Precise Modelling of Instruction Cache Behaviour

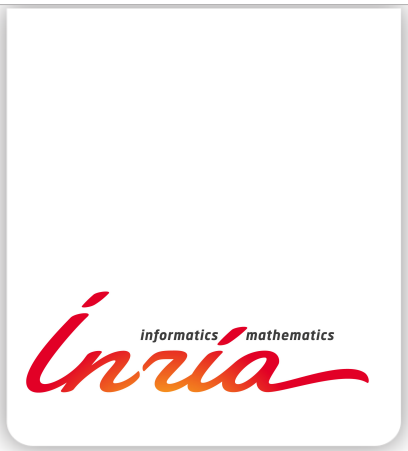
Sidharta Andalam, Roopak Sinha, Partha S. Roop, Alain Girault, Jan Reineke

Precise Modelling of Instruction Cache Behaviour

Sidharta Andalam, Roopak Sinha, Partha S. Roop,
Alain Girault, Jan Reineke

RESEARCH REPORT
N° 8214
January 2013
Project-Teams SPADES

ISSN 0249-6399 ISRN INRIA/RR--8214--FR+ENG



Precise Modelling of Instruction Cache Behaviour

Sidharta Andalam, Roopak Sinha, Partha S. Roop,
Alain Girault, Jan Reineke

Project-Teams SPADES

Research Report n° 8214 — January 2013 — 62 pages

Abstract: Safety critical real-time applications in aviation, automotive and industrial automation have to guarantee not only the functionality, but also the timeliness of the results. Here, a deadline is associated with the software tasks, and a failure to complete prior to this deadline could lead to a catastrophic consequences. Hence, for the correctness of real-time systems, it is essential to be able to compute the worst case execution time (WCET) of the tasks in order to guarantee their deadlines. However, the problem of WCET analysing is difficult, because of processors use cache-based memory systems that vary memory access time significantly. Any pessimistic estimation of the number of cache hits/misses will result in loose precision of the WCET analyses, which could lead to over use of hardware resources. In this paper, we present a new approach for statically analysing the behaviour of instructions on a direct mapped cache. The proposed approach combines binary representation and a new abstraction that reduces the analysis time without sacrificing the precision. This is unlike the existing cache analysing approaches where either precision or scalability (analysis time) is sacrificed. Experimental results are presented that demonstrate the practical applicability of this analysis.

Key-words: Instruction Cache Analysis, WCET, worst case execution analysis, direct mapped cache.

This project is partially funded by the SPADES project

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Modélisation précise du comportement du cache d'instructions

Résumé : Les applications temps-réel critiques en avionique, automobile, et informatique industrielle doivent garantir non seulement leur fonctionnalité mais aussi le respect des contraintes temporelles. Dans ce domaine, une date limite est associée à chaque tâche logicielle, et un échec dans le respect d'une telle date limite peut conduire à des conséquences catastrophiques. Aussi, pour la correction des applications temps-réel, il est essentiel d'être capables de calculer le temps d'exécution au pire cas (WCET) des tâches logicielles, de façon à garantir le respect des dates limites. Toutefois, l'analyse du WCET est difficile, parce que les processeurs utilisent des mémoires cache (antémémoire) qui font varier significativement les temps d'accès à la mémoire. Toute estimation pessimiste du nombre de succès et d'échecs des accès au cache va entraîner une perte de précision de l'analyse de WCET, et donc une utilisation non-optimale des ressources matérielles. Dans cet article, nous présentons une nouvelle approche pour analyser statiquement le comportement des instructions avec une mémoire cache à correspondance préétablie (« direct-mapped cache »). Notre approche combine une représentation binaire et une nouvelle abstraction qui réduit le temps d'analyse sans perdre de précision. Ceci diffère des approches existante dans lesquelles on doit faire le choix entre la précision et le passage à l'échelle. Les résultats expérimentaux que nous présentons démontrent l'utilité de notre nouvelle analyse de caches.

Mots-clés : Analyse du cache instructions, WCET, analyse du temps d'exécution au pire cas, cache à correspondance préétablie.

Contents

1 Introduction

Most applications can benefit significantly from a fast, cheap and large memory. However in reality, as the memory gets faster, the cost increases and the size decreases. For example, caches are fast and small, but expensive compared to hard disks which are slow, cheap and large. This restriction has forced the computer architects to physically place the fast and smaller memories close to the processor. In contrast, slower and larger memories are placed further away from the processor. Hence, we have a hierarchy of memories as presented in Figure 1. Memory hierarchies are effective because, during an execution of a program, the frequency of accessing storage units at any particular level is more than the frequency of accessing storage at the next lower level. Hence, the next level storage units can be slower. The overall effect is a large memory unit that is very cost effective.

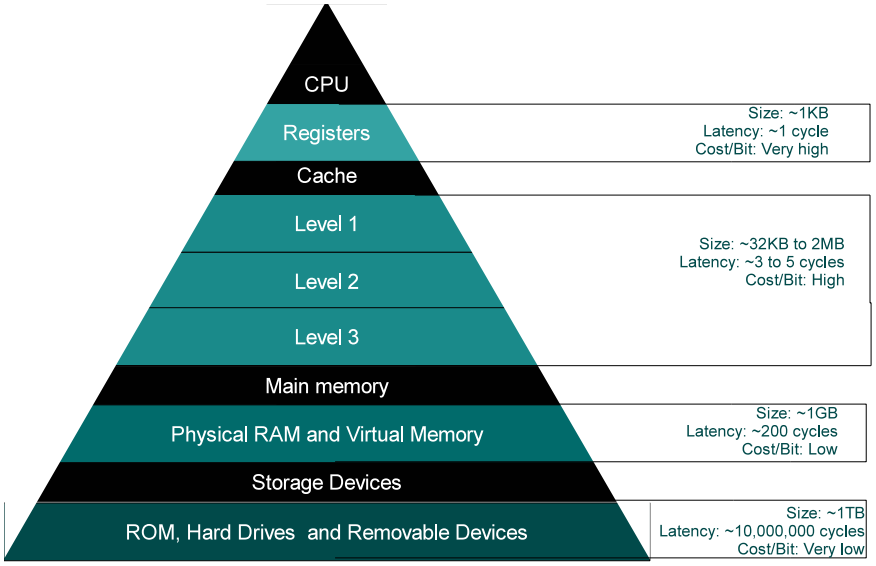


Figure 1: Memory hierarchy

In embedded systems, as the application complexity is growing continuously, the complexity of the underlying memory hierarchy is also increasing. For example, for a simple automatic door control, the computation is very simple and hence a memory architecture with only registers and main memory could meet the requirements. In contrast, applications in automotive industry (such as engine control unit) are much more complicated and would require a complex memory hierarchy [22].

After registers, caches are the next nearest memory to the processor. They are used for temporary storage of instructions, such that future requests can be processed much faster. They act as a bridge between the fast processors and the slower main memory. Caches are effective, because of the following two principles:

1. Temporal locality: Memory locations that are accessed recently are likely to be accessed again, e.g., cycle through instructions of a loop.

2. Spatial locality: Memory locations that are close to the locations that are currently accessed are likely to be accessed soon, e.g., reference to instructions in sequence.

When a processor requests the cache for data at a location in the main memory, the cache first checks it's memory contents. If the requested data is in the cache (cache hit), the data is sent to the processor. Otherwise, if the data is not in the cache (cache miss), it is copied from the main memory into the cache and then forwarded to the processor. However, during this copy, if the cache is already full, a replacement policy will decide which location of the cache memory is to be replaced by the new data. The performance of a cache is measured by the number of cache misses (or cache hits). Thus, the goal of the replacement policy is to reduce the number of cache misses. There are several policies and some are architecture specific. The cache architectures are either classified as set-associative or direct mapped caches [5]. For a set associative cache, each location of the main memory is mapped to more than one location in the cache memory. In general the following three policies are used during replacement [21]:

- Least recently used (LRU): replaces the least recently accessed memory location.
- First in first out (FIFO): replaces the memory that has been in the cache for the longest amount of time.
- Pseudo-LRU (PLRU): is a binary tree based approximation of the LRU policy. The history of the access define the structure of the tree and the leaves represent the cache lines. For more details refer to [15].

