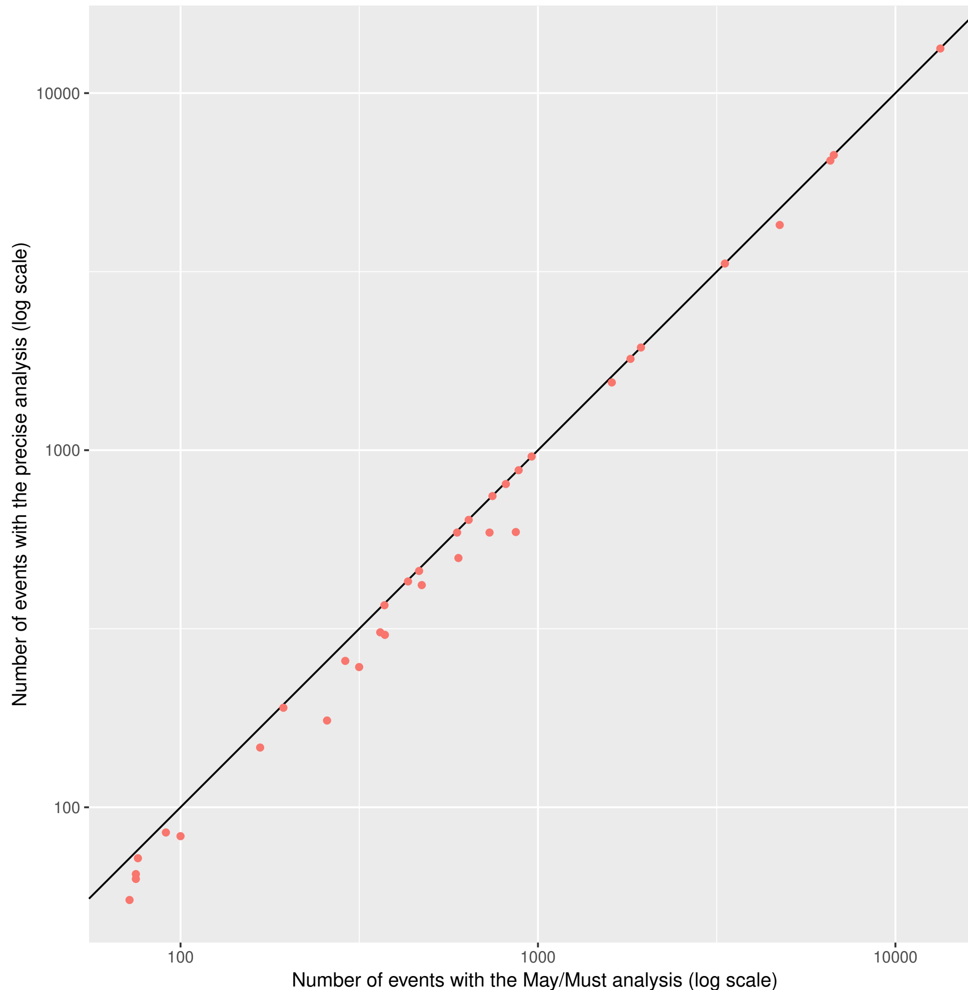



Figure 5.4 – Benefit of the precise cache analysis in terms of pipeline events generated

when resumed, and its associated WCET often increases. Some approaches [AMR10, ZGJ+17] tackle this problem by bounding the maximum Cache-Related Preemption Delay (CRPD) of the preempted task. By refining the classification of accesses in both tasks, our analysis could reduce the CRPD bound if adapted to a preemption-sensitive cache analysis.

Similarly, it would be interesting to evaluate our analysis in the context of multicore CPUs. Indeed, a core might suffer interference from other cores when accessing a shared memory, because the memory bus is already used. In this situation, the memory access is delayed, increasing the task WCET. By refining some memory accesses from the *Unknown* category to *Always-Hit*, our approach could reduce the number of interferences to consider.

### Conclusion

Finally, we have shown that using a more precise cache analysis can reduce the overall time to perform a complete WCET estimation. Indeed, because of the possibility of timing anomalies, a sound pipeline analysis has to consider a high number of pipeline executions, which may become untractable. By reducing uncertainty about the cache behavior, a precise cache analysis eliminates some of these possible executions, speeding the pipeline analysis up. In some cases, a cache analysis performed in less than one second reduces the pipeline analysis from 10 hours or more, bringing the total analysis time to a few hours.

## 5.3 Security

As shown in [Ber05, CLS06, KSWH00], cache memories can be used by an attacker to retrieve secret information from a program execution. Intuitively, the value of some secret data might impact the behavior of the program manipulating them, and caches then provide a way for an attacker to observe these variations of the program behavior. Three kind of side-channel attacks are usually distinguished in the case of caches:

- Timing attacks: Cache memories influence the execution time of program. Thus, by measuring the overall execution time of a victim, an attacker can estimate the number of hits and misses that occurred during the execution. In case these numbers are correlated with the secret data, the attacker get some information about the secret data.
- Access-based attacks: These attacks require the attacker and the victim to share the same platform. In this setting, the attacker fills the cache and measures the latency of its own memory accesses later on. Using these measures the attacker can guess what were the instructions executed or data loaded by the victim, and deduce the secret data. These attacks are usually more powerful than timing attacks because the attacker acquires fine grain information about the cache content. Indeed, the attacker can monitor separately every cache set, or even cache line in case of shared memory between the attacker and the victim.
- Trace-based attacks: In some cases, an attacker measure or guess the outcome (hit or miss) of every cache access of the victim. This can be done for example by measuring the electromagnetic emission or power consumption of the CPU, and is usually done on embedded systems like smart cards. If the attacker possesses (a copy of) the victim program, he can link the hit/miss trace obtained to the execution path taken in the program and get some information about the secret data manipulated.

One solution to avoid information leakage through the cache is to rely on constant-programming techniques, that consist in removing all dependencies between secret data and memory accesses. By doing so, any observation made by an attacker on the program behavior can not be linked back to any sensitive information. The main advantage of this method is that it can be used to counter a vast set of microarchitectural attacks. However, it imposes strong constraint on the program and can easily lead to false positive. To refine this kind of analysis, one can perform hardware analyses that retrieve information about the microarchitectural state that can be used to weaken the hard constraints coming from constant-programming.

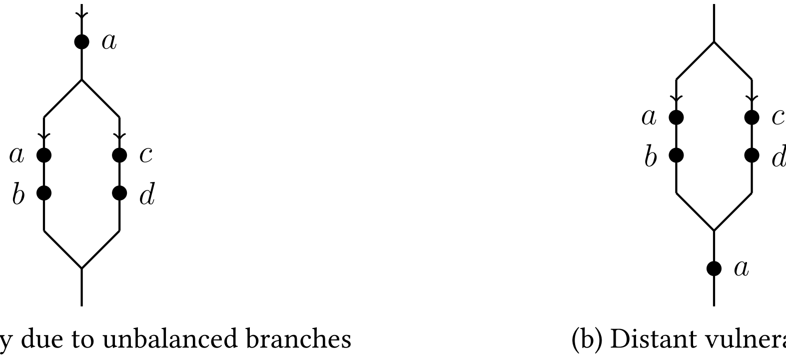
The cache analyses that we know are geared specifically toward a security goal are CacheAudit [DKMR15] and CacheFix [CR18]. CacheAudit uses abstract interpretation to quantify the amount of information leaked by a program, while CacheFix uses an SMT-solver to either prove the absence of side-channels or give examples of differing execution traces and theoretical ways to fix those differences.

In this thesis, we focus on the trace-based attack that target instruction caches. One reason for such attacks being difficult to spot by hand is that a vulnerable access (i.e. that allows the attacker to distinguish between two execution traces) might be located far away from the instruction handling the sensitive data. Figure 5.5 shows two examples of cache conflict graphs vulnerable[2] to trace-based attack. Both cases consist in a test that leaks its outcome through the cache. In the first case (Figure 5.5a), the attacker can get two different hit/miss traces: (*miss*, *hit*, *miss*) on the left path and (*miss*, *miss*, *miss*) on the right path. Obviously, the source of the leak is the repeated

[2]assuming an empty initial cache, a single cache set and 4 ways.

access to the memory block $a$ on the left path. The second case (Figure 5.5b) is similar, but the leak appears at the very end of the program. In this situation, the information leakage occurs after the two branches have merged, because the variation of the cache content induced by the test survives the merging of path. In practice, this kind of information leakage can appear far away from the junction of paths, making them difficult to spot.

This section describes some cache analyses[3] that gives some feedback to the developer, by identifying program points where this kind of distant vulnerability may appear. Note that we do not claim to spot every vulnerability and, for instance, our analyses would not detect the leakage illustrated on Figure 5.5a.




(a) Vulnerability due to unbalanced branches

(b) Distant vulnerability

Figure 5.5 – A CFG and the associated CCG for a fully associative cache

The main idea behind our analyses is that information leakage as illustrated by Figure 5.5b occurs when the outcome of a memory access change depending on a secret value. A coarse approximation of the set of such memory accesses is thus given by the *Definitely Unknown* classification which captures all accesses which outcome can both be hit or miss. Thus, we consider all *Definitely Unknown* memory accesses as potential source of information leakage. In addition, when a *Definitely Unknown* access leads to a hit, it is interesting to know where the block accessed is loaded. Indeed, this information can help the developer to:

- remove some false positives. Locating the access that loads the leaking blocks makes identification of the paths leading to a hit easier. If these path are not feasible, there is no information leakage.
- locate the test which outcome is potentially leaking. For example, if a program consists in a sequence of tests similar to Figure 5.5b followed by a *Definitely Unknown* access, the location where the corresponding block is loaded indicates which of the test is leaking information.

### 5.3.1 Program model and semantics

The program model used to perform our analyses is similar to previous model used and based on the cache conflict graph of a given program and cache set. However, our analyses aim at locating the source of information leakage, i.e. the access that loads the blocks leaking information when accessed again. We thus model the program to analyze as a graph of its control flow, decorated by the memory accesses to the chosen cache set. Because we focus on analyzing the instruction cache these accesses are in fact instructions and are thus assumed to be unique in the graph.

[3]This work was done in collaboration with Marva Ramarijaona.

More formally, the program is modeled by a graph $(Access, E)$, where $E \subseteq Access^2$. We note $I \subseteq Access$ the set of accesses that can be reached from the program entry point without making any access to the chosen cache set. In case of a fully associative cache $I$ is a singleton containing the access to the very first instruction of the program. In addition, we note $blk : Access \rightarrow Blocks$ the function that maps an access to the block it accesses.

In this model, a program execution is described by the sequence $\sigma$ of accesses it performs.

**Definition 103.** *We define $\Sigma$ the set of valid finite access sequences in the program:*

$$\Sigma = \{\varepsilon\} \cup \{\sigma = \sigma_0.\sigma_1.\ldots.\sigma_n \in Access^*, \sigma_0 \in I \land \forall i \in \{0, \ldots, n-1\}, (\sigma_i, \sigma_{i+1}) \in E\}$$

*where $\varepsilon$ is the empty sequence*

$\Sigma$ is will be the concrete domain of our analyses. To classify an access as vulnerable, it is interesting to reason about the set of traces that lead to it. We thus define the collecting semantics of our analyses:

**Definition 104.** *For any access $A \in Access$, we note $F(A)$ this set of traces that reach $A$. Formally, $F$ is defined as the least fixpoint solution of the following equation:*

$$\forall A' \in Access, F(A') = F_0(A') \cup \bigcup_{(A,A') \in E} \{\sigma.A, \sigma \in F(A)\}$$

$$\textit{where } \forall A \in Access, F_0(A) = \begin{cases} \{\varepsilon\} & \textit{if } A \in I \\ \{\} & \textit{otherwise} \end{cases}$$

*Note that the existence of this least fixpoint is guaranteed by Tarski's fixpoint theorem, $2^{\Sigma}$ being a complete lattice and by the monotonicity of $\mathcal{F} : F \mapsto \lambda A'.F_0(A') \cup \bigcup_{(A,A') \in E} \{\sigma.A, \sigma \in F(A)\}$.*

As a consequence of the trace semantics used to define our concrete domain, the concrete transformers takes accesses as parameters and not memory blocks. We thus overload the definition of the $update$ function defined in Chapter 2.3 as follows:

$$\forall q \in D_{LRU}, \forall A \in Access, update(q, A) = update(q, blk(A))$$

This function can then be lifted to an arbitrary number of accesses.

$$update(q, \sigma) = \begin{cases} q & \text{if } \sigma = \varepsilon \\ update(update(q, \sigma'), A) & \text{if } \sigma = \sigma'.A \end{cases}$$