In contrast, for a direct mapped caches, the replacement policy is very simple. Each location of the main memory is mapped to exactly only one location in the cache memory. This policy does not require any history bits and updating, unlike the replacement policies mentioned above. Most embedded applications use Direct Mapped Caches because of their simple design, low complexity (requires small hardware), and low power consumption. It also simplifies the static analysis that is required for hard real time systems.

Alternatively, to achieve timing predictability and to simplify static timing analysis, there has been a shift towards predictable memory hierarchy for hard real-time systems. One such approach is to use a cache locking mechanism [14]. In this approach, the contents of the cache are decided at compile time and are loaded prior to the execution of the program. This simplifies the analysis and provides a predictable platform. However, this significantly reduces the throughput. This simple approach has been further extended to dynamically reload and lock the cache at statically determined control points [13]. This increases the throughput at the expense of the analysis time.

Recently, scratchpad memories (SPMs) have been introduced as an alternative to caches for predictable systems. In SPMs, the allocation and replacement decisions are made by software, guided by the decisions made at compile time. In contrast, the allocation and replacement of traditional caches are done dynamically, guided by the history of the cache states. Recent work on SPMs focuses on developing software allocation algorithms [16], [19], [3] and/or designing tailored architectures with SPMs [16], [10], [11].

In this paper, we focus only on single core processors with direct mapped caches. We present two prominent related works in analysing direct mapped caches. In the NUS approach [12], the problem of computing all possible cache states is mapped to a problem of computing the least fixed point. This approach performs very precise analysis. However, as the program size increases, this approach experiences state-space explosion and the analysis time grows exponentially. To address this issue, the same research group has formulated a new probabilistic approach for modelling cache behaviour [9]. It is used for design space explorations and reduces overall analysis time by exploiting the structural similarities among related cache configurations. Experimental results indicate that their new probabilistic based analysis is 24 to 3855 times faster than simulation. Also, they have employed the idea of cache conflict graphs (CCF) in [7]. This

work analyses one cache line at a time and presents a more scalable approach. However, it abstracts the relation between cache lines, thus, sacrificing precision. Also, it would be interesting to quantitatively compare the precision and the analysis time with other existing techniques, like the Absint approach [4], [15]. The Absint approach performs an abstraction based WCET analysis by sacrificing precision for scalability. As a result, it is the preferred approach for analysing cache based architectures for industrial applications [18], [2], [17].

The organisation of this paper is as follows. In Section 2, we formalise the cache analysis problem. In Sections 3 and 4 we present two prominent related works in analysing direct mapped caches. First, the NUS approach [12] is presented in Section 3. Second, the Absint approach [4] is presented in Section 4. We compare both approaches in Section 5. In Section 6, we present the proposed cache analysis approach (UoA approach), along with some optimisations. In Section 7 we present a comparisons between all three (NUS, Absint and UoA) approaches. Finally, in Section 8 we conclude the chapter with some discussions.

2 Cache Analysis

2.1 Cache model

The first step in the cache analysis is the creation of a suitable cache model. Before we formally define the cache model, we provide an illustrative description. Cache analysis requires the following information:

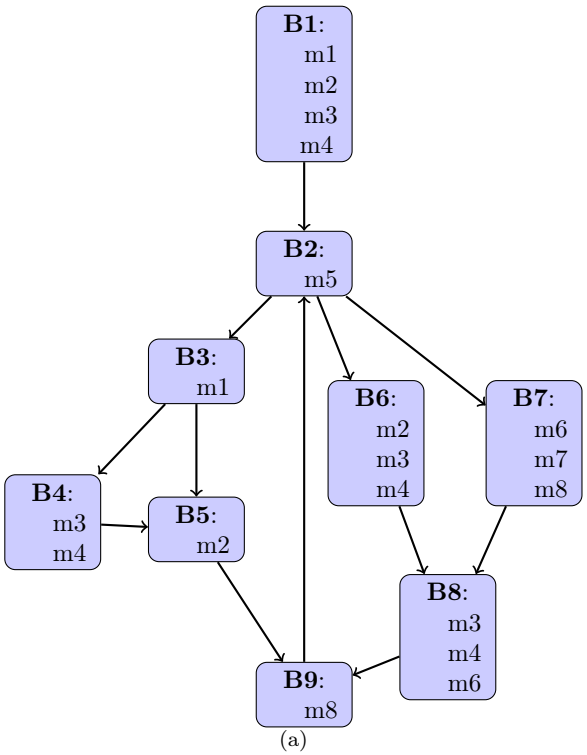
1. *A control flow graph (CFG).* This is extracted from the binaries of an embedded program [20]. The CFG contains basic blocks which are annotated with instructions that are executed in this block. An example CFG is presented in Figure 2(a). It contains nine basic blocks, $B1$ to $B9$. The basic block $B1$ is annotated with four instructions $m1$, $m2$, $m3$ and $m4$ that are executed in this basic block. The entire program (CFG) has eight different instructions $m1$ to $m8$.

2. *A mapping of all instructions to cache lines.* Since, the number of instructions in a program is usually greater than the number of cache lines, more than one instruction is mapped to the same cache line. For the example CFG (presented in Figure 2(a)), we present the instructions to cache lines mapping using Figure 2(b). The cache has four cache lines c_0, c_1, c_2, c_3. Instructions $m1$ and $m5$ are both mapped to the cache line c_0. This means that during program execution, cache line c_0 can contains either $m1$ or $m5$ or *nothing* (when empty).

The above information can be used to map the instructions in each basic block to the cache lines. E.g., for block $B1$, instructions $m1$, $m2$, $m3$, $m4$ are mapped to cache lines c_0, c_1, c_2, c_3 respectively (as presented in Figure 2(b)). If we enforce $C = \{c_0, c_1, c_2, c_3\}$, the set of cache lines, to be an ordered set, then the instructions of $B1$ can be described using a vector $[m1, m2, m3, m4]$ where the index of the i^{th} element corresponds to the i^{th} cache line. E.g., the 2^{nd} element $m2$ corresponds to the 2^{nd} cache line c_1. Similarly, the basic block $B2$ has only one instruction $m5$ that is mapped to cache line c_0 and the rest of the cache lines have *no instructions* (represented using the symbol $\mp$). Hence, the instructions of $B2$ can be described using the vector $[m5, \mp, \mp, \mp]$.

For the rest of this section, we formalise the cache model, the behaviour of the cache, and the cache analysis problem.

Definition 1 (Cache model)**.** *The cache model for a given program is defined as a tuple* $CM = \langle I,\ C,\ CI,\ G,\ BI \rangle$*, where:*



(a)

Cache line	Instructions
c_0	m1,m5
c_1	m2,m6
c_2	m3,m7
c_3	m4,m8

(b)

Figure 2: (a) A simple control flow graph consisting of nine basic blocks (B1 to B9) and the instructions that are accessed during execution of the basic block. (b) Mapping of instructions on to four cache lines (c_0 to c_3).

- *I is a finite set of instructions and $|I|$ is the total number of instructions in the program.*
- *$C = \{c_0, c_1, \ldots, c_{N-1}\}$ is an ordered set of cache lines. $N = |C|$ is the total number of cache lines.*
- *$CI : C \rightarrow 2^I$ is the cache line to instructions mapping function. For a cache line $c \in C$, $CI(c)$ is a subset of I representing the set of instructions that are mapped to cache line c. We restrict CI such that for any two cache lines c_k and c_l, $CI(c_k) \cap CI(c_l) = \emptyset$ and further $\bigcup_{k=0}^{N-1} CI(c_k) = I$ i.e., CI partitions I into N partitions, where each partition represents the instructions mapped to a given cache line.*

  Also, we define the instruction to cache line mapping function $IC : I \rightarrow C$. For an instruction $i \in C$, $IC(i) = c$, where $i \in CI(c)$.
- *G is a directed graph $G = \langle B, b_{init}, E \rangle$ where:*
  - *B is a finite set of basic blocks.*
  - *$b_{init} \in B$ is the initial basic block.*
  - *$E \subseteq B \times B$ is the set of edges. For short hand, $(b_0, b_1) \in E$ is represented using $b_0 \rightarrow b_1$.*

- $BI : B \rightarrow (I \cup \{\mp\})^N$ *is the block to instruction mapping function. For any given basic block* $b \in B$, $BI(b) = [inst_0, inst_1, \ldots, inst_{N-1}]$. *For any* $i \in [0, N-1]$, $BI(b)[i]$ *is the instruction mapped to the* i^{th} *cache line* $c_i \in C$. *Thus,* $BI(b)[i] \in CI(c_i) \cup \{\mp\}$.