Similarly, all cache analyses mentioned in this thesis can be raised to the trace domain. We thus have, noting $q_0$ the entry cache state of the program:

$$\begin{aligned} \gamma_{May} : D_{May} &\rightarrow \mathcal{P}(\Sigma) \\ \hat{q}_{May} &\mapsto \{\sigma \in \Sigma, \forall b \in Blocks, q(b) \geq \hat{q}_{May}(b), \text{where } q = update(q_0, \sigma)\} \end{aligned}$$

$$\begin{aligned} &\gamma_{EH} : D_{EH} \rightarrow \mathcal{P}(\mathcal{P}(\Sigma)) \\ &(\hat{q}_{EH}, \hat{q}_{Must}) \mapsto \{T \subseteq \gamma_{Must}(\hat{q}_{Must}), \forall b \in Blocks, \exists \sigma \in T / q(b) \leq \hat{q}_{EH}(b), \text{where } q = update(q_0, \sigma)\} \end{aligned}$$

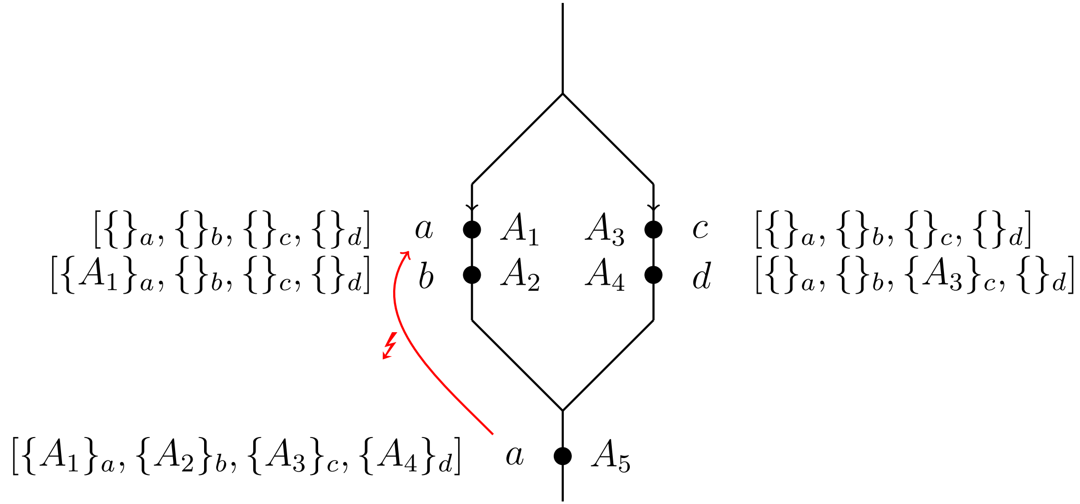



Figure 5.6 – Loading access location

### 5.3.2 Our vulnerability analysis

#### An illustrative example

Our analysis aims at finding program points at which one may distinguish two execution paths from the hit/miss to the instruction cache. Figure 5.6 illustrates such a case. In this example, we consider a 2-ways fully-associative cache where $a$,$b$,$c$ and $d$ are the instruction memory blocks. At the execution point $A_5$ the access to instructions of block $a$ is a hit when the left branch is previously executed and a miss when the execution comes from the right branch. Access $A_5$ is thus a *Definitely Unknown* access and a potential vulnerability. Then, our analysis aims at identifying the previous access that make $A_5$ a hit. Thus, our analysis keeps the previous access localization ($A_n$) for each block. On Figure 5.6, this previous accesses is given in brackets. For instance, before program point $A_5$ the previous access at block $a$ is at point $A_1$.

In summary, the access to $a$ at point $A_5$ is identified as *Definitely Unknown* and the analysis of the previous accesses leading to a hit at $A_5$ tells us that $A_1$ is the only loading location possible. With this information a developer may identify that the branch condition value may leak due to cache instruction.

The *Definitely Unknown* analysis is defined in Section 4.1. We thus focus on identifying where are the previous accesses in case the current access is a hit.

#### Our vulnerability analysis

As mentioned above, our analysis tries to locate the last access(es) that loaded a block, leading to a potential vulnerability.

**Definition 105.** *This last access to a block $b \in Blocks$ in a sequence $\sigma \in \Sigma$ is formally defined by:*

$$LastAccess(b, \sigma) = \{\sigma_i, blk(\sigma_i) = b \wedge \forall j > i, blk(\sigma_j) \neq b\}$$

*The set $LastAccess(b, \sigma)$ is either a singleton containing the last access to $b$ in the sequence $\sigma$, or the empty set in case the sequence $\sigma$ does not contain any access to $b$.*

Because of the potentially high number of blocks a program might contain, tracking the last access for each block might be expensive. Instead, one can restrict the blocks to track to those that

are still cached. However, because it is not always statically known whether a block is still in the cache or not, we propose one over-approximation and one under-approximation of last accesses of blocks.

**Over-approximating analysis** The first step to over-approximate the set of last access locations a block might have been loaded is to over-approximate the set of blocks that might have been loaded. This is what the May analysis achieve: any block in the may cache might have been loaded previously. The second step is then for each of these blocks to over-approximate the set of locations it might have been loaded from.

**Definition 106.** *The domain of our analysis is then the product* $\overline{D} = D_{May} \times (Blocks \rightarrow \mathcal{P}(Access))$*, and the concretization* $\overline{\gamma} : \overline{D} \rightarrow \mathcal{P}(\Sigma)$ *is defined by:*

$$\begin{aligned}\overline{\gamma}(\hat{q}_{May}, \overline{q}) = \{\sigma \in \Sigma, \forall b \in Blocks, & q(b) \geq \hat{q}_{May}(b) \wedge \\ & q(b) < k \Rightarrow LastAccess(b, \sigma) \subseteq \overline{q}(b) \\ & \text{, where } q = update(q_0, \sigma)\}\end{aligned}$$

**Definition 107.** *The update function* $\overline{update}$ *tracks for each block the last access to it, until one can ensure the block is not cached anymore.*

$$\begin{aligned}\overline{update}((\hat{q}_{May}, \overline{q}), A) &= (\hat{q}'_{May}, \overline{q}') \\ \text{where } \hat{q}'_{May} &= update_{May}(\hat{q}_{May}, blk(A)) \\ \text{and } \forall b, \overline{q}'(b) &= \begin{cases} \{A\} & \text{if } b = blk(A) \\ \{\} & \text{if } \hat{q}'_{May}(b) = k \\ \overline{q}(b) & \text{otherwise} \end{cases}\end{aligned}$$

The join operator used in this new domain is simply the may join operator for the may part, and the set union for the last access tracking part.

Figure 5.7 gives an example of how our analysis behaves. The may cache states are shown on the left of the figure, whereas the potential last accesses to $a$ are shown on the right. Once accessed, $a$ always remains in the may cache, the potential accesses to $a$ is thus never reset. When reaching a new access to $a$, the analysis shows that the previous last access (if there was a previous access at all) is the access $A_1$. One can thus deduce that the if/then/else that can leaks data is the first one, and not the second.

**Under-approximating analysis** In addition to the above over-approximating analysis, we propose another analysis that computes safe under-approximation of the locations of accesses that might lead to vulnerabilities. Similarly to its over-approximating counterpart, it is based on an existing analysis that tracks the cache content. This analysis we rely on is the Existing-Hit analysis, that is able to guarantee that some accesses are hits for at least one path.

**Definition 108.** *The domain for the under-approximation is thus the product* $\underline{D} = D_{EH} \times (Blocks \rightarrow \mathcal{P}(Access))$*, and the associated concretization* $\underline{\gamma} : \underline{D} \rightarrow \mathcal{P}(\Sigma)$ *is defined by:*

$$\begin{aligned}\underline{\gamma}(\hat{q}_{EH}, \underline{q}) = \{T \subseteq \Sigma, \forall b \in Blocks, \forall A \in Access, \exists \sigma \in T / & q(b) \leq \hat{q}_{EH}(b) \wedge \\ & (A \in \underline{q}(b) \Rightarrow LastAccess(b, \sigma) = \{A\}) \\ & \text{, where } q = update(q_0, \sigma)\}\end{aligned}$$

*Intuitively, this concretization gives the following guarantees:*

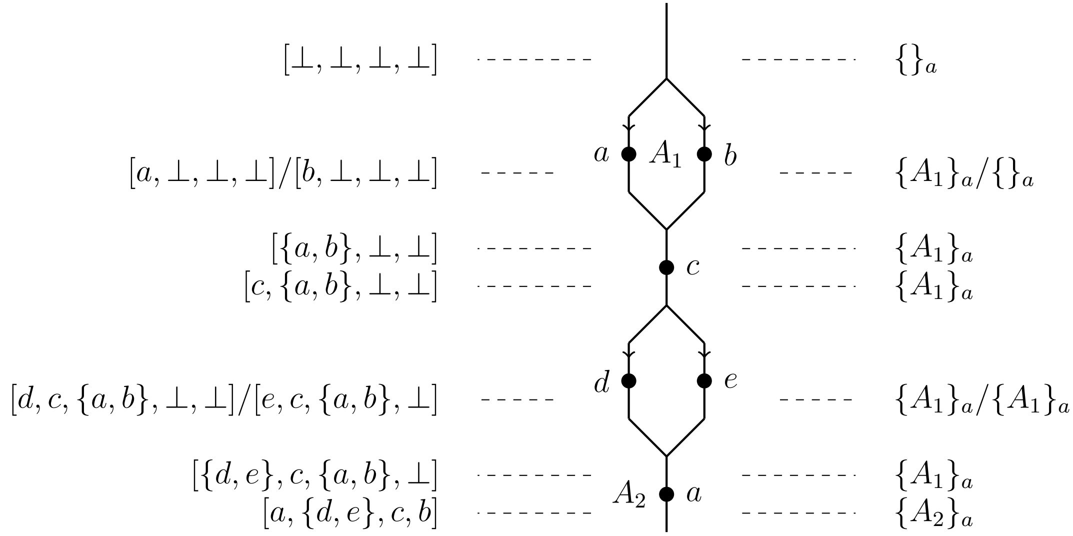



Figure 5.7 – Over-approximating analysis

- *For any set of traces in the concretization and any block, there is at least one trace that leads to an age younger in the concrete state than predicted by the Exist-Hit analysis. Thus, if the Exist-Hit part of the analysis predicts the existence of a hit, there is indeed a concrete trace that leads to a hit.*
- *Moreover, in this case where the Exist-Hit analysis predicts a hit, for any access $A$ in $\underline{q}(b)$ there is such a trace leading to hit where $A$ is the last access to $b$ before the hit.*

**Definition 109.** *The update function for this under-approximating analysis is then defined by:*

$$\begin{aligned}\underline{update}((\hat{q}_{EH}, \underline{q}), A) &= (\hat{q}'_{EH}, \underline{q}')\\ \textit{where } \hat{q}'_{EH} &= update_{EH}(\hat{q}_{EH}, blk(A))\\ \textit{and } \forall b, \underline{q}'(b) &= \begin{cases}\{A\} & \textit{if } b = blk(A)\\ \{\} & \textit{if } \hat{q}'_{EH}(b) = k\\ \underline{q}(b) & \textit{otherwise}\end{cases}\end{aligned}$$

**Definition 110.** *We define the following join operator for the under-approximating analysis:*

$$\begin{aligned}(q_{EH1}, \underline{q}_1) \sqcup (q_{EH2}, \underline{q}_2) &= (q_{EH1} \sqcup_{EH} q_{EH2}, \underline{q}')\\ \textit{where } \forall b \in Blocks, \underline{q}'(b) &= \begin{cases}\underline{q}_1(b) & \textit{if } q_{EH1}(b) < q_{EH2}(b)\\ \underline{q}_2(b) & \textit{if } q_{EH1}(b) > q_{EH2}(b)\\ \underline{q}_1(b) \cup \underline{q}_2(b) & \textit{if } q_{EH1}(b) = q_{EH2}(b)\end{cases}\end{aligned}$$

One can notice that this join operator does not simply compute the union of last accessed sets. Indeed, this would be incorrect. The *Existing-Hit* analysis keeps the best upper-bound possible on blocks age (i.e. the lowest upper-bound). Several accesses after a join, one of the joined upper-bound might reach the associativity $k$, and the corresponding under-approximated set of accesses should be discarded. However, because the *Exist-Hit* join keep the minimum of available upper-bound, this discarding operation might be delayed, leading to an invalid set of last accesses. In other words, the Exist-Hit analysis join might overwrite the upper-bound associated to a set

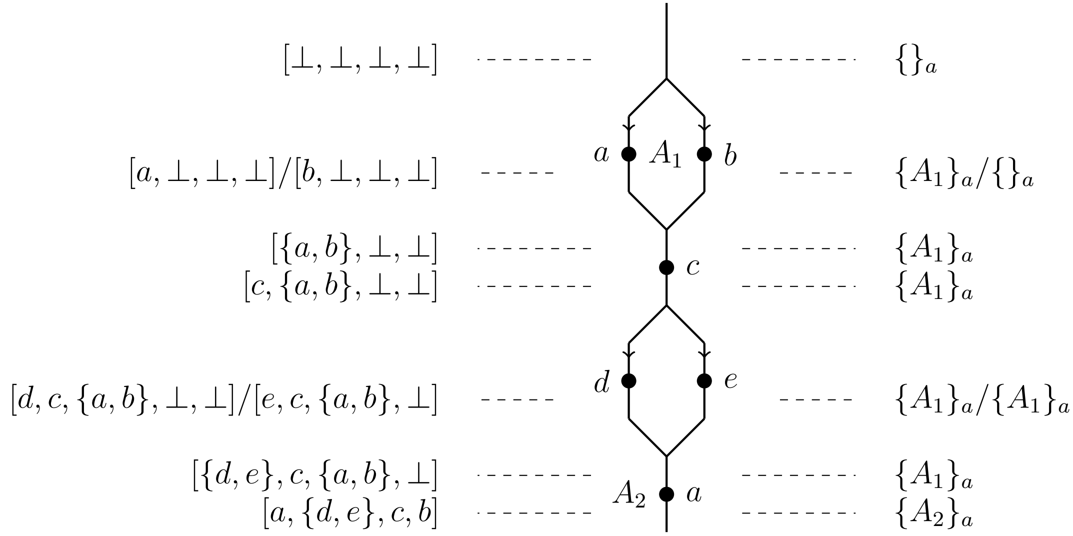



Figure 5.8 – Under-approximating analysis

of accesses, and one should keep this set of accesses only if the associated upper-bound is the minimal one, hence the above definition.