Illustration of the cache model

Using Figure 2, we illustrate the cache model $CM = \langle I, C, CI, G, BI \rangle$ as follows:

- $I = \{m1,\ m2,\ m3,\ m4,\ m5,\ m6,\ m7,\ m8\}$ and $|I| = 8$.
- $C = \{c_0, c_1, c_2, c_3\}$ is an ordered set and $N = |C| = 4$.
- $CI(c_0) = \{m1, m5\}$ and $IC(m1) = c_0$.
- $G = (B, b_0, E)$ where $B = \{B1, B2, \ldots, B9\}$, $b_0 = B1$ and $E = \{(B1, B2), (B2, B3), \ldots\}$.
- $BI(B1) = [m1, m2, m3, m4]$ and $BI(B1)[1] = m2$. Similarly, $BI(B8) = [\mp, m6, m3, m4]$ and $BI(B8)[0] = \mp$, which represents that block $B9$ does not contain any instruction that is mapped to the cache line c_0. The order of the instructions in the vector are based on the mapping of the instructions to the cache lines. It is not based on the order of the instructions specified by the programmer, i.e., $BI(B8) \neq [\mp, m3, m4, m6]$.

2.2 Cache states

We describe the behaviour of the cache by first describing cache states. A cache state describes the instructions in the cache at some instance in time. It is described as a vector $[inst_0, inst_1, \ldots, inst_{N-1}]$ where each $inst_i$ represents the instructions contained in cache line $c_i \in C$.

For the example CFG shown in Figure 2(a), when the program starts executing, we assume the cache is empty. This is represented as the cache state $cs_\top = [\top, \top, \top, \top]$. Instructions are loaded into the cache as the basic blocks are executed (starting from the initial block). E.g., after executing the basic block $B1$ the cache state is $cs = [m1, m2, m3, m4]$.

Generally, the status of the cache may be *unknown* when a block is being analysed. We used the symbol $\perp$ to represent an unknown instruction in a cache line. E.g., $cs = [\perp, \perp, \perp, \perp]$ represents the status of every cache line is unknown. This notion of unknown cache state is used during the cache analysis and is explained later in Section 3.

Note that a cache state has identical structure to the block to instructions mapping function $BI(b)$ for any block $b \in B$. E.g., The cache state after executing $B1$ is $cs = [m1, m2, m3, m4]$ and also, $BI(B1) = [m1, m2, m3, m4]$. The cache state represents the instructions in the cache, while the function BI represents the instructions that are executed by the basic block. This identical structure simplifies the cache analysis as simple comparison between vectors allows us to analyse the cache misses. This is later explained in later part of this section.

We now define the cache states.

Definition 2 (Cache state)**.** *A Cache state* $cs \in (I \cup \{\top, \perp\})^N$, *can be described as a vector* $[inst_0, inst_1, \ldots, inst_{N-1}]$ *where each element* $cs[i]$ *represents an instruction in cache line* c_i. *Further, we restrict cache states such that, for any* $i \in [0, N-1]$, $cs[i] \in CI(c_i) \cup \{\top, \perp\}$. *We denote* $cs_\top = [\top, \ldots, \top]$ *as a empty cache state where all the elements of the vector are* $\top$. *Similarly, we denote* $cs_\perp = [\perp, \ldots, \perp]$ *as a unknown cache state where all the elements of the vector are* $\perp$. *Also, the set of all cache states is defined as* CS.

Before we illustrate cache states, we introduce two key terms essential to cache analysis. For any basic block b, the *Reaching cache states* (RCS) represent the set of cache states prior to the

execution of a basic block and the *Leaving cache states* (LCS) represent the set of cache states after the execution of the basic block. We denote the RCS of a basic block b as RCS_b and the LCS of a basic block b as LCS_b.

Illustration of cache states

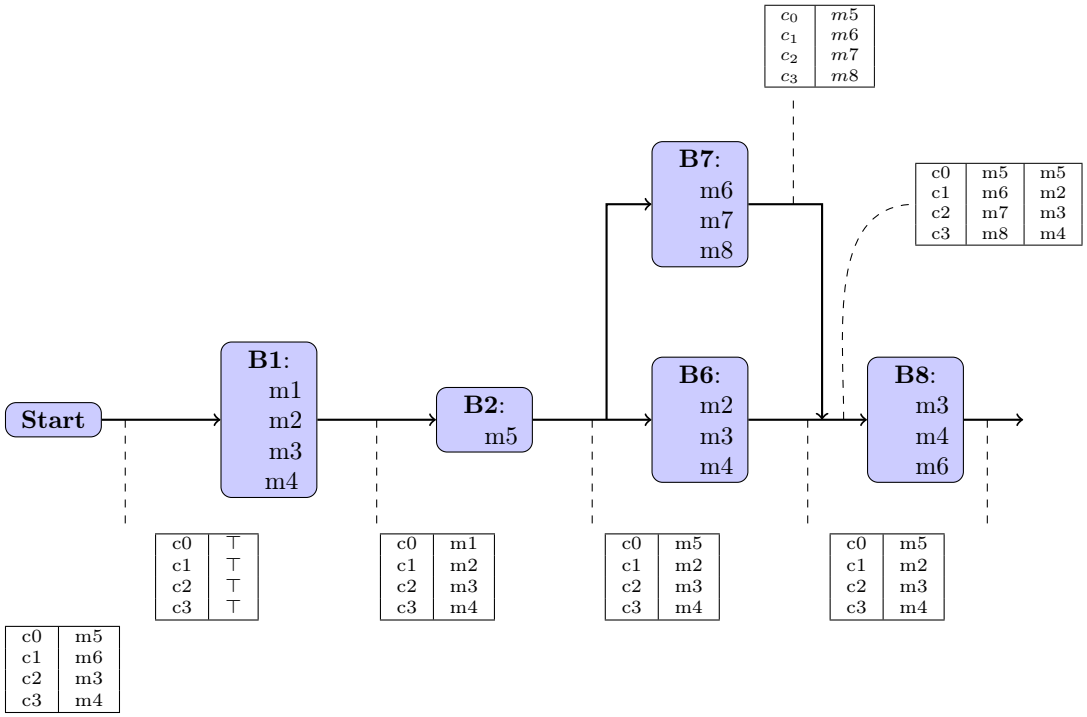


Figure 3: Illustration of the cache states.

Using Figure 3, we illustrate reaching/leaving cache states of a block. Initially the cache is empty. The state of the cache is $cs_\top = [\top, \top, \top, \top]$. After the execution of block $B1$, cache lines c_0, c_1, c_2, c_3 will contain instructions $m1$, $m2$, $m3$, $m4$ respectively, resulting in the state $[m1, m2, m3, m4]$. The cache state $[\top, \top, \top, \top]$ is the state of the cache prior to the execution of the basic block $B1$. Thus, $RCS_{B1} = \{[\top, \top, \top, \top]\}$ is the set of reaching cache states of block $B1$. Similarly, $LCS_{B1} = \{[m1, m2, m3, m4]\}$ is the set of leaving cache states of block $B1$.

The leaving cache state of $B1$ is also the reaching cache state of $B2$, $RCS_{B2} = LCS_{B1}$. Block $B2$ has only one instruction $m5$, and is mapped to cache line c_0. Thus, the previous instruction in line c_0 will be replaced by $m5$. Since the $RCS_{B2} = \{[m1, m2, m3, m4]\}$, after executing the instructions in $B2$, the resulting leaving cache state is $LCS_{B2} = \{[m5, m2, m3, m4]\}$.

Now, the control reaches a branch due to which, it is possible to execute either block $B6$ or block $B7$. In this case, $RCS_{B6} = LCS_{B2} = RCS_{B7}$. After executing blocks $B6$ or $B7$, the state of the cache is either $[m5, m2, m3, m4]$ or $[m5, m6, m7, m8]$.

Block $B8$ has two incoming edges: from blocks $B6$ and $B7$. In this case, in order to compute RCS_{B8}, we need to *join* LCS_{B6} and LCS_{B7}. There are various approaches to describe this join function [12], [4]. This join function is the key difference between existing approaches and could dictate the precision and the scalability of the cache analysis. Later in this chapter, we formally present several join functions. See the summary in Section 7

For now, we assume the join function is a union over the set of cache states. Thus, $RCS_{B8} = LCS_{B7} \cup LCS_{B6}$, resulting in $RCS_{B8} = \{[m5, m2, m3, m4], [m5, m6, m7, m8]\}$. After execution of block $B8$, for both reaching cache states the resulting leaving cache state is $[m5, m6, m3, m4]$. Thus, the $LCS_{B8} = \{[m5, m6, m3, m4]\}$.

2.3 Analysing the cache states

The purpose of cache analysis is to compute the number of cache misses experienced during the execution of every basic block. For example, for block $B8$ consider the cache state $cs = [m5, m2, m3, m4]$ before executing the block. Since $cs[1] = m2$, the cache contains the instruction $m2$ on the cache line c_1, before executing $B8$. Now, $BI(B8) = [\mp, m6, m3, m4]$ and $BI(B8)[1] = m6$ for cache line c_1. This means, for cache line c_1, the cache contains the instruction $m2$, whereas the block needs to execute $m6$. Since, we have a mismatch ($BI(B8)[1] \neq cs[1]$), $B8$ has a cache *miss* during the execution of instruction $m6$ on the cache line c_1.

Similarly, since $BI(B8)[2] = m3 = cs[2]$, we have a cache *hit*. Also, $BI(B8)[0] = \mp$, which represents that block $B8$ does not execute an instruction from cache line c_0. Therefore, $B8$ can not have a miss for cache line c_0, regardless of the content of a reaching cache state. Thus, a cache miss on a cache line c_i can only occur when the block b has an instruction that can be mapped to a cache line c_i (i.e., $BI(b)[i] \neq \mp$) and further the instruction does not match with the instruction in the reaching cache state (i.e., $BI(b)[i] \neq cs[i]$). Using Definition 20, we define a function that computes the number of cache misses.

Definition 3 (Miss count)**.** *The cache miss function, $mc : B \times CS \times C \to \{0, 1\}$, where for any block $b \in B$, cache state cs and a cache line $c_i \in C$ with $i \in [0, N-1]$ is defined as follows:*

$$mc(b, cs, c_i) = \begin{cases} 1 & : if \quad BI(b)[i] \neq \mp \ \wedge BI(b)[i] \neq cs[i] \\ 0 & : otherwise \end{cases}$$

We extend the function mc to return the total number of misses for a block b and a cache state cs as, $mc : B \times CS \to \mathbb{N}^0$ where,

$$mc(b, cs) = \sum_{i=0}^{N-1} mc(b, cs, c_i)$$

2.3.1 Illustration

c_i		c_0	c_1	c_2	c_3
cs	$=$	$[m5$	$m2$	$m3$	$m4]$
			$\neq$	$=$	$=$
$BI(B8)$	$=$	$[\ \mp$	$m6$	$m3$	$m4]$
$mc(B8, cs, c_i)$	$=$	0	1	0	0

$$mc(B8, cs) \quad = \quad 0 + 1 + 0 + 0 = 1$$

Figure 4: Illustration of the function mc

Given a $cs = [m5, m2, m3, m4]$ and $BI(B8) = [\mp, m6, m3, m4]$, we now illustrate the mc function using Figure 4. For all cache lines $c_i \in C$, the figure shows that there is only one cache miss and it occurs on cache line c_1, because instruction $m2$ is present in the reaching cache state cs while the block needs instruction $m8$. Thus, $mc(B8, cs, c_1) = 1$. Similarly, $mc(B8, cs, c_0) = 0$, $mc(B8, cs, c_2) = 0$ and $mc(B8, cs, c_3) = 0$. Hence, $mc(B8, cs) = 0+1+0+0 = 1$. Thus, only one cache miss occurs during the execution of block $B8$ for reaching cache state of $[m5, m2, m3, m4]$.

The mc function computes the number of cache misses for a given reaching cache state and a basic block. However, if a block has more than one reaching cache states, the number of cache misses can vary. For cache analysis, we are interested in finding the best case and the worst case number of cache misses that may be experienced by a basic block.

Definition 4 (Worst miss count)**.** *We define the function* $wmc : B \times 2^{CS} \rightarrow \mathbb{N}^0$ *where, for a given block* $b \in B$ *and a set* $RCS' \subseteq CS$ *such that* $RCS' = \{rcs_1, rcs_2, \ldots, rcs_m\}$ *and* $m = |RCS'|$,

$$wmc(b, RCS') = MAX(mc(b, rcs_1), \ldots, mc(b, rcs_m))$$

Definition 5 (Best miss count)**.** *We define the function* $bmc : B \times 2^{CS} \rightarrow \mathbb{N}^0$ *where, for a given block* $b \in B$ *and a set* $RCS' \subseteq CS$ *such that* $RCS' = \{rcs_1, rcs_2, \ldots, rcs_m\}$,

$$bmc(b, RCS') = MIN(mc(b, rcs_1), \ldots, mc(b, rcs_m))$$

Illustration of *wmc* **and** *bmc*
Using the reaching caches states for $B8$ (RCS_{B8}), we now illustrate the functions wmc and bmc. Since, RCS_{B8} contains two cache states of $\{[m5, m2, m3, m4], [m5, m6, m7, m8]\}$, using the $mc(B8, [m5, m2, m3, m4]) = 1$ or $mc(B8, [m5, m6, m7, m8]) = 2$. Since, the maximum number of misses is 2, thus, $wmc(B8, RCS_{B8}) = 2$. Similarly, the function $bmc(B8, RCS_{B8}) = 1$.

2.4 Cache analysis problem

The cache analysis problem consists of two sub problems. The first sub problem is to compute all the possible reaching cache states for every basic block in the given cache model. The second sub problem is to compute the number of cache misses in the best case and the worst case, using the reaching cache states computed in sub problem one.

Definition 6 (Cache Analysis Problem)**.** *Given* CM*, the cache analysis problem can be stated as the computation of:*

1. *All possible* RCS_b *for every block* $b \in B$*. We refer to this first problem as* sub problem one.
2. $P : B \rightarrow \mathbb{N}^0 \times \mathbb{N}^0$*, where for every block* $b \in B$*,* $P(b) = (wmc(b, RCS_b), bmc(b, RCS_b))$ *for all* $b \in B$*. We refer to this second problem as* sub problem two.

Different cache analysis approaches differ in how they compute sub problem one (presented in Definition 6). We present two existing approaches, the NUS approach [12] and the Absint approach [4] to solve the cache analysis problem. In Table 1 we present a summary of the symbols and the definitions that are presented so far.