Figure 5.8 shows an example of under-approximation. By performing the analysis described in this section, one is able to tell that there exists at least one path on which the last access to $a$ is hit, and that $A_1$ is one of the accesses where $a$ could have been loaded.

### 5.3.3 Analyses soundness

This section contains all the proofs of soundness of the described analyses.

**Lemma 111.** *The $\overline{update}$ function is safe, i.e. it conserves over-approximations:*

$$\forall (\hat{q}_{May}, \overline{q}) \in \overline{D}, \forall A \in Access, \{\sigma.A, \sigma \in \overline{\gamma}(\hat{q}_{May}, \overline{q})\} \subseteq \overline{\gamma}(\overline{update}((\hat{q}_{May}, \overline{q}), A))$$

*Proof.* Let $(\hat{q}_{May}, \overline{q})$ in $\overline{D}$, $A$ in $Access$ and $\sigma$ in $\overline{\gamma}(\hat{q}_{May}, \overline{q})$. We want to show that $\sigma.A \in \overline{\gamma}(\overline{update}((\hat{q}_{May}, \overline{q}), A))$. This is equivalent to show that, for any $b$ in $Blocks$, $q'(b) \geq \hat{q}'_{May}(b)$ and $q'(b) < k \Rightarrow LastAccess(b, \sigma.A) \subseteq \overline{q}'(b)$, where $q' = update(q, A), q = update(q_0, \sigma)$ and $\hat{q}'_{May} = update_{May}(\hat{q}_{May}, b)$.

The first part $q'(b) \geq \hat{q}'_{May}(b)$, immediately follows from the correctness of the May analysis. We thus focus on the second part, the implication $q'(b) < k \Rightarrow LastAccess(b, \sigma.A) \subseteq \overline{q}'(b)$. We know from the definition of $\sigma$ that $q(b) < k \Rightarrow LastAccess(b, \sigma.A) \subseteq \overline{q}'(b)$.

We then proceed by cases distinction:

- If $q'(b) = k$, then the implication is trivially true because its left-hand side is false.

- If $q'(b) < k$ and $b = blk(A)$, then: $LastAccess(b, \sigma.A) = \{A\}$. Moreover $\overline{q}'(b) = \overline{update}(\overline{q}, A)(b) = \{A\}$. Thus $LastAccess(b, \sigma.A) \subseteq \overline{q}'(b)$, proving that the implication holds.

- Finally, if $q'(b) < k$ and $b \neq blk(A)$, we have: Necessarily, $q(b) < k$ because $q'(b) < k$ and $b \neq blk(A)$. Thus, $LastAccess(b, \sigma) \subseteq \overline{q}(b)$. Moreover $\hat{q}'_{May}(b) \leq q'(b) < k$, and thus $\overline{q}'(b) = \overline{q}(b)$. Thus, $LastAccess(b, \sigma.A) = LastAccess(b, \sigma) \subseteq \overline{q}(b) = \overline{q}'(b)$, proving the desired implication.

□

**Lemma 112.** *The join operator (point-wise union of sets)* $\overline{\sqcup}$ *is safe, i.e.*

$$\forall((q_{May1},\overline{q}_1),(q_{May2},\overline{q}_2)) \in \overline{D}^2, \overline{\gamma}(q_{May1},\overline{q}_1) \cup \overline{\gamma}(q_{May2},\overline{q}_2) \subseteq \overline{\gamma}(q_{May1} \sqcup_{May} q_{May2}, \overline{q}_1 \overline{\sqcup} \overline{q}_2)$$

*Proof.* Let $((q_{May1},\overline{q}_1),(q_{May2},\overline{q}_2))$ in $\overline{D}^2$. We show (without loss of generality), that $\overline{\gamma}(q_{May1},\overline{q}_1) \subseteq \overline{\gamma}(q_{May1} \sqcup_{May} q_{May2}, \overline{q}_1 \overline{\sqcup} \overline{q}_2)$.

Let $\sigma \in \overline{\gamma}(q_{May1},\overline{q}_1)$ and $b \in Blocks$.

We have $q(b) \geq q_{May1}(b)$ and $q(b) < k \Rightarrow LastAccess(b,\sigma) \subseteq \overline{q}_1(b)$, where $q = update(q_0,\sigma)$. We want to prove that $q(b) \geq (q_{May1} \sqcup_{May} q_{May2})(b)$ and $q(b) < k \Rightarrow LastAccess(b,\sigma) \subseteq \overline{q}_1(b) \cup \overline{q}_2(b)$.

The first part follows from the soundness of the may analysis, whereas the second part trivially holds, by the inclusion $\overline{q}_1(b) \subseteq \overline{q}_1(b) \cup \overline{q}_2(b)$. □

**Theorem 113.** *For any trace $\sigma$ reaching an access $A$, $\sigma$ belongs to the concretization of the abstract value $(\hat{q}_{May},\overline{q})$ associated to $A$.*

*Proof.* This is a direct consequence of the two preceding lemmas applied to the fixpoint equation of the concrete and abstract collecting semantics. □

**Lemma 114.** *The $\underline{update}$ function is safe, i.e. it conserves the last access sets under-approximations:*

$$\forall(\hat{q}_{EH},\underline{q}) \in \underline{D}, \forall A \in Access, \{\{\sigma.A, \sigma \in T\}, T \in \underline{\gamma}(\hat{q}_{EH},\underline{q})\} \subseteq \underline{\gamma}(\underline{update}((\hat{q}_{EH},\underline{q}),A))$$

*Proof.* Let $(\hat{q}_{EH},\underline{q})$ in $\underline{D}$, $A$ in $Access$ and $T$ in $\underline{\gamma}(\hat{q}_{EH},\underline{q})$.

We note $T' = \{\sigma.A, \sigma \in T\}$ and want to show that $T' \in \underline{\gamma}(\hat{q}'_{EH},\underline{q}')\}$, where $(\hat{q}'_{EH},\underline{q}') = \underline{update}(\hat{q}_{EH},\underline{q}),A)$. Let $b \in Blocks$ and $A' \in Access$. We then want to show that there exists $\sigma'$ in $T'$ such that:

- $q'(b) \leq \hat{q}'_{EH}(b)$
- $A' \in \underline{q}'(b) \Rightarrow LastAccess(b,\sigma') = \{A'\}$

where $q' = update(q_0,\sigma')$.

Let $\sigma \in T$ such that $q(b) \leq \hat{q}_{EH}(b)$ (with $q = update(q_0,\sigma)$) and $A' \in \underline{q}(b) \Rightarrow LastAccess(b,\sigma) = \{A'\}$. Such $\sigma$ exists by definition of $T$.

Our candidate $\sigma'$ is then $\sigma.A$.

$q'(b) \leq \hat{q}'_{EH}(b)$ is ensured by the correctness of the Exist-Hit analysis.

- If $b = blk(A)$, then $\underline{q}'(b) = \{A\}$. Thus, $A' \in \underline{q}'(b)$ implies $A' = A$ and $LastAccess(b,\sigma') = LastAccess(b,\sigma.A) = \{A'\}$
- Else, $b \neq blk(A)$. Suppose $A' \in \underline{q}'(b)$, $\underline{q}'(b) \neq \{\}$ and $b \neq blk(A)$, thus $\underline{q}'(b) = \underline{q}(b)$. We then have $A' \in \underline{q}(b)$, and $LastAccess(b,\sigma) = \{A'\}$. Finally, $LastAccess(b,\sigma.A) = LastAccess(b,\sigma) = \{A'\}$ as desired.

□

**Lemma 115.** *The join operator* $\underline{\sqcup}$ *is safe, i.e.*

$$\forall((q_{EH1},\underline{q}_1),(q_{EH2},\underline{q}_2)) \in \underline{D}^2, \forall T_1 \in \underline{\gamma}(q_{EH1},\underline{q}_1), \forall T_2 \in \underline{\gamma}(q_{EH2},\underline{q}_2), T_1 \cup T_2 \in \underline{\gamma}(q_{EH1} \sqcup_{EH} q_{EH2}, \underline{q}_1 \underline{\sqcup} \underline{q}_2)$$

*Proof.* Let $((q_{EH1}, \underline{q}_1), (q_{EH2}, \underline{q}_2))$ in $\underline{D}^2$, $T_1$ in $\underline{\gamma}(q_{EH1}, \underline{q}_1)$ amd $T_2$ in $\underline{\gamma}(q_{EH2}, \underline{q}_2)$.

We note $T_3 = T_1 \cup T_2$, and we want to show that $T_3 \in \underline{\gamma}(q_{EH3}, \underline{q}_3)$, where $q_{EH3} = q_{EH1} \sqcup_{EH} q_{EH2}$ and $\underline{q}_3 = \underline{q}_1 \underline{\sqcup} \underline{q}_2$.

Let $b \in Blocks$ and $A \in Access$.

We then proceed by case distinction:

- If $q_{EH1}(b) = q_{EH2}(b)$, then: $q_{EH3}(b) = q_{EH1}(b) = q_{EH2}(b)$ and $\underline{q}_3(b) = \underline{q}_1(b) \cup \underline{q}_2(b)$. Suppose $A \in \underline{q}_3(b)$. We assume, without loss of generality, that $A \in \underline{q}_1(b)$. Then, there exist $\sigma \in T_1$ such that:
  - $q(b) \leq q_{EH1}(b)$
  - $LastAccess(b, \sigma) = \{A\}$

  , where $q = update(q_0, \sigma)$.

  Because $\sigma \in T_3$, and by equality of $q_{EH1}(b)$ and $q_{EH3}(b)$, we have $T_3 \in \underline{\gamma}(q_{EH3}, \underline{q}_3)$
- If $q_{EH1}(b) \neq q_{EH2}(b)$, we suppose without loss of generality that $q_{EH1}(b) < q_{EH2}(b)$. Then: $q_{EH3}(b) = q_{EH1}(b)$ and $\underline{q}_3(b) = \underline{q}_1(b)$. Thus, $A \in \underline{q}_3(b) \Rightarrow A \in \underline{q}_1(b)$ and the remaining of the proof is identical to the preceding case.

□

**Theorem 116.** *Let $T$ denote the set of traces reaching an access $A$, then $T$ belongs to the concretization of the abstract value $(\hat{q}_{EH}, \underline{q})$ associated to access $A$.*

*Proof.* This is a direct consequence of the two preceding lemmas applied to the fixpoint equation of the concrete and abstract collecting semantics. □

## Conclusion

Some vulnerabilities in program are due to the ability of an attacker to distinguish between cache hits and cache misses. Our *Definitely Unknown* analysis can be used to spot some memory accesses potentially leaking information in this setting. However, deducing what information is leaking from the results of the *Definitely Unknown* analysis is difficult. We thus designed two analyses to help a developer locating the information leak. These analyses approximate the set of accesses that lead to the classification as *Definitely Unknown* analysis.