Symbol/Definition	**Description**	**Example**
CM	Cache model of a given program	see Section 2.1
B	Set of basic blocks	$\{B1, B2, \ldots, B9\}$ in Figure 2
I	Set of instruction	$\{m1, m2, \ldots, m8\}$ in Figure 2
C	Ordered set of cache lines	$\{c_0, c_1, c_2, c_3\}$ in Figure 2
$CI : C \rightarrow 2^I$	Maps cache line to a set of instructions	$CI(c_0) = \{m1, m5\}$
$IC : I \rightarrow C$	Maps instruction to a cache line	$IC(m1) = c_0$
$BI : B \rightarrow (I \cup \{\mp\})^N$	Maps instructions executed by the block to cache lines	$BI(B1) = [m1, m2, m3, m4]$ then $BI(B1)[0] = m1$
$BI(b)[i] = \mp$	Block b has no instruction on cache line c_i	$BI(B1) = [\mp, m2, m3, m4]$ then $BI(B1)[0] = \mp$
cs	Cache state	$cs = [m1, m2, m3, m4]$ and $cs[0] = m1$
$cs[i] = \top$	Cache state has no instruction on cache line c_i	$cs[0] = \top$
$cs[i] = \bot$	Cache state has an unknown instruction on cache line c_i	$cs[0] = \bot$
$cs_\top$	Empty cache state	$cs = [\top, \top, \top, \top]$
$cs_\bot$	Unknown cache state	$cs = [\bot, \bot, \bot, \bot]$
RCS_b	Set of reaching cache states of block b	$RCS_{B8} = \{[m5, m2, m3, m4], [m5, m6, m7, m8]\}$
LCS_b	Set of leaving cache states of block b	$LCS_{B8} = \{[m5, m6, m3, m4]\}$
$\mathbb{N}^0$	Natural numbers that includes zero	$\mathbb{N}^0 = \{0, 1, 2, 3, \ldots\}$
$mc : B \times CS \rightarrow \mathbb{N}^0$	Computes the number of cache misses experienced by the block for a given cache state	$BI(B8) = [\mp, m6, m3, m4]$ and $cs = [m5, m2, m3, m4]$ then $mc(B8, cs) = 1$
$wmc : B \times 2^{CS} \rightarrow \mathbb{N}^0$	Computes the maximum number of cache misses	$wmc(B8, RCS_{B8}) = 2$
$bmc : B \times 2^{CS} \rightarrow \mathbb{N}^0$	Computes the minimum number of cache misses	$bmc(B8, RCS_{B8}) = 1$

Table 1: Some of the symbols and the definitions (Illustrated using Figures 3)

3 The NUS approach

We first present two functions, *join* and *transfer*, that compute the of reaching/leaving cache states of basic blocks. These two functions are used in the cache analysis algorithm, employed by this approach, to compute all possible reaching cache states of every basic block of a cache model (sub problem 1, Definition 6).

3.1 The NUS join function

The NUS join function describes how to compute the reaching cache states of a basic block when the block has more than one incoming edges. E.g., in Figure 3, the basic block $B8$ has two incoming edges: from blocks $B6$ and $B7$. To compute RCS_{B8} we need to *join* LCS_{B6} and LCS_{B7}. A simple join function could be described as an union over the sets of reaching cache states, i.e., $RCS_{B8} = LCS_{B6} \cup LCS_{B6}$. However, in the NUS approach, during the computation of the reaching cache states *unknown cache states* ($cs_\perp$) are introduced as part of the analysis (later explained in Algorithm 1). This introduces spurious cache states that *must* be removed as described in the NUS approach [12]. Given two cache states cs_1 and cs_2, cs_1 is *subsumed* by cs_2, if for all cache lines c_i, the instruction in cs_2 is equal to the instruction in cs_1 ($cs_2[i] = cs_1[i]$) or the instruction in cs_1 at cache line c_i is unknown $\perp$ ($\forall i, cs_2[i] = cs_1[i]$ *or* $cs_1[i] = \perp$). In this case, we can remove cs_1, as it does not contain any extra information compared to cs_2.

For example, let $RCS_b = LCS_1 \cup LCS_2$, it contains two cache states $cs_1 = [m1, \perp, \perp, \perp]$ and $cs_2 = [m1, \perp, m2, \perp]\}$. Since cs_1 is subsumed by cs_2, we can reduce the reaching cache states of b to $RCS_b = \{[m1, \perp, m2, \perp]\}$. Also, a cache state $cs_\perp = [\perp, \perp, \ldots, \perp]$ will always we subsumed by any other cache state. This idea of subsumed cache states is further explained during the fixed point computation (see Section 3.3).

This join function for the NUS approach is defined in Definition 7.

Definition 7 (NUS join function)**.** *The NUS join function is* $J_{NUS} : 2^{CS} \times 2^{CS} \rightarrow 2^{CS}$ *where, for any two sets of cache states* $CS_1 \subseteq 2^{CS}$ *and* $CS_2 \subseteq 2^{CS}$*,*

$$J_{NUS}(CS_1, CS_2) = CS_1 \cup CS_2 \setminus \{cs_1 \in (CS_1 \cup CS_2) | \exists cs_2 \in (CS_1 \cup CS_2) \wedge S_{NUS}(cs_1, cs_2)\}$$

where, $S_{NUS} : CS \times CS \rightarrow \{true, false\}$ *where, for any* $cs_1, cs_2 \in CS$*,*

$$S_{NUS}(cs_1, cs_2) = \begin{cases} true & : if \ \ \forall i \in [0, N-1], cs_2[i] = cs_1[i] \ or \ cs_1[i] = \perp \\ false & : otherwise \end{cases}$$

cs_1	cs_2	$S_{NUS}(cs_1, cs_2)$
$[\perp, m2, \perp, m4]$	$[m5, m2, \perp, m4]$	$true$
$[m5, m2, \perp, m4]$	$[\perp, m2, \perp, m4]$	$false$

Table 2: Illustrate the subsumed function S_{NUS}

Using Table 2 we first illustrate the NUS subsumed function (S_{NUS}). In the first row, given two cache states $cs_1 = [\perp, m2, \perp, m4]$ and $cs_2 = [m5, m2, \perp, m4]$, $S_{NUS}(cs_1, cs_2) = true$. This shows the case when cs_1 is subsumed by a cs_2. In the second row, given two cache states

CS_1	CS_2	$CS_1 \cup CS_2$	$J_{NUS}(CS_1, CS_2)$
$\{[\perp, m2, \perp, m4],$ $[m5, m6, m7, m8]\}$	$\{[m5, m2, \perp, m4],$ $[m5, m6, m7, m8]\}$	$\{[\perp, m2, \perp, m4],$ $[m5, m2, \perp, m4],$ $[m5, m6, m7, m8]\}$	$\{[\perp, m2, \perp, m4],$ $[m5, m6, m7, m8]\}$

Table 3: Illustrate the join function J_{NUS}

$cs_1 = [m5, m2, \perp, m4]$ and $cs_2 = [\perp, m2, \perp, m4]$, $S_{NUS}(cs_1, cs_2) = false$. This shows the case when cs_1 is not subsumed by cs_2.

Using Table 3, we illustrate the NUS join function. Given two sets of cache states $CS_1 = \{[\perp, m2, \perp, m4], [m5, m6, m7, m8]\}$ and $CS_2 = \{[m5, m2, \perp, m4], [m5, m6, m7, m8]\}$, $CS_1 \cup CS_2 = \{[\perp, m2, \perp, m4], [m5, m2, \perp, m4], [m5, m6, m7, m8]\}$. As illustrated in first row of Table 2, cache state $[\perp, m2, \perp, m4]$ is subsumed by $[m5, m2, \perp, m4]$. Thus, $J_{NUS}(CS_1, CS_2) = \{[m5, m2, \perp, m4], [m5, m6, m7, m8]\}$.

3.2 The NUS transfer function

A transfer function described how a reaching cache state of a basic block is transformed into a leaving cache state, after executing the instructions in the basic block. E.g., in Figure 3, the reaching cache state of block $B2$ is $cs = [m1, m2, m3, m4]$ and after execution, where the instructions of the block as described by $BI(B2) = [m5, \mp, \mp, \mp]$, the leaving cache state of $B2$ is $cs' = [m5, m2, m3, m4]$. This transformation can be described as follows. First, if a block b does not have an instructions mapped to a cache line c_i $(BI(b)[i] = \mp)$, then after execution of the block, the contents of the cache corresponding to that cache line c_i remains the same. E.g., basic block $B2$ has no instruction mapped to the cache line c_1, $(BI(B2)[1] = \mp)$. Thus, after execution, the leaving cache state has the same instruction as the reaching cache state $cs'[1] = m2 = cs[1]$. Secondly, if a block has an instruction mapped to a cache line $(BI(b)[i] \neq \mp)$, then, after execution of the block, the content of the cache corresponding to that cache line is changed to the instruction executed by the block. E.g., basic block $B2$ has the instruction $m5$ mapped to the cache line c_0, $(BI(B2)[0] \neq \mp)$. Thus, after execution, the leaving cache state has the instruction $m5$, $cs' = m5 = BI(B2)[0]$. The transfer function, T_{NUS}, formalises this transformation as follows.