Usually, side channels leakage are prevented by relying on constant programming, i.e. by forbidding any dependency between memory access or control dependency, and secret data. The compliance to the security policy can then be checked by automatic tools. Our approach is a first step toward the weakening the these constraints. Indeed, instead of assuming that all dependency between control and secret data lead to vulnerabilities, we propose a finer grain approach to spot vulnerabilities more precisely. An interesting approach would be to provide the results of our analysis to the automatic tools checking the compliance to the security policy, so that it knows some potential leakage are spurious.

# Chapter 6

# Conclusion

Modern processors use cache memories to speed up computation. By storing frequently accessed memory blocks into a fast memory closer to the core than the main memory, caches reduce the average delay to retrieve information from memory. However, by supplying a second (or more) area to retrieve memory blocks, one introduces a variability in the delay of memory accesses. In some cases, this variability is undesirable. This is the case, for instance, in the domain of critical real-time systems which require to be predictable to pass certification. Another example is the domain of security, where this variability can be used by an attacker to obtain secret information from the program execution. In both cases, one is interested in statically obtaining information about the program behavior relatively to the cache. This is what cache analyses achieve.

In this thesis, we focus mainly on cache analyses that classify memory accesses into the one that result in cache hits, and the one that result in cache misses. We focus on the analysis of L1 instruction caches using the Least Recently Used policy. More precisely, our goal is to classify memory accesses into three categories:

- accesses that always result in a cache hit.
- accesses that always result in a cache miss.
- accesses that can result in a hit or a miss depending on the execution path.

Knowing whether a given access results in a hit or a miss being undecidable in general, we assume that all paths in the program CFG are feasible. By doing so, we overapproximate the set of possible program behavior and are thus guaranteed to obtain a safe classification of memory accesses.

As a first contribution, this thesis explores the complexity of problems linked to cache analysis under this assumption. More precisely, we look at the theoretical complexity of finding path leading to a hit (or a miss) depending on the cache replacement policy. We show that LRU leads to NP-hard classification problems, while other common replacement policies, namely PLRU, FIFO and NMRU, lead to PSPACE-hard classification problems. This tends to confirm the idea that LRU is the easiest replacement policy to analyze.

In addition to this assumption, some existing analyses sometimes classify a memory access as *Unknown*, meaning it can belong to any of the three categories mentioned. More precisely, the Must (respectively May) analysis respectively computes a safe under-approximation of the set of accesses always resulting in a hit (respectively a miss). Thus, after performing these analyses, one does not have any information about the accesses that do not belong to these under-approximating set of accesses. Our second contribution is thus an analysis called *Definitely Unknown* that, similarly to May and Must analyses, provides a safe under-approximation of the set of accesses that can result both in a hit or a miss. By using this analysis, one thus reduce the uncertainty about

the accesses that remained unclassified after the May and Must analyses. The *Definitely Unknown* analysis asserts the existence of paths leading to a hit and a miss, and that the classification as *Unknown* by the May and Must analyses did not come from a coarse approximation. Our experiments show that using this analysis, we are able to classify 98% of the remaining accesses as *Definitely Unknown*.

Our third contribution of this thesis is an exact analysis (relatively to the model where all paths are feasible), that is able to classify all the remaining *Unknown* accesses into one of the *Always-Hit*, *Always-Miss* or *Definitely Unknown* category, by using a model-checker. This approach consists in encoding the program and the cache in a model-checker, and to encode the possibility of a hit or a miss into a logic formula. The novelty of this approach is that it relies on a technique called *block focusing* that is able to abstract cache state relatively to a given memory block. By doing so, the cache state representation obtained is smaller because it does not keep information about the blocks accessed before the block we are interested in. The resulting set of blocks that are younger than the block we focus on is called the *younger set* associated to the focus block. However, this abstraction is still able to provide the exact age of the block we focus on, and enables us to exactly classify it among the three categories *Always-Hit*, *Always-Miss* and *Definitely Unknown*. Due to this precision improvement, this exact analysis increases the number of blocks classified as *Always-Hit* or *Always-Miss* to 18.2% (where the May and Must analyses alone only classify 16.8% of the accesses).

Our next contribution is an improved exact analysis also based on block focusing, that performs additional pruning of the state space by removing some younger sets that are not minimal or maximal. Indeed, one can show that one do not lose precision by removing these younger sets, and that the resulting analysis still provides an exact classification. To compactly represent the antichain of maximal (or minimal) younger set and efficiently prune them when needed, we propose an implementation relying on Zero-suppressed Decision Diagrams. By doing so, we obtain an analysis that is one order of magnitude faster (19.5 times faster in average, and up to 300 times faster) than the model-checking approach, and that is comparable to the May and Must approaches while providing an exact classification of accesses.

This thesis also investigates two applications of our exact analysis, one in the framework of WCET computation, the other one in the domain of security. In the case of WCET computation, we study the relation between cache and pipeline analyses. Indeed, a pipeline analysis relying on a coarse classification of memory accesses would have to treat both hit and miss cases, resulting in the pipeline state space blow-up. Our experiments show that coupling our exact cache analysis in the OTAWA pipeline analysis can result in a reduction by 70% of the global analysis time (cache+WCET), corresponding to a gain of 10 hours for some benchmark. However, providing an exact classification to the pipeline analysis does not seem to highly impact WCET estimation itself. First, most of the unknown accesses are refined as definitely unknown. In these cases, both cases must still be considered. It is also unlikely that refining *Unknown* accesses to *Always-Miss* leads to better WCET. This only occurs in case of timing anomalies. Finally, the access refined that actually lead to a shorter execution time of a basic block might not be located on the critical path. In the domain of security, we propose to use the *Definitely Unknown* analysis to spot some accesses potentially leaking information. In addition, to ease the task of confirming or denying the existence of a leak, we provide two cache analyses that track the set of possible loading location of the block accessed at the leaking point (that would indicate the executed path). One of them provides a safe over-approximation of the loading location based on the May analysis, the other one provides a safe under-approximation based on the Exist-Hit analysis.

To conclude, this thesis explores the feasibility of optimally precise cache analyses under the hypothesis that all program paths are feasible. While the classification problems are shown to

have high complexity in theory, we provide analyses that are efficient in practice for LRU L1 instruction caches. One can thus hope that analyzing other kind of caches (data caches, other replacement policies, etc.) or micro-architectural optimizations (pipeline, branch predictors, etc.) can be done efficiently on practical examples.

## 6.1 Future Work

This section gives some ideas that could be used to improve the current state of our analyses. Mainly, three possibilities are considered:

- Partially take into account the program semantics to avoid covering some infeasible paths that would lead to precision loss.
- Extend our analyses to cover data caches. Depending on the writing policy, conserving the exactness of our approach can be challenging.
- Extend our work to other replacement policies. Because no abstraction are currently known to perform exact analyses, we propose two approaches: either drop the exactness requirement for efficiency and use abstractions that can lose precision, or stick to the concrete semantics using hash consing (as done when using the ZDD-based approach) to compactly represent set of states.

### 6.1.1 Program Semantics

This thesis extensively uses the assumption that all execution paths in the CFG are feasible. However, this is not the case in many real programs. This section describes some ideas that could be used to partially fill the gap between the real program containing infeasible paths and its ideal model where all paths are feasible.

#### Encoding infeasible paths in the Model-Checker

First, we propose an attempt[1] to encode some infeasible paths into the model checker. More precisely, we focus on encoding infeasible paths that result from mutually exclusive tests, and that do not contain cycles.

Figure 6.1a is an example of program containing an infeasible path: the second condition can not be true if the first is false. This impossible path is highlighted in red on the program CFG represented in Figure 6.1b. Assuming that all paths are feasible, a cache analysis would classify the second access to $b$ as *Definitely Unknown*[2], whereas this access always leads to a miss on real execution. Indeed, the only path leading to a hit is the infeasible path $BB_0 - BB_1 - BB_3 - BB_4 - BB_5$.

For the sake of simplicity, we only consider one memory access per basic block. We then note $state(x)$ the predicate which is true when the current basic block being executed is $x$, and $miss(y)$ a predicate which is true when the current cache state does not contain the block $y$. Using these notations, the LTL formula used to check that the access to $b$ always misses in basic block $BB_4$ of our example is the following:

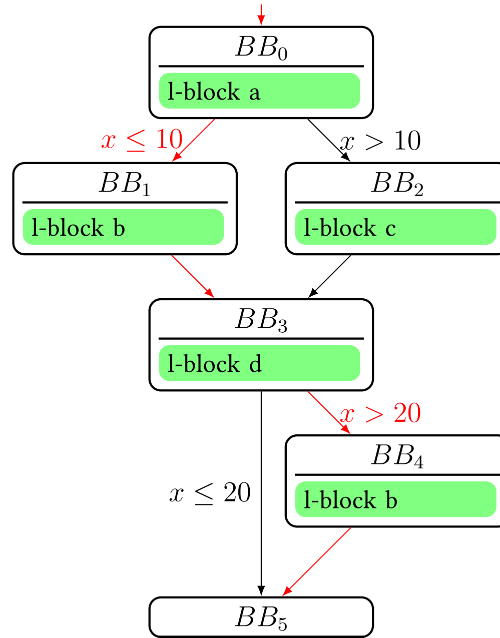
$$A[G(state(BB_4) \Rightarrow miss(b))]$$

[1] This work was done in collaboration with Florian Barrois.

[2] assuming a cache of associativity greater than 2.

```
int x = rand();
if(x > 10) {
        ... // x unchanged
}
else {
        ... // x unchanged
}
if(x > 20) {
        ... // x unchanged
}
```

(a) Source code




(b) Control Flow Graph

Figure 6.1 – Example of program containing infeasible path

This formula can be read as follows: "For any path $\pi$, and any position $i$ in $\pi$, being in basic block $BB_4$ implies that block $b$ is not cached". In our example, this formula is obviously false due to the infeasible path.

To model the fact that this path is infeasible, we add some constraints about the path. In our example, the infeasible path can be modelled by the subformula $F(state(BB_1) \wedge F(state(BB_4)))$ which can be read as "$\pi$ goes through $BB_1$ and, from there, goes through $BB_4$".

The modified formula given to the model checker is thus the following:

$$[\underbrace{G(state(BB_4) \Rightarrow miss(b))}_{\text{Always-Miss}} \vee \underbrace{(F(state(BB_1) \wedge F(state(BB_4))))}_{\text{infeasible path}}]$$

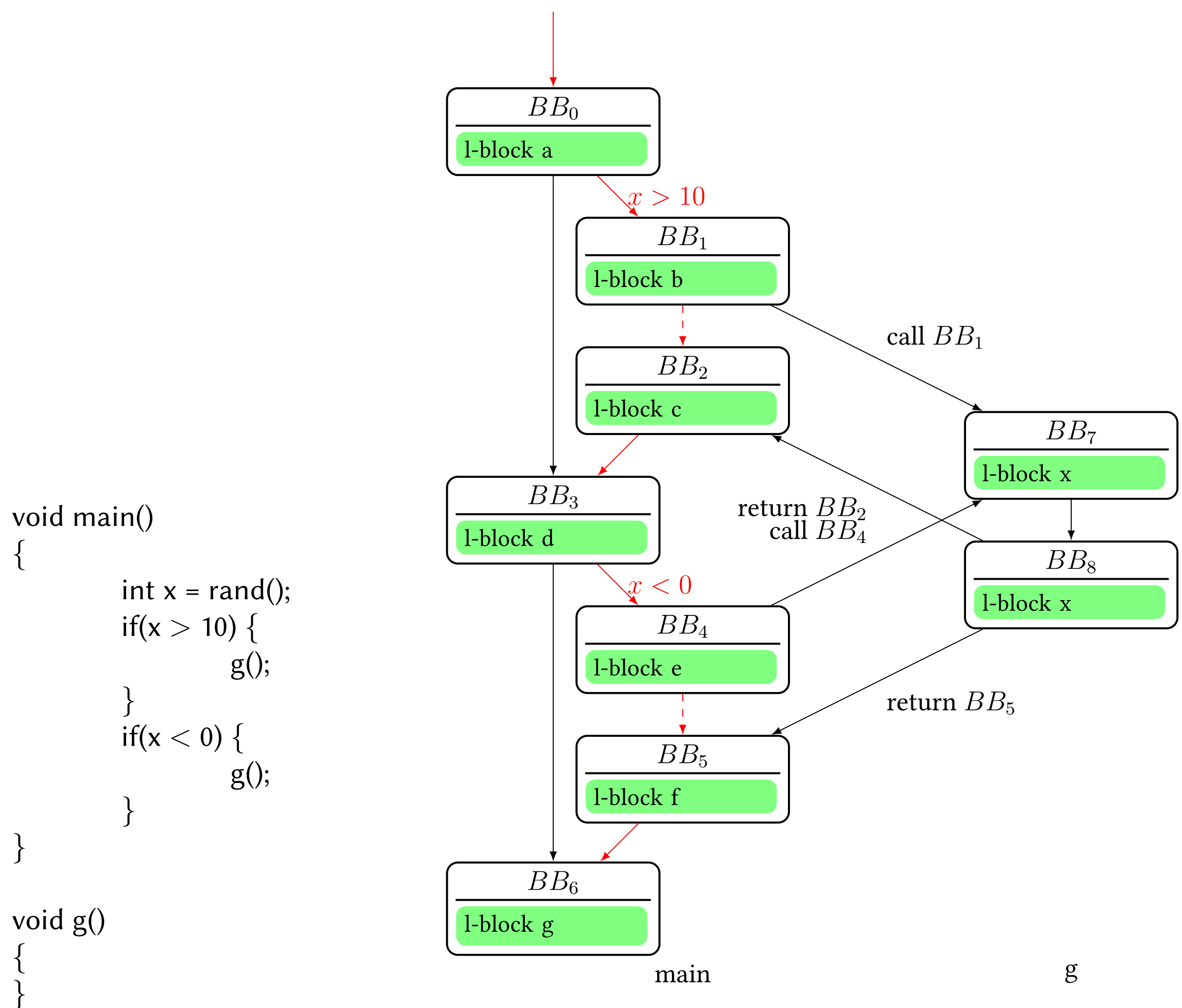
This formula can be read as "For any path $\pi$, either $\pi$ leads to a miss at $BB_4$, or $\pi$ is not feasible". For example 6.1b, the model-checker asserts that this formula is always true. This methods can be extended to any set of infeasible paths as soon as they can be unambiguously expressed by a LTL formula. However, finding such a formula can be difficult in some situation. For instance, we do not have any automatic method to encode mutually exclusive path nested in loops.

**Treating function calls** In addition to mutually exclusive conditions, treating function calls is interesting, because they can be the source of infeasible paths. Gathering all functions of a program in a single CFG can be done in several ways:

- One way to gather the CFGs of all functions consists in inlining function call. Every time a function is called, the CFG of the called function is duplicated and several edges are added to connect the calling block in the callee to the entry block of the caller, and from the exit block of the callee to the return block of the caller. This method is simple and does not create spurious paths in the program. However, it can not handle recursive functions and duplicating several times the same functions can lead to a huge CFG.

- Another possibility is to use one single copy of each function. As previously, edges are added between calling block and entry blocks, and exit points and return blocks. This method creates a more compact graph, but has the drawback of adding spurious path in the program model. Indeed, the program flow can go from a caller to a callee, reach the callee exit and return to another potential caller instead of the original one. In the following, we show how these spurious paths can be removed in the case of non-recursive function.

Consider source code of Listing 6.2a, which corresponding CFG is given on Figure 6.2b. Function main is formed by basic blocks $BB_0$ to $BB_6$, whereas f is formed by basic blocks $BB_7$ and $BB_8$. Dashed arrows represent function calls and associate a call site to its return block. These arrows cannot be taken, and do not even exist in the real implementation of the program model. As previously, the red path symbolizes an infeasible path.




(a) Source code (b) Control Flow Graph

Figure 6.2 – Example of program containing infeasible path

Because of the exclusive conditions, g can be called only once. Thus, the access to block $x$ in basic block $BB_7$ is always a miss. However, the spurious path $BB_0 - BB_3 - BB_4 - BB_7 - BB_8 - BB_2 - BB_3 - BB_4 - BB_7 - BB_8 - BB_5 - BB_6$ leads to a hit to block $x$ on the second

execution of basic block $BB_7$. To avoid this problem, we force paths going through a call site to exit the callee by the associated return block. In our example, if one goes from basic block $BB_4$ to basic block $BB_7$, it is not possible to enter again the same function until return point $BB_5$ is met. One thus obtain the following formula:

$$G(\underbrace{(state(BB_4) \wedge X(state(BB_7)))}_{\text{function call}} \Rightarrow X(X(\underbrace{(\neg state(BB_7))U\,state(BB_5)}_{\text{forbids entering the same function again}})))$$

The double "next" operator is necessary to delay the restriction to enter the function after it is entered the first time. This constraint can be generalized to any function call:

$$G(\underbrace{(state(\text{call_site}) \wedge X(state(\text{callee_entry})))}_{\text{function call}}$$
$$\Rightarrow X(X(\underbrace{(\neg state(\text{callee_entry}))U\,state(\text{return_point})}_{\text{forbids entering the same function again}})))$$

One such formula can be added for every calling edge in the program. In our example, the formula checked to assert $x$ is Always-Miss in basic block $BB_7$ is the following:

$$\begin{aligned}
&A[G(state(BB_7) \Rightarrow miss(x)) \\
&\quad \vee F(state(BB_1) \wedge F(state(BB_4))) \\
&\quad \vee \neg G((state(BB_1) \wedge X(state(BB_7))) \Rightarrow X(X((\neg state(BB_7))U\,state(BB_2)))) \\
&\quad \vee \neg G((state(BB_4) \wedge X(state(BB_7))) \Rightarrow X(X((\neg state(BB_7))U\,state(BB_5))))]
\end{aligned}$$

Deciding if a path is feasible is an undecidable problem. Thus, one can not find all infeasible paths in a program and encode them in the model checker. However, the encoding above allows to systematically treat functions calls. Moreover, it is expressive enough to encode infeasible paths that can be described by a sequence of basic blocks they go through (this however requires that the basic blocks are not repeated). One can thus hope to improve the cache analysis precision using these methods. However, the range of situation that can be handled by this approach is currently limited. Indeed, the problem of recursive function calls[3] and mutually exclusive tests nested in loops are still open questions.

### Improving efficiency of the abstract interpretation cache analyses

In addition to the model checking method explained above, this section describes some methods that can be used to improve the precision of the analysis presented in Section 4.3. Mainly, we describe some abstract interpretation techniques that can be use to prune the set of traces covered by an abstract interpretation analyzer.

**Trace partitioning** *Trace Partitioning* [MR05] is a method to improve the precision of abstract interpretation analyses. Usually, analyses by abstract interpretation compute one abstract value for each basic block of a program. This abstract value summarizes all the program traces reaching the associated basic block. The main idea of trace partitioning is to compute several abstract values instead of one, each of them summarizing a set of program traces. By splitting (or more precisely, partitioning) the set of traces into several parts, one hopes to obtain a more precise

[3]Note that recursive function calls are usually not a problem in critical systems, because developpers avoid them and compilers try to remove them.

value for each subset. Of course, the precision versus cost ratio highly depends on how the set of traces is partitioned.

A common refinement consists in constructing the partition of program traces based on the control flow. For instance, one uses an abstract value to describe the program traces going through the left branch of a test and a second abstract value to describe the program traces going through the right branch. Similarly, one can use this method to distinguish the first iteration of a loop from the others. To improve efficiency, set of traces can be merged later one during the analysis. For example, one could decide that distinguishing between the two branches of a test is not relevant one or two basic blocks after the join location. The efficiency and precision of the analysis is then highly dependent on the heuristics used to merge partitions. For instance, in the case of cache analyses, one could use the predictability metrics described in [Rei09] as heuristics. In the case of LRU caches, two concrete cache states converge after $k$ different accesses. One could thus wait $k$ different accesses after the join location to merge the associated set of traces.

**Context-sensitive abstraction** As for the model-checking approach, there are different ways to manage function calls when performing abstract interpretation. The first method simply consists in analyzing a function with the correct input value each time a function call is met. This is roughly equivalent to inline the function by duplicating its body. If a function is called twice, it is then analyzed twice, with a potentially different abstract value at the basic block entry. An other possibility is to merge all the functions into one CFG as done for building the model checker program model in Section 6.1.1, and to analyze it as a single function. This is the current approach used in our implementation in OTAWA . This then creates some infeasible paths from some call sites to wrong return points. Similarly to the model checker approach, one would like to cut these impossible flows. One possibility to do this is to use an abstraction of the call-stack in addition the cache abstraction. Given an abstract domain $D^{\#}$ and the corresponding concretization $\gamma$, one can build a new domain $D^{\#}_{Context} = Context \rightarrow D^{\#}$ and concretization $\gamma^{\#}_{Context}$ by associating one abstract value in $D^{\#}$ for every possible calling context. The concretization is then defined by:

$$\gamma^{\#}_{Context}(f) = \bigcup_{c \in Context} \{\gamma^{\#}(f(c)) \cap \gamma_{Context}(c)\}$$

where $\gamma_{Context}(c)$ is the set of traces that are compatible with the context $c$. This is a form of trace-partitioning. One possible choice for $Context$ is to use a call-string. This consists in stacking tokens associated to call site when a function call is met, and to unstack it when the associated return point is reached. The concretization $\gamma_{Context}(c)$ is then the set of traces that can lead to the context $c$. Note that the formulation above allows unbounded call string, which leads to the existence of infinite ascending chain in the domain $D^{\#}_{Context}$. One usual solution to ensure the analysis termination is thus to bound the length of the call string.

Finally, one last solution[4] can be considered to treat function calls, that consists in building a summary for each function. The idea is to analyze each function only once[5] to build a summary of its behavior. This summary is then used when a call to the associated function is met. Using the value abstracting the cache state at the entry of the function and the summary of the function, one computes an abstraction of the output cache state without analyzing the function again.

In the case of our exact analysis, this approach is worth considering. In what follow we focus on the case of the Must exact analysis. Given a block $a$ to focus on, one can build the summary of a function by analyzing it starting from an empty younger set. First consider the case of Figure 6.3

[4] This approach is the result of a joint work with Jan Reineke.

[5] This assumes that the analyzed function are not recursively called.

where $a$ is not accessed in the summarized function. Then, the final abstract value obtained represents the set of maximal sets of blocks that the function can access. For instance, if one obtain the abstract $\{\{b\}, \{c, d\}\}$, we know that for any paths in the function, the set of blocks met is a subset of $\{b\}$ or $\{c, d\}$. This summary can then be combined with an abstract value representing the cache state at the entry of the function. For example, if the cache state is represented by the set $\{\{d\}, \{c, e\}\}$, after returning from the function, the cache state can be abstracted by $\{\{b, d\}, \{b, c, e\}, \{c, d, e\}\}$. From the point of view of the ZDDs, the composition of an abstract value with the summary is done by computing the dot product of the two ZDDs and pruning by removing non-maximal elements.

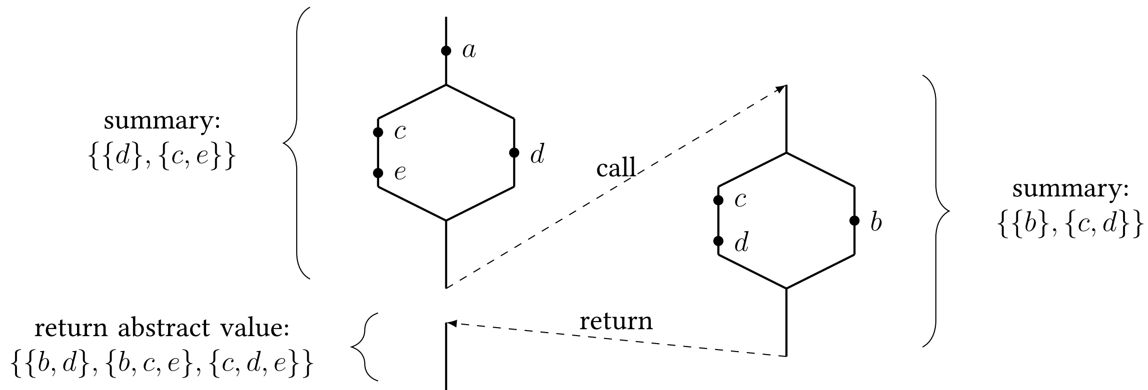



Figure 6.3 – Composing function summary

Consider now the case where $a$ can be accessed in the function to summarize. All the paths from one access to $a$ to the end of the function can be abstracted as usual using a ZDD. However, this ZDD should not be composed with the input value of the function. Indeed, all the blocks at the entry of the function where accessed before executing the function and, thus, before accessing $a$. As a consequence, one has to maintain two different abstract values when summarizing a function. One is used as long as no $a$ is encountered and summarize all paths where $a$ is not met. The other one is used to represent paths where an accessed to $a$ has been met. On a function call, the first one is composed with the input value by performing a dot product, and the result is joined as usual with the second one.