Definition 8 (NUS transfer function)**.** *The NUS transfer function is $T_{NUS} : B \times CS \to CS$ where, for a given basic block $b \in B$ and a cache state cs,*

$$T_{NUS}(b, cs) = cs'$$

where for all cache lines $c_i \in C$ where $i \in [0, N-1]$,

$$cs'[i] = \begin{cases} cs[i] & : if\ BI(b)[i] = \mp \\ BI(b)[i] & : otherwise \end{cases}$$

Illustration of the transfer function T_{NUS}

Figure 5 illustrates the operation of the transfer function T_{NUS}. Given block b, with $BI(b) = [m5, \mp, \mp, \mp]$, and a reaching cache state $cs = [m1, m2, m3, m4]$, the leaving cache state $cs' = T_{NUS}(b, cs)$ is computed as follows.

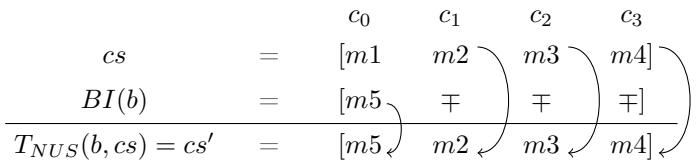


Figure 5: Illustration of the NUS transfer function

For cache line c_0, block b requires an instruction $m5$ on cache line c_0 ($BI(b)[0] = m5$). Thus, irrespective of the reaching cache state, the leaving cache (cs') contains the instruction $m2$ on its cache line c_0, i.e., $cs'[0] = m5 = BI(b)[0]$.

For the cache line c_1, block b has no instruction mapped to cache line c_1 ($BI(b)[1] = \mp$). Thus, the instruction $m2$ on cache line c_1 in cs ($m2 = cs[1]$), is copied to the cache line c_1 of the leaving cache state cs' i.e., $cs'[1] = m2 = cs[1]$. Similarly, block b has no instructions mapped to cache lines c_2 and c_3, resulting in $cs'[2] = m3 = cs[2]$ and $cs'[3] = m4 = cs[3]$. Thus, given a block b, with $BI(b) = [m5, \mp, \mp, \mp]$ and a reaching cache state $cs = [m1, m2, m3, m4]$, $T_{NUS}(b, cs) = cs' = [m5, m2, m3, m4]$.

The transfer function, as defined in Definition 8, computes a leaving cache state for a basic block given a reaching cache state. During cache analysis, a block may have a set of reaching cache states. E.g., in Figure 3, block $B8$ has two reaching cache states, $RCS_{B8} = \{[m5, m2, m3, m4], [m5, m6, m7, m8]\}$. Thus, we extend the transfer function to compute a set of leaving cache states for a given set of reaching cache states for a block.

Definition 9 (Extended NUS transfer function)**.** *The NUS transfer function is* $T_{NUS} : B \times 2^{CS} \rightarrow 2^{CS}$ *where, for a given basic block* $b \in B$ *and a set of cache states* $CS \subseteq 2^{CS}$*,*

$$T_{NUS}(b, CS) = CS'$$

where for all cache states $cs_i \in CS$ *where* $i \in [1, |CS|]$*,*

$$CS' = \bigcup_{i=1}^{|CS|} \{T_{NUS}(b, cs_i)\}$$

In Figure 3, block $B8$ has two reaching cache states ($RCS_{B8} = \{cs_1, cs_2\}$) where $cs_1 = [m5, m2, m3, m4]$ and $cs_2 = [m5, m6, m7, m8]$. In this case, $T_{NUS}(B8, \{cs_1, cs_2\}) = \{T_{NUS}(B8, cs_1)\} \cup \{T_{NUS}(B8, cs_2)\}$. In this example, $T_{NUS}(B8, cs_1) = [m5, m6, m3, m4]$ and $T_{NUS}(B8, cs_2) = [m5, m6, m3, m4]$. Thus, $T_{NUS}(B8, \{cs_1, cs_2\}) = \{[m5, m6, m3, m4]\} \cup \{[m5, m6, m3, m4]\} = \{[m5, m6, m3, m4]\}$

3.3 Fixed point computation

As described in Definition 6, the cache analysis problem consists of two sub problems. The first sub problem is the computation of all reaching cache states for every block $b \in B$. The NUS approach [12] solves this problem using a fixed point computation algorithm presented in Algorithm 1. We will present the NUS solution to the second sub problem following the fixed point algorithm (Algorithm 1).

Algorithm 1 Fixed point computation for the NUS approach

Input: A cache model $CM = \langle I,\ C,\ CI,\ G,\ BI \rangle$
Output: RCS_b for every block $b \in B$.

```
1:  i = 1 {iteration counter}
2:  {Initialise the RCS of all blocks}
3:  for each b ∈ B do
4:     if b = b_0 then
5:        RCS^i_{b_0} = {cs_⊤}
6:     else
7:        RCS^i_b = {cs_⊥}
8:     end if
9:  end for

10: repeat
11:    {Compute the set of leaving cache states for all blocks for iteration i}
12:    for each b ∈ B do
13:       LCS^i_b = T_NUS(b, RCS^i_b)
14:    end for

15:    i = i + 1; {Next iteration}
16:    {Compute RCS}
17:    for each b ∈ B do
18:       if b = b_0 then
19:          RCS^i_b = {cs_⊤}
20:       else
21:          RCS^i_b = ∅
22:          for each LCS^i_{b'}, where (b', b) ∈ E do
23:             RCS^i_b = J_NUS(RCS^i_b, LCS^{i-1}_{b'})
24:          end for
25:       end if
26:    end for

27: until ∀b ∈ B, RCS^i_b = RCS^{i-1}_b {Termination condition}
28: return RCS^i_b for all b ∈ B
```

In Algorithm 1, we first initialise the reaching cache states for all blocks (lines 3 to 9, Algorithm 1). Since we assume that initially the state of the cache is empty, on line 5 for the initial block b_0 we set its reaching as $RCS^i_{b_0} = \{cs_\top\}$. Here, the notation RCS^i_b represents the reaching cache states of block b in iteration i. E.g., $RCS^1_{b_0}$ represents the reaching cache states of block b_0 for iteration 1. For rest of the blocks, the execution of the CFG will impact the state of the cache, the state of the cache is unknown. Thus, on line 7, we set their reaching cache states as $RCS^1_b = \{cs_\bot\}$.

After initialisation, we compute the leaving cache states of each block, on lines 11 to 14. We apply the transfer function T_{NUS} to every block and its corresponding reaching cache states.

The iteration index (i) is incremented (line 15) to signal the start of the next iteration. Next, on lines 16 to 26, the reaching cache states of each block are computed. For the initial block b_0, we know that the reaching cache state is always empty. Thus, on line 19, we always set its reaching as $RCS^i_{b_0} = \{cs_\top\}$. For rest of the blocks, on lines 22 to 24, the reaching cache states are computed by looking at the leaving cache states of the predecessors (b') of the block b and using the NUS join function.

The iterative process, repeat-until loop on lines 10 to 27, is repeated until a fixed point is

reached, i.e., if two consecutive iterations have the same sets of reaching cache states for all blocks (line 27).

Illustration of the fixed point algorithm

We illustrate the above fixed-point algorithm using the Figure 6. The CFG and the mapping of the instructions to cache lines are presented in Figure 6(a) and Figure 6(b), respectively. These are reproduced from Figure 2. Figure 6(c) presents a table showing the reaching and leaving cache states for all blocks, during each iteration of the algorithm.

We first initialise the reaching cache states for all blocks (lines 3 to 9). For the CFG in Figure 6(a), the initial block is b_0 is $B1$. Since we assume that initially the state of the cache is empty, on line 5, we set RCS^1_{B1} as $\{[\top,\top,\top,\top]\}$. For rest of the blocks, on line 7, the state of the cache is unknown. Thus, we set their reaching cache states as $\{[\bot,\bot,\bot,\bot]\}$.