### 6.1.2 Analyzing Data Caches

In addition to consider the program semantics to improve our analyses precision, it could be interesting to cover the cases of data caches. As mentioned in Section 2.1, analyzing data caches can implies running additional preliminary analyses than analyzing instruction caches, for two reasons:

- First, the addresses of memory blocks accessed are less predictable than addresses of instructions, which are mainly executed in sequence. Contrarily, data memory accesses can be much more complex, and require additional analyses, such as pointer aliasing or memory shape, to allow precise cache analysis.
- Moreover, whereas instructions are only mostly read after the program is loaded, data caches manage write accesses too. Analyzing them thus requires to take the write policy into account. This section investigates how the analyses proposed in this thesis can be adapted to treat different write policies in the case of L1 data caches.

**Write allocate/No-write allocate** Write allocate policies are easy to analyze. Indeed, write accesses load/refresh memory blocks when accessed as read accesses. Any cache abstract domain can thus be used without any modification, as reads and writes are handled identically.

On the other hand, no-write allocate policies require some modifications of the abstract domains to be analyzed. In case of a write miss, the data are written to the backing store and the cache is not modified. In case of a write hit, the cache content is not modified (there is no need to evict a block in case of hit), but the position of block might change, and other metadata are updated. Thus, in case one can not ensure that the block is cached (or not) when accessed, both outcomes should be taken into account. One way to handle this is to update the cache state (assuming a hit) and join the result with its previous value (representing the miss case). However, this can lead to precision loss. For instance, one loses the exactness of the block focusing based analyses by doing so. Consider the younger set $\{x, y\}$ when focusing on block $a$, and a write access to $z$. The younger set abstraction does not allow to distinguish a hit and miss. Indeed, $\{x, y\}$ can be concretize to $[x, y, a, z]$ which results in a hit, or to $[x, y, a, w]$ (with $w \neq z$) which result in a miss.

Note that the no-write allocate caches are easy to handle in case of a FIFO replacement policy. In case of a write-miss, the access is taken into account at the baking store level, and the cache content is left unmodified. In case of a write-hit, the cache is updated according to the FIFO policy, and is thus not modified. In both cases, the cache is left unmodified. Thus, one can simply ignore the write-accesses in case of FIFO replacement policy.

The distinction of write-back and write-through caches is not relevant when analyzing the *content* of L1 caches. Indeed, it only impacts the order of accesses to the baking store. However, knowing when the dirty blocks are evicted might be interesting when the cache is analyzed together with the pipeline, which might benefit from the knowledge of cache delay. Similarly, information about dirty bits is useful when analyzing higher level caches.

### 6.1.3 Reducing analysis cost

**Simultaneous computation** We have explained our exact analyses for classifying accesses to each address $a$ separately. It is also possible to simultaneously classify all addresses together, by updating the abstractions (e.g. $C^{\min}_{l,a}$) for all $a$ all together when updating the abstract state at location $l$. By performing the analyses in parallel, one can hope that sharing of ZDDs between the analyses of different blocks would reduce the memory consumption and analysis time. However, one should be careful when sharing the cache containing the results of ZDDs operations, because two update transformers (one focus on a block $a$, and the other focused on a block $b$) applied to the same ZDD and block might give different results. Moreover, when this happens, and assuming that caching is done with caution, both results should be cached, increasing the probability to evict a useful result. Thus, the problem of the efficiency of a parallel implementation is still an open question.

**On-demand backward analysis** We have presented our exact analysis in a forward fashion: to classify hits and misses to $a$, we compute at each location the collection of the set of addresses found along path $\sigma$ for all paths $\sigma$ from the nearest preceding occurrences of $a$ (truncated at length $k$). We could formulate our analysis in a backward fashion: given a specific location $A$ in the cache conflict graph, we compute at each access $A'$ the collection of the set of addresses found along path $\sigma$ for all paths $\sigma$ from $A'$ to $A$. This computation stops at other edges labeled with $a$, start vertices, or when computing the special value $\dashv$. Then, an access $A$ to block $a$ may result in a miss if and only if at least one value $\dashv$ was reached during this backward propagation, and it

may result in a hit if and only if at least one value different ⊣ was reached during this backward propagation. This backward analysis could then be called on demand when reaching an access a coarse forward analysis is not able to classify. By starting the analysis from the problematic access, we avoid analyzing all the remaining part of the code. In addition, by analyzing in the program in the opposite direction, one can hope that the backward analysis will not meet the join that caused the precision loss of the forward analysis, keeping the number of younger sets to consider low.

### 6.1.4 Other replacement policies

This thesis focuses on the analyses of the LRU replacement policy. In this section we explore the possibilities of designing precise analyses for other replacement policies. Contrarily to the LRU case, which analysis benefits from block focusing abstraction, we are not aware of any abstraction that would permit to forget some information about the cache state but still lead to an exact classification. In the following, we look at some other analyses that be used to classify accesses in the presence of a non-LRU replacement policy.

**An abstract interpretation analysis for PLRU** A PLRU cache state can be formally described by function that map a block to its position in the cache similarly to LRU cache state. As for the LRU case, we use $k$ for blocks that are not in the cache. In addition to the description of cache line, describing a PLRU cache state requires to represent the associated tree bits. However, by flipping any bit in this tree and swapping the two associated subtrees, one obtain an equivalent cache state (i.e. a cache state that behaves exactly the same in terms of hit and miss for any access sequence). For example, the PLRU cache states on Figure 6.4 are equivalent because one can be obtained from the other one by swapping the innermost right subtree. To represent the equivalent class

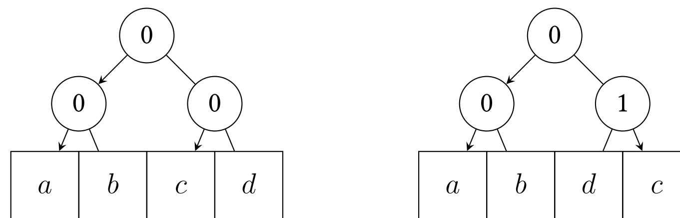



Figure 6.4 – Two equivalent PLRU cache state

of a PLRU cache state, we choose the only cache state which tree bits are all set to 0 (like the left cache state of Figure 6.4). This choice is arbitrary, but ensure the uniqueness of the representing cache state (see [GR10]). Then, cache line 0 contains the next block to evict, whereas cache line $k-1$ contains the most recently accessed block. A possible concrete domain for describing a PLRU cache state is thus $D_{PLRU} = Blocks \to \{0, \ldots, k\}$.

Accessing a cache line might flip some tree bits of the current cache state. One then need to swap subtrees to go back to an equivalent cache state where all tree bits are set to 0. When operating on cache states where all tree bits are 0, an access can thus be seen as a permutation of cache lines. We note $\pi_i$ the permutation induced by a hit on line $i$. The concrete update for PLRU

caches can thus be formalized as follows:

$$\begin{aligned}
update : D_{PLRU} \times Blocks &\to D_{PLRU} \\
(q, b) &\mapsto q'
\end{aligned}$$

$$\text{where } \forall b' \in Blocks, q'(b') = \begin{cases} k-1 & \text{if } b = b' \\ \pi_{q(b)}(q(b')) & \text{if } b \neq b' \wedge q(b) < \text{k} \\ 0 & \text{if } b \neq b' \wedge q(b) = k \wedge q(b') = 0 \\ \pi_0(q(b')) & \text{if } b \neq b' \wedge q(b) = k \wedge q(b') \neq 0 \end{cases}$$

These three cases correspond to the following behaviors:

- When a block is accessed, all the tree bits on its path are set to 1. It thus takes position $k-1$ in the canonical cache state representation.
- When the access results in a hit (i.e. $q(b) < k$), all blocks are permuted according to position of the accessed block.
- When the access results in a miss (i.e. $q(b) = k$), the block at position 0 is evicted. All blocks are permuted according to position of the accessed block.
- All other blocks are permuted exactly as if the evicted block was accessed.

In the following approach[6], we abstract a set of concrete cache state by a function that, for every memory block, associates the set of position it can occupy. This defines the abstract domain $D^{\#}_{PLRU} = Blocks \to \mathcal{P}(\{0, \ldots, k\})$ and the associated concretization function:

$$\forall q^{\#} \in D^{\#}_{PLRU}, \gamma_{PLRU}(q^{\#}) = \{q \in D_{PLRU}, \forall b \in Blocks, q(b) \in q^{\#}(b)\}$$

One can then define the abstract transformer $update^{\#}$ that modifies an abstract value $q^{\#}$ when accessing $b$ as follows:

- For all position $i$ in $q^{\#}(b)$, suppose that $b$ is in position $i$.
- Find all the position a block $b'$ can take when $b$ is accessed and is in $i$.
- Merge all the obtained set for all value of $i$. This defines the new value $q^{\#\prime}(b')$.

The main advantage of this abstract transformer is that the potential positions of a block can be encoded as a vector of bits, and that the permutation $\pi_i$ involved can be viewed as a matrix multiplication applied to these vector of bits. Finally, merging abstract cache state can be done by performing a bitwise logical or of bitvectors. Moreover, some reduction can be used to improve the precision of this analysis. For example, when a $b$ is temporary assumed to have position $i$, $i$ can be removed from the set of available position of all other blocks. By removing it, some other blocks might end up with a single available position, which is then forced. This new forced position can in turn be removed from the set of available positions for other blocks, and so on.

```
type block = int
type plru_tree = Node of plru_tree * plru_tree
               | Leaf of block
```

Figure 6.5 – Representing PLRU concrete cache

**Exact analyses for other replacement policies** Although we are not aware of any abstraction that would lead to an exact analysis for other replacement policies, one can investigate the usage of hash consing methods similar to ZDD to compactly represent sets of concrete cache states. The remaining of this section describes how concrete cache states for some replacement policies can be represented in a way that enable information sharing between similar concrete caches.

In the case of PLRU, a concrete cache state is represented as previously by its canonical cache state. One can then encode the cache state as a tree, where information about tree bits is removed. Listing 6.5 represents a possible implementation of this structure.

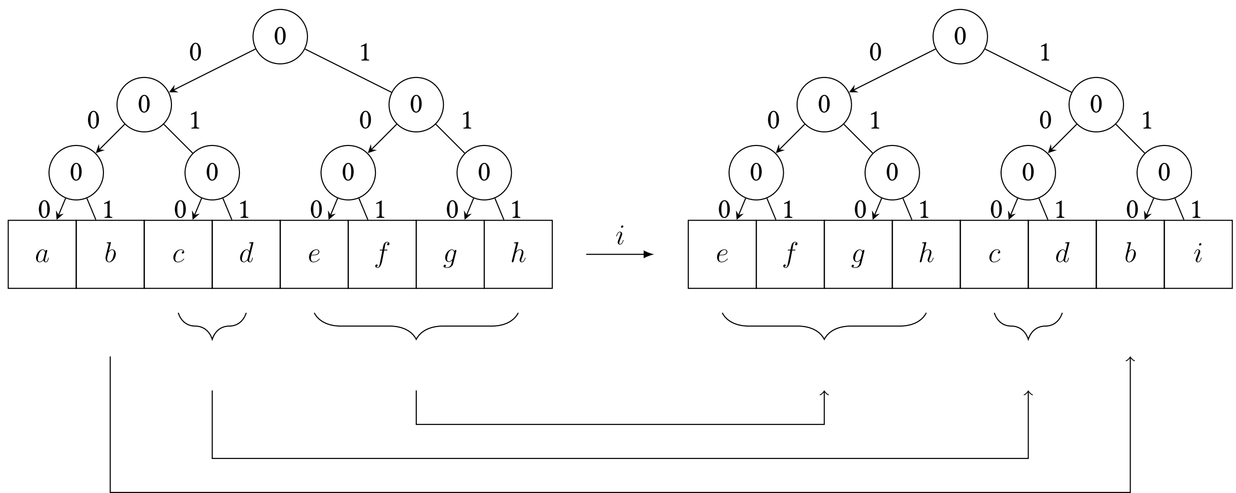



Figure 6.6 – PLRU subtree sharing

From this implementation, the hash-consing implementation is straight forward: one simply allows only one copy of every subtree. Consider the PLRU cache state of Figure 6.6. After accessing block $i$, one obtain the cache state on the right (after swapping subtrees to set all tree bits to 0), which shares three separated subtrees with the original cache state.

Updating a concrete cache state is then done recursively as shown on Listing 6.7. A more efficient implementation would benefit from hash consing by caching the result of functions *contains* and *update*.

**First-In First-Out policy** Similarly to the PLRU replacement policy, FIFO cache states can be represented by a recursive data structure that only is modified partially when a block is accessed. The main idea is to implement the concrete cache state as a table of cache lines, and a pointer indicating which of the cache line will be evicted next. On a cache hit, the structure is not modified as imposed by the FIFO policy. On a cache miss, the line given by the pointer is replaced by the accessed block, and other cache lines are not modified. One possibility to optimize the sharing

[6]This work was done in collaboration with Zhenyu Bai.

```
let rec contains cache block = match cache with
        Node (left, right) −> (contains left block) || (contains right block)
      | Leaf b −> b = block

let rec update cache accessed = match cache with
        Node (left, right) −>
              if (contains right accessed) then
                     Node(left, (update right accessed))
              else
                     Node(right, (update left accessed))
      | Leaf b −> Leaf accessed
```

Figure 6.7 – Updating PLRU concrete cache

of cache lines between cache states consists in implementing the table of cache lines as a tree, as shown on Figure 6.8.

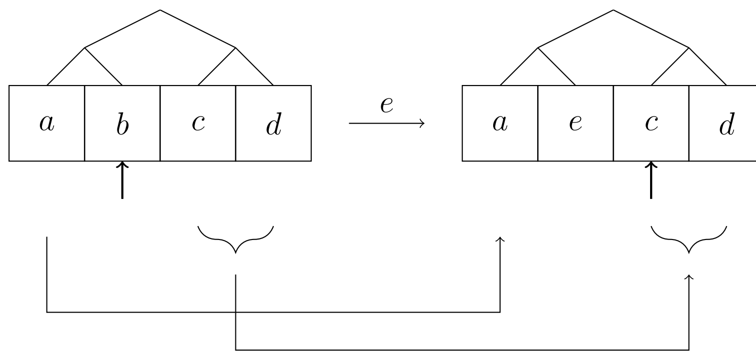



Figure 6.8 – FIFO subtree sharing

Again, by sharing the identical subtrees between cache states, one can hope to represent all the concrete cache states reaching a program point in a compact way. Moreover, as in the PLRU case, doing so enables the possibility of caching the result of contains, update and join functions.

One problem needs to be solved to implement these analyses, which is the representation of unknown values. Indeed, if the entry cache state is unknown, one would need to represent all possible combination of $k$ memory blocks. This is of course not possible in practice for program of reasonable size. One thus need a way to represent unknown cache states and/or partially unknown cache states.

# Open Question

Due to the always increasing need for performance, modern processors incorporate many hardware optimizations. However, this growing complexity of the hardware ressources tend to make the analysis of programs running on it harder. As a result, ensuring safety and security properties of these programs is more and more challenging. At one point, one will have to answer the following question: what safety/security properties are we willing to lose to gain performance?

# Bibliography


[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach.* Cambridge University Press, 2009.

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996.

[AMM04] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In Seong-Moo Yoo and Letha H. Etzkorn, editors, *Proceedings of the 42nd Annual Southeast Regional Conference, 2004, Huntsville, Alabama, USA, April 2-3, 2004*, pages 267–272. ACM, 2004.

[AMR10] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. In Jaejin Lee and Bruce R. Childers, editors, *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, LCTES 2010, Stockholm, Sweden, April 13-15, 2010*, pages 153–162. ACM, 2010.

[AR14] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 141–142. IEEE Computer Society, 2014.

[BC08] 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.

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES, 2005.

[Ber06] Christoph Berg. PLRU cache domino effects. In Frank Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC77] 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.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CLS06] Anne Canteaut, Cédric Lauradoux, and André Seznec. Understanding cache attacks. Technical Report 5881, INRIA, April 2006.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'état ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, March 1978.

[CR11] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, pages 193–203. IEEE Computer Society, 2011.

[CR18] Sudipta Chattopadhyay and Abhik Roychoudhury. Symbolic verification of cache side-channel freedom. *CoRR*, abs/1807.04701, 2018.

[CS08] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008*, pages 51–65, 2008.

[DKMR15] 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):1–32, 2015.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[EH83] E. Allen Emerson and Joseph Y. Halpern. ”sometimes” and ”not never” revisited: On branching versus linear time. In John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum, editors, *Conference Record of the Tenth Annual ACM Symposium*

*on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 127–140. ACM Press, 1983.

[FAH+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASICS*, pages 2:1–2:10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[GLBD14] David Griffin, Benjamin Lesage, Alan Burns, and Robert I. Davis. Lossy compression for worst-case execution time analysis of PLRU caches. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS'14, Versaille, France, October 8-10, 2014*, page 203. ACM, 2014.

[GLYY14] 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.

[GR09] Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In 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*, pages 120–136. Springer, 2009.

[GR10] 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.

[GYLY13] Nan Guan, Xinping 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.

[HJR11] 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.

[HLTW03a] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[HLTW03b] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition.* Morgan Kaufmann, 2012.

[Jr.78] Sheldon B. Akers Jr. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[KCB+15] N. Kurd, M. Chowdhury, E. Burton, T. P. Thomas, C. Mozak, B. Boswell, P. Mosalikanti, M. Neidengard, A. Deval, A. Khanna, N. Chowdhury, R. Rajwar, T. M. Wilson, and R. Kumar. Haswell: A family of ia 22 nm processors. *IEEE Journal of Solid-State Circuits*, 50(1):49–58, Jan 2015.

[Knu11] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, part 1*, volume 4A. Pearson, 2011.

[KSWH00] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *J. Comput. Secur.*, 8(2,3):141–158, August 2000.

[LGY+10] Mingsong Lv, Nan Guan, Wang Yi, Qingxu Deng, and Ge Yu. Efficient instruction cache analysis with model checking. *RTAS, work-in-progress session*, 2010.

[LS99] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium (RTSS)*, 1999.

[Lun02] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories.* PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2002.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 1998.

[McM93] Kenneth L. McMillan. *Symbolic model checking.* Kluwer, 1993.

[Min93] Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In Alfred E. Dunlop, editor, *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993.*, pages 272–277. ACM Press, 1993.

[Min01] Shin-ichi Minato. Zero-suppressed bdds and their applications. *Int. J. on Software Tools for Technology Transfer (STTT)*, 3(2):156–170, 2001.

[Mis14] Alan Mishchenko. An introduction to zero-suppressed binary decision diagrams. In Tsutomu Sasao and Jon T. Butler, editors, *Applications of Zero-Suppressed Decision Diagrams.* Morgan Claypool, 2014.

[MPH94] Adam Malamy, Rajiv N. Patel, and Norman M. Hayes. Methods and apparatus for implementing a pseudo-lru cache memory replacement scheme with a locking feature. US patent 5,353,425, US Patent Office, October 1994.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[R+06] Jan Reineke et al. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2006.

[Rei09] Jan Reineke. *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, Saarland University, 2009.

[Rei18] Jan Reineke. The semantic foundations and a landscape of cache-persistence analyses. *LITES*, 5(1):03:1–03:52, 2018.

[RGBW07a] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[RGBW07b] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[RS09] Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. *Trans. HiPEAC*, 2:222–241, 2009.

[RWT+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.

[SB11] Fabio Somenzi and Aaron R. Bradley. IC3: where monolithic and incremental meet. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 3–8. FMCAD Inc., 2011.

[Som01] Fabio Somenzi. Efficient manipulation of decision diagrams. *Int J. Software Tools for Technology Transfer (STTT)*, 3(2):171–181, 2001.

[TAMV19] Sudarshan Tsb, Rahil Abbas Mir, and S Vijayalakshmi. Highly efficient lru implementations for high associativity cache memory. 03 2019.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[ZGJ+17] Wei Zhang, Fan Gong, Lei Ju, Nan Guan, and Zhiping Jia. Scope-aware useful cache block analysis for data cache related preemption delay. In Gabriel Parmer, editor, *2017 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburg, PA, USA, April 18-21, 2017*, pages 63–74. IEEE Computer Society, 2017.

# Nomenclature

$\alpha$ — An Abstraction function, page 20

$Blocks$ — The set of memory blocks accessed in a program, page 29

$\overline{\sqcup}$ — The join operator of the over-approximating loading location analysis, page 114

$\underline{\sqcup}$ — The join operator of the under-approximating loading location analysis, page 114

$D_{abs}$ — An Abstract Domain, page 20

$D_{conc}$ — A Concrete Domain, page 18

$D_{EH}$ — The Exist-Hit abstract domain, page 72

$D_{EM}$ — The Exist-Miss abstract domain, page 72

$D_{LRU}$ — The concrete domain of LRU valid cache states, page 29

$D_{May}$ — The May analysis abstract domain, page 34

$D_{Must}$ — The Must analysis abstract domain, page 32

$\overline{D}$ — The domain of over-approximating loading location analysis, page 112

$\underline{D}$ — The domain of under-approximating loading location analysis, page 112

$D_{PLRU}$ — The concrete domain of PLRU valid cache states, page 126

$\alpha_{\odot}$ — The abstraction function from concrete domain to focus semantics domain, page 84

$D_{\odot}$ — The focus semantics domain, page 84

$update_{\odot}$ — The abstract update of the focus semantics, page 84

$\overline{\gamma}$ — The concretization function from the domain of over-approximating loading location analysis to the trace domain, page 112

$\underline{\gamma}$ — The concretization function from the domain of under-approximating loading location analysis to the trace domain, page 112

$\sqcup_{EH}$ — The Exist-Hit join operator, page 72

$\sqcup_{EM}$ — The Exist-Miss join operator, page 72

$\pi$ — A path in a Control-Flow Graph, page 18

$\hat{q}_{EH}$ A Exist-Hit abstract cache state, page 72

$\hat{q}_{EM}$ A Exist-Miss abstract cache state, page 72

$\hat{q}_{May}$ A May abstract cache state, page 34

$\hat{q}_{Must}$ A Must abstract cache state, page 32

$\overline{q}$ An abstract value in $\overline{D}$, page 112

$\underline{q}$ An abstract value in $\underline{D}$, page 112

$update$ The concrete update transformer, page 29

$\overline{update}$ The update transformer of the over-approximating loading location analysis, page 112

$\underline{update}$ The update transformer of the under-approximating loading location analysis, page 112

$E$ The set of edges of a Control Flow Graph, page 18

$F$ The concrete collecting semantics, page 19

$f_v$ The concrete transformer associated to basic block $v$., page 18

$G$ A Control Flow Graph, page 18

$V$ The set of basic blocks of the program, page 18

# Chapitre 7

# Résumé en français

Les centrales électriques, les trains, les satellites de communications ou les engins spatiaux sont autant d'exemples de systèmes temps-réel critiques. En effet, une défaillance d'un de ces systèmes peut entraîner des pertes de vies humaines ou, dans une moindre mesure, un coût économique important. En d'autres termes, les systèmes temps-réels critiques sont les systèmes qui doivent impérativement respecter les échéances qui leurs sont imposées car leur éventuelle défaillance aurait des conséquences importantes. Par exemple, une tâche qui consisterait à lire périodiquement la valeur de capteurs pour corriger la trajectoire d'un avion est un système critique temps-réel : si la tâche possède une période d'une milliseconde, alors elle doit s'exécuter un moins d'une milliseconde.

Toutefois, à cause des débits grandissants de données à traiter, les processeurs modernes emploient de nombreuses optimisations matérielles qui rendent difficile la prédiction du temps d'exécution exact d'un programme. En particulier, l'utilisation de mémoires caches accélère certains calculs en présence d'un mémoire DRAM, en gardant les instructions et données fréquemment accédées dans une mémoire rapide proche du processeur. Ainsi, la latence d'une instruction impliquant seulement le CPU ne subit pas de pénalité d'accès à la mémoire (dont la latence est plus élevée de plusieurs ordres de grandeur). La latence d'un accès mémoire peut donc varier significativement en présence de caches, selon qu'il en résulte un *cache hit* (l'information cherchée se trouve déjà dans le cache) ou un *cache miss* (l'information se trouve en mémoire centrale seulement). Assurer qu'un programme termine avant l'échéance imposée est donc plus compliqué en présence de caches.

En pratique, les analyses statiques permettant de borner le temps d'exécution d'un programme doivent prendre les éventuels caches en considération pour ne pas fournir une borne trop pessimiste. L'objectif des analyses de cache est donc de classer les accés mémoires dans une des trois catégories suivantes :

1. *Always Hit* : cet accès mémoire résulte toujours en un cache hit ;
2. *Always Miss* : cet accès mémoire résulte toujours en un cache miss ;
3. cet accès peut résulter en un hit ou un miss, selon le chemin d'exécution emprunté.

La question de l'appartenance d'un accès à l'une de ces catégorie est en générale indécidable ; et les analyses impliquées ont donc recours à des abstractions, et peuvent classer certains accès comme "unknown". Plus précisément, le problème de l'indécidabilité est souvent résolu en supposant que tous les chemins d'exécution du programme sont possibles. Si construire un programme ne vérifiant pas cette hypothèse est trivial (par exemple, un ajoutant du code mort), le modèle du

programme obtenu (qui suppose tous les chemins comme possibles) couvre toutes les comportements possibles du programme. Une analyse qui traite toutes les exécutions décrites par le modèle traite donc toutes les exécutions du véritable programme.