After initialisation, to compute the leaving cache states of each block, we apply the transfer function T_{NUS} (lines 11 to 14). For example, in iteration 1 ($i = 1$), $LCS^1_{B1} = T_{NUS}(B1, RCS^1_{B1}) = T_{NUS}(B1, \{[\top,\top,\top,\top]\}) = \{[m1, m2, m3, m4]\}$. For block $B2$, which has one instruction $m5$ that is mapped to line c_0, the leaving cache states $LCS^1_{B2} = T_{NUS}(B2, \{[\bot,\bot,\bot,\bot]\}) = \{[m5,\bot,\bot,\bot]\}$. Similarly, the leaving cache states (column 4) for rest of the blocks are computed for iteration 1 as shown in Figure 6(c).

The iteration index (i) is incremented (line 15) to signal the start of the next iteration ($i = 2$). Next, the reaching cache states of each block, for iteration $i = 2$, are computed (lines 16 to 26). For the initial block $B1$, we know that the reaching cache state is always $cs_\top$. Thus, on line 19, we set its reaching cache state as $RCS^2_{B1} = \{cs_\top\}$. For rest of the blocks, on lines 22 to 24, the reaching cache states are computed by looking at the leaving cache states of the predecessors (b') of the block b and using the NUS join function. E.g., the predecessors of $B2$ are $B1$ and $B9$, $b' \in \{B1, B9\}$, and from the previous iteration ($i = 1$) the $LCS^1_{B1} = \{[m1, m2, m3, m4]\}$ and $LCS^1_{B9} = \{[\bot,\bot,\bot, m8]\}$. The process for computing RCS^2_{B2} is as follows.

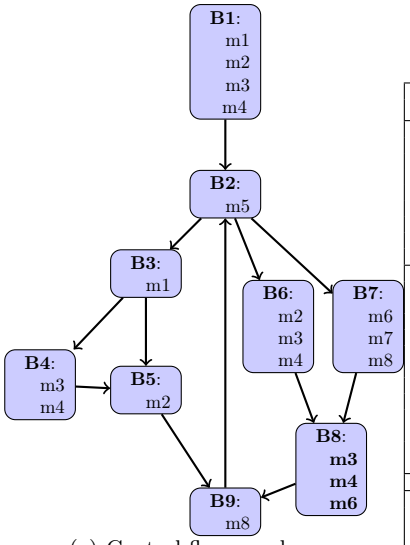
1. Initialise $RCS^2_{B2} = \emptyset$ (line 21).
2. Using the NUS join function we process the LCS of each predecessor block ($B1, B9$), on lines 22 to 24, one at a time (in no particular order). For illustration we will first analyse LCS^1_{B1} followed by LCS^1_{B9} as follows.

$$\begin{aligned} RCS^2_{B2} &= J_{NUS}(RCS^2_{B2}, LCS^1_{B1}) \\ &= J_{NUS}(\emptyset, \{[m1, m2, m3, m4]\}) \\ &= \{[m1, m2, m3, m4]\} \\ RCS^2_{B2} &= J_{NUS}(RCS^2_{B2}, LCS^1_{B9}) \\ &= J_{NUS}(\{[m1, m2, m3, m4]\}, \{[\bot,\bot,\bot, m8]\}) \\ &= \{[m1, m2, m3, m4], [\bot,\bot,\bot, m8]\} \end{aligned}$$

The iterative process, repeat-until loop on lines 10 to 27, is repeated until a fixed point is reached, i.e., if two consecutive iterations have a same sets of reaching cache states for all blocks (line 27). For our example, it happens at the 9^{th} iteration, where the same sets of reaching cache states are computed for all blocks. As a result, the reaching cache states of the 9^{th} iteration represents all the possible reaching cache states of the program.

Note that during the first iteration, we assumed the reaching cache states for blocks $B2$ to $B9$ as unknown ($\{[\bot,\bot,\bot,\bot]\}$), but when we reach the fixed-point, all the unknown cache states are resolved.

Finally, using the functions $wmc(b, RCS^9_b)$ and $bmc(b, RCS^9_b)$ (presented in Definitions 4 and Definition 5), the number of cache misses in the worst and the best case is computed. E.g.,



(a) Control flow graph

Cache line	Instructions
c_0	m1,m5
c_1	m2,m6
c_2	m3,m7
c_3	m4,m8

(b) Instructions to cache line mapping

Iteration (i)	Block (b)	Reaching Cache States (RCS_b^i)	Leaving Cache States (LCS_b^i)
1	B1	$\{[\top, \top, \top, \top]\}$	$\{[m1, m2, m3, m4]\}$
	B2	$\{[\bot, \bot, \bot, \bot]\}$	$\{[m5, \bot, \bot, \bot]\}$
	B3	$\{[\bot, \bot, \bot, \bot]\}$	$\{[m1, \bot, \bot, \bot]\}$
	B4	$\{[\bot, \bot, \bot, \bot]\}$	$\{[\bot, \bot, m3, m4]\}$
	B5	$\{[\bot, \bot, \bot, \bot]\}$	$\{[\bot, m2, \bot, \bot]\}$
	B6	$\{[\bot, \bot, \bot, \bot]\}$	$\{[\bot, m2, m3, m4]\}$
	B7	$\{[\bot, \bot, \bot, \bot]\}$	$\{[\bot, m6, m7, m8]\}$
	B8	$\{[\bot, \bot, \bot, \bot]\}$	$\{[\bot, m6, m3, m4]\}$
	B9	$\{[\bot, \bot, \bot, \bot]\}$	$\{[\bot, \bot, \bot, m8]\}$
2	B1	$\{[\top, \top, \top, \top]\}$	$\{[m1, m2, m3, m4]\}$
	B2	$\{[m1, m2, m3, m4], [m1, \bot, \bot, m8]\}$	$\{[m5, m2, m3, m4], [m5, \bot, \bot, m8]\}$
	B3	$\{[m5, \bot, \bot, \bot]\}$	$\{[m1, \bot, \bot, \bot]\}$
	B4	$\{[m1, \bot, \bot, \bot]\}$	$\{[m1, \bot, m3, m4]\}$
	B5	$\{[m1, \bot, \bot, \bot], [\bot, \bot, m3, m4]\}$	$\{[m1, m2, \bot, \bot] [\bot, m2, m3, m4]\}$
	B6	$\{[m5, \bot, \bot, \bot]\}$	$\{[m5, m2, m3, m4]\}$
	B7	$\{[m5, \bot, \bot, \bot]\}$	$\{[m5, m6, m7, m7]\}$
	B8	$\{[\bot, m2, m3, m4], [\bot, m6, m7, m8]\}$	$\{[\bot, m6, m3, m4]\}$
	B9	$\{[\bot, m2, \bot, \bot], [\bot, m6, m3, m4]\}$	$\{[\bot, m2, \bot, m8], [\bot, m6, m3, m8]\}$
...	...	...	...
8	B1	$\{[\top, \top, \top, \top]\}$	$\{[m1, m2, m3, m4]\}$
	B2	$\{[m1, m2, m3, m4], [m1, m2, m3, m8], [m5, m6, m3, m8]\}$	$\{[m5, m2, m3, m4], [m5, m2, m3, m8], [m5, m6, m3, m8]\}$
	B3	$\{[m5, m2, m3, m4], [m5, m2, m3, m8], [m5, m6, m3, m8]\}$	$\{[m1, m2, m3, m4], [m1, m2, m3, m8], [m1, m6, m3, m8]\}$
	B4	$\{[m1, m2, m3, m4], [m1, m2, m3, m8] [m1, m6, m3, m8]\}$	$\{[m1, m2, m3, m4], [m1, m6, m3, m4]\}$
	B5	$\{[m1, m2, m3, m4], [m1, m2, m3, m8], [m1, m6, m3, m8], [m1, m6, m3, m4]\}$	$\{[m1, m2, m3, m4], [m1, m2, m3, m8]\}$
	B6	$\{[m5, m2, m3, m4], [m5, m2, m3, m8], [m5, m6, m3, m8]\}$	$\{[m5, m2, m3, m4]\}$
	B7	$\{[m5, m2, m3, m4], [m5, m2, m3, m8], [m5, m6, m3, m8]\}$	$\{[m5, m6, m7, m8]\}$
	B8	$\{[m5, m2, m3, m4], [m5, m6, m7, m8]\},$	$\{[m5, m6, m3, m4]\}$
	B9	$\{[m1, m2, m3, m4], [m1, m2, m3, m8], [m5, m6, m3, m4]\}$	$\{[m1, m2, m3, m8], [m5, m6, m3, m8]\}$
9	B1	$\{[\top, \top, \top, \top]\}$	$\{[m1, m2, m3, m4]\}$
	B2	$\{[m1, m2, m3, m4], [m1, m2, m3, m8], [m5, m6, m3, m8]\}$	
	B3	$\{[m5, m2, m3, m4], [m5, m2, m3, m8], [m5, m6, m3, m8]\}$	
	B4	$\{[m1, m2, m3, m4], [m1, m2, m3, m8] [m1, m6, m3, m8]\}$	
	B5	$\{[m1, m2, m3, m4], [m1, m2, m3, m8], [m1, m6, m3, m8], [m1, m6, m3, m4]\}$	
	B6	$\{[m5, m2, m3, m4], [m5, m2, m3, m8], [m5, m6, m3, m8]\}$	
	B7	$\{[m5, m2, m3, m4], [m5, m2, m3, m8], [m5, m6, m3, m8]\}$	
	B8	$\{[m5, m2, m3, m4], [m5, m6, m7, m8]\},$	
	B9	$\{[m1, m2, m3, m4], [m1, m2, m3, m8], [m5, m6, m3, m4]\}$	