Toutefois, malgré cette hypothèse que “tous les chemins d’exécutions sont possibles”, le problème de classement des accès mémoires reste difficiles, et les analyses efficaces utilisent davante d’abstractions.

Dans cette thèse, nous étudions l’impact de cette hypothèse sur les analyses de caches. Plus précisément :

- Nous étudions la complexité algorithmique d’une analyse de cache théorique optimalement précise sous cette hypothèse. En effet, si cette hypothèse rend le problème décidable, certains caches sont plus difficiles à analyser que d’autres. Nous analysons donc la difficulté intrinsèque de différentes politiques de remplacement (l’algorithme qui choisit quel bloc évincer du cache pour pouvoir en stocker d’autres) sous cette hypothèse, du point de vue de la théorie de la complexité.

- Comme mentionné précédemment, certaines analyses sont capables de garantir que certains accès mènent toujours à un hit ou à un miss. Toutefois, aucune analyse à notre connaissance n’est capable de classer des accès mémoires dans la catégorie 3 ci-dessus.

- Finalement, nous nous intéressons à l’efficacité d’une analyse de cache optimale dans l’hypothèse que tous les chemins d’exécution sont faisables. En pratique, les binaires analysés sont des programmes industriels dont les flots de contrôle sont restreints, et les caches ont une associativité relativement faibles. Il est donc intéressant d’observer l’efficacité d’une telle analyse dans des cas réels d’utilisation.

### Contributions

**Complexité d’une analyse de précision optimale** Tandis que l’intuition dicte qu’un cache retient les mots accédés autant sa taille le permet, la réalité est plus complexe : le fonctionnement du cache dépend du nombre de niveaux de caches, de la taille de chacun d’entre eux, de leur nombre de voies (également appelé associativité) et de leur politique de remplacement. L’analyse de cache utilisée dépend donc de la politique de remplacement du cache et on peut remarquer une nette préférence pour la politique LRU (*Least Recently Used*) dans la littérature, notamment grâce à l’analyse par interprétation abstraite de Ferdinand [AFMW96] et à ses variations. À l’inverse, certaines politiques comme PLRU (*Pseudo-LRU*), NMRU (*Not Most Recently Used*) ou FIFO (*First-In, First-Out*) ont la réputation d’être difficiles à analyser [HLTW03b] et peu prédictibles [RGBW07b]. Il est donc légitime de se demander si ces politiques de remplacement sont intrinsèquement difficiles à analyser, ou si la recherche n’a pas encore aboutie à des algorithmes efficaces. En effet, la difficulté d’analyser des politiques de caches différente n’est pas liée à l’efficacité de ces politiques de remplacement. L’analyse statique s’intéresse au comportant en pire-cas du programme, et des politiques de comportement similaires en moyenne[1] peuvent être très différentes l’une de l’autre du point de vue de l’analyse de caches. Bien que PLRU et NMRU soient par conception des politiques inspirés de LRU (mais de faible coût d’implémentation) et que leurs efficacités soient comparables [AMM04], elles sont très différentes du point de vue

[1] ici nous ne référons pas à une moyenne au sens d’une distribution de probabilité, mais plutôt au sens informel de valable sur des exemples industriels pertinents, par opposition à des exemples conçus pour exhiber de très bons ou très mauvais comportements

d'une analyse de pire-cas. Nous évaluons donc la complexité des problèmes de classification des accès mémoire sous différentes politiques de remplacement.

**Accès Mémoires *Definitely Unknown*** Comme mentionné précédemment, les analyses comme *May* et *Must* présentées dans [AFMW96] reposent sur des approximations pour classer des accès comme*Always-Miss* ou *Always-Hit*. Dans cette configuration, les analyses de cache ne donnent aucune informations concernant les autres accès (non classés). En effet, un accès non classé peut être un accès *Always-Hit* ou *Always-Miss* non détecté par l'analyse à cause des approximations effectuées, ou résulter en des hits ou miss selon le chemin d'exécution. Nous introduisons donc la notion d'accès *Definitely Unknown* pour désigner les accès pouvant résulter en un hit ou en un miss selon le contexte d'exécution. Cette distinction entre accès *Unknown* et *Definitely Unknown* permet alors de de charactériser la précision d'une analyse. Les accès *Unknown* peuvent en effet être raffinés en *Always-Hit*, *Alway-Miss* ou *Definitely Unknown* en utilisant une analyse de cache plus précise. En revanche, les accès classés comme *Definitely Unknown* sont une conséquence du programme analysé et de la configuration du cache. Ces accès ne peuvent pas être raffinés en *Always-Hit* ou *Always-Miss*, quelque soit la méthode employée. Nous proposons donc une analyse similaire aux analyses *May* et *Must*, capable d'approximer l'ensemble des accès *Definitely Unknown* pour la politique *LRU* et sous l'hypothèse que tous les chemins sont exécutables.

**Analyse de cache exacte** Finalement, l'incertitude résultant d'une classification imprécise des accès mémoire peut impacter fortement les analyses de WCET (Worst Case Execution Time - temps d'exécution dans le pire cas). Par exemple, il est important de connaître avec précision le comportement du cache lorsque l'on cherche à analyser le comportement d'un processeur superscalaire et/ou avec pipeline. En effet, lorsque l'analyse de cache n'est pas capable de classer un accès mémoire comme *Always-Hit* ou *Always-Miss*, l'analyse de pipeline doit envisager les deux possibilités, menant éventuellement à une explosion du nombre d'état de pipeline à analyser. L'imprécision de l'analyse de cache peut donc avoir deux conséquences distinctes sur l'analyse de WCET : (a) une surestimation du véritable pire temps d'exécution, par exemple en cas de classification trop grossière d'accès qui en réalité mènent à des hits. Un utilisateur peut suspecter ce type de phénomènes lorsque la borne sur le WCET est loin des temps d'exécution obtenus expérimentalement. Ceci peut alors décourager l'utilisateur d'utiliser des outils d'analyse statique. (b) un temps d'analyse excessivement élevé, causé par l'explosion de l'espace d'états. Améliorer la précision des analyses de caches est donc important dans le contexte de l'analyse de WCET, mais ces améliorations doivent être réalisable pour un coût raisonnable. Dans cette thèse, nous proposons deux approches pour éliminer l'incertitude sur la classification des accès mémoire, une basée sur l'usage d'un model checker, l'autre réalisée par interprétation abstraite.

## Abstract

The certification of real-time safety critical programs requires bounding their execution time. Due to the high impact of cache memories on memory access latency, modern Worst-Case Execution Time estimation tools include a cache analysis. The aim of this analysis is to statically predict if memory accesses result in a cache hit or a cache miss. This problem is undecidable in general, thus usual cache analyses perform some abstractions that lead to precision loss. One common assumption made to remove the source of undecidability is that all execution paths in the program are feasible. Making this hypothesis is reasonable because the safety of the analysis is preserved when adding spurious paths to the program model. However, classifying memory accesses as cache hits or misses is still hard in practice under this assumption, and efficient cache analysis usually involve additional approximations, again leading to precision loss. This thesis investigates the possibility of performing an optimally precise cache analysis under the common assumption that all execution paths in the program are feasible.

We formally define the problems of classifying accesses as hits and misses, and prove that they are NP-hard or PSPACE-hard for common replacement policies (LRU, FIFO, NMRU and PLRU). However, if these theoretical complexity results legitimate the use of additional abstraction, they do not preclude the existence of algorithms efficient in practice on industrial workloads.

Because of the abstractions performed for efficiency reasons, cache analyses can usually classify accesses as *Unknown* in addition to *Always-Hit* (Must analysis) or *Always-Miss* (May analysis). Accesses classified as *Unknown* can lead to both a hit or a miss, depending on the program execution path followed. However, it can also be that they belong to one of the *Always-Hit* or *Always-Miss* category and that the cache analysis failed to classify them correctly because of a coarse approximation. We thus designed a new analysis for LRU instruction that is able to soundly classify some accesses into a new category, called *Definitely Unknown*, that represents accesses that can lead to both a hit or a miss. For those accesses, one knows for sure that their classification does not result from a coarse approximation but is a consequence of the program structure and cache configuration. By doing so, we also reduce the set of accesses that are candidate for a refined classification using more powerful and more costly analyses.

Our main contribution is an analysis that can perform an optimally precise analysis of LRU instruction caches. We use a method called *block focusing* that allows an analysis to scale by only analyzing one cache block at a time. We thus take advantage of the low number of candidates for refinement left by our *Definitely Unknown* analysis. This analysis produces an optimal classification of memory accesses at a reasonable cost (a few times the cost of the usual May and Must analyses).

We evaluate the impact of our precise cache analysis on the pipeline analysis. Indeed, when the cache analysis is not able to classify an access as *Always-Hit* or *Always-Miss*, the pipeline analysis must consider both cases. By providing a more precise memory access classification, we thus prune the state space explored by the pipeline analysis and hence the WCET analysis time.

Aside from this application of precise cache analysis to WCET estimation, we investigate the possibility of using the *Definitely Unknown* analysis in the domain of security. Indeed, caches can be used as side-channel to extract some sensitive data from a program execution, and we propose a variation of our *Definitely Unknown* analysis to help a developer finding the source of some information leakage.



## Résumé

Dans le cadre des systèmes critiques, la certification de programmes temps-réel nécessite de borner leur temps d'exécution. Les mémoires caches impactant fortement la latence des accès mémoires, les outils de calcul de pire temps d'exécution incluent des analyses de cache. Ces analyses visent à prédire statiquement si ces accès aboutissent à des cache-hits ou des cache-miss. Ce problème étant indécidable en général, les analyses de caches emploient des abstractions pouvant mener à des pertes de précision. Une hypothèse habituelle pour rendre le problème décidable consiste à supposer que toutes les exécutions du programme sont réalisables. Cette hypothèse est raisonnable car elle ne met pas en cause la validité de l'analyse : tous les véritables chemins d'exécutions du programme sont couverts par l'analyse. Néanmoins, la classification des accès mémoires reste difficile en pratique malgré cette hypothèse, et les analyses de cache efficaces utilisent des approximations supplémentaires. Cette thèse s'intéresse à la possibilité de réaliser des analyses de cache de précision optimale sous l'hypothèse que tous les chemins sont faisables.

Les problèmes de classification d'accès mémoires en hits et miss y sont définis formellement et nous prouvons qu'ils sont NP-difficiles, voire PSPACE-difficiles, pour les politiques de remplacement usuelles (LRU, FIFO, NMRU et PLRU). Toutefois, si ces résultats théoriques justifient l'utilisation d'abstractions supplémentaires, ils n'excluent pas l'existence d'un algorithme efficace en pratique pour des instances courantes dans l'industrie.

Les abstractions usuelles ne permettent pas, en général, de classifier tous les accès mémoires en *Always-Hit* et *Always-Miss*. Certains sont alors classifiés *Unknown* par l'analyse de cache, et peuvent aboutir à des cache-hits comme à des cache-miss selon le chemin d'exécution emprunté. Cependant, il est aussi possible qu'un accès soit classifié comme *Unknown* alors qu'il mène toujours à un hit (ou un miss), à cause d'une approximation trop grossière. Nous proposons donc une nouvelle analyse de cache d'instructions LRU, capable de classifier certains accès comme *Definitely Unknown*, une nouvelle catégorie représentant les accès pouvant mener à un hit ou à un miss. On est alors certain que la classification de ces accès est due au programme et à la configuration du cache, et pas à une approximation peu précise. Par ailleurs, cette analyse réduit le nombre d'accès candidats à une reclassification par des analyses plus précises mais plus coûteuses.

Notre principale contribution est une analyse capable de produire une classification de précision optimale. Celle-ci repose sur une méthode appelée *block focusing* qui permet le passage à l'échelle en analysant les blocs de cache un par un. Nous profitons ainsi de l'analyse *Definitely Unknown*, qui réduit le nombre de candidats à une classification plus précise. Cette analyse précise produit alors une classification optimale pour un coût raisonnable (proche du coût des analyses usuelles May et Must).

Nous étudions également l'impact de notre analyse exacte sur l'analyse de pipeline. En effet, lorsqu'une analyse de cache ne parvient pas à classifier un accès comme *Always-Hit* ou *Always-Miss*, les deux cas (hit et miss) sont envisagés par l'analyse de pipeline. En fournissant une classification plus précise des accès mémoires, nous réduisons donc la taille de l'espace d'états de pipeline exploré, et donc le temps de l'analyse.

Par ailleurs, cette thèse étudie la possibilité d'utiliser l'analyse *Definitely Unknown* dans le domaine de la sécurité. Les mémoires caches peuvent être utilisées comme canaux cachés pour extraire des informations de l'exécution d'un programme. Nous proposons une variante de l'analyse *Definitely Unknown* visant à localiser la source de certaines fuites d'information.