(c) Reaching and leaving cache states

Figure 6: Computing all possible reaching cache states using the NUS approach.

in Figure 6(c), at the fix-point, $RCS^9_{B8} = \{[m5, m2, m3, m4], [m5, m6, m7, m8]\}$ and $BI(B8) = [\mp, m6, m3, m4]$. Then, $mc(B8, [m5, m2, m3, m4]) = 1$ and $mc(B8, [m5, m2, m3, m4]) = 2$. Thus, $wmc(B8, RCS_B 8^9) = 2$ and $bmc(B8, RCS_B 8^9) = 1$.

In summary, we have solved the first sub problem of cache analysis (Definition 6) using the fixed-point computation (Algorithm 1). For the second sub problem (Definition 6), we compute the number of cache misses for the worst and the best case using Definitions 4 and Definition 5.

4 The Absint approach

In the Absint approach [4], the notion of cache states is different from that of the NUS approach. We first describe the notion of *abstract cache states* used during the Absint approach.

4.1 Abstract cache states

An abstract cache state describes the possible combination of instructions in the cache at some instance in time. It is described as a vector $[set_0, set_1, \ldots, set_{N-1}]$ where each set_i represents a set of instructions, and the set contains a possible instruction contained in cache line $c_i \in C$ of a cache state.

For the CFG shown in Figure 2(a), when the program starts executing, we assume the cache to be empty. This is represented as $[\{\top\}, \{\top\}, \{\top\}, \{\top\}]$. Instructions are loaded into the cache as the basic blocks are executed (starting from the initial block). E.g., After executing the basic block $B1$ the abstract cache state is $[\{m1\}, \{m2\}, \{m3\}, \{m4\}]$.

Generally, the status of the cache may be *unknown* when a block is being analysed. In the Absint approach, we use the symbol $\perp$ to represent an unknown instruction in a cache line. E.g., $acs_\perp = [\{\perp\}, \{\perp\}, \{\perp\}, \{\perp\}]$ represents the status of every cache line is unknown. This notion of unknown cache state is used during the cache analysis and is explained later.

We now define the abstract cache states.

Definition 10 (Abstract cache state)**.** *Given a cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$, an abstract cache state $acs \subseteq (2^{I \cup \{\top, \perp\}})^N$, can be represented as a vector $[inst_0, inst_1, \ldots, inst_{N-1}]$ where each element $acs[i]$ represents a set of possible instructions in cache line c_i. Further, we restrict cache states such that, for any $i \in [0, N-1]$, $acs[i] \subseteq CI(i) \cup \{\top, \perp\}$. We denote $acs_\top = [\{\top\}, \{\top\}, \ldots, \{\top\}]$ as the empty cache state, where all the elements of acs are $\{\top\}$. Similarly, we define $acs_\perp = [\{\perp\}, \{\perp\}, \ldots, \{\perp\}]$, where all the elements of acs are $\{\perp\}$. Also, the set of all possible abstract cache states are referred to as ACS.*

Before we illustrate abstract cache states, we introduce two key terms essential to cache analysis. For any basic block b, an *abstract reaching cache state* represents the abstract state of the cache prior to the execution of b. *Abstract leaving cache state* represents the abstract state of the cache after the execution of the basic block. We denote the abstract reaching cache state of a basic block b as $arcs_b$ and the abstract leaving cache state of a basic block b as $alcs_b$.

Illustration of abstract cache states

Using Figure 7, we illustrate abstract reaching/leaving cache states of a block. Initially the cache is empty. The state of the cache is $acs_\top = [\{\top\}, \{\top\}, \{\top\}, \{\top\}]$. After the execution of block $B1$, cache lines c_0, c_1, c_2, c_3 will contain instructions $m1$, $m2$, $m3$, $m4$ respectively, resulting in the $acs = [\{m1\}, \{m2\}, \{m3\}, \{m4\}]$. The abstract cache state $[\{\top\}, \{\top\}, \{\top\}, \{\top\}]$ is the abstract state of the cache prior to the execution of the basic block $B1$. Thus, $arcs_{B1} = \{[\{\top\}, \{\top\}, \{\top\}, \{\top\}]\}$ is the abstract reaching cache state of block $B1$. Similarly, $alcs_{B1} = [\{m1\}, \{m2\}, \{m3\}, \{m4\}]$ is the abstract leaving cache state of block $B1$.

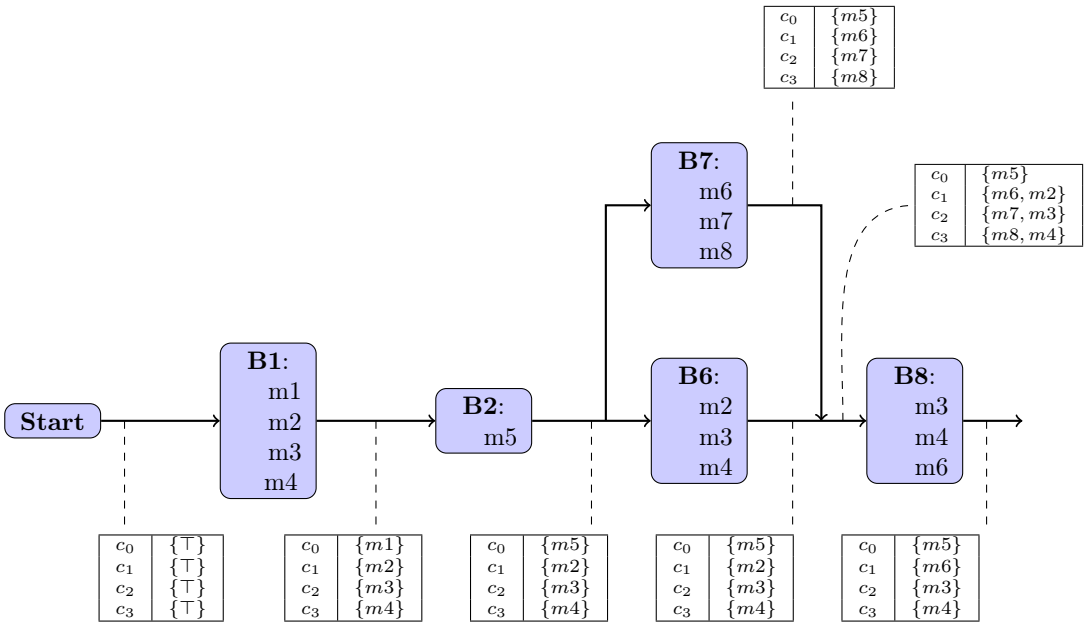


Figure 7: Illustration of the abstract cache states.

The abstract leaving cache state of $B1$ is also the abstract reaching cache state of $B2$, $arcs_{B2} = alcs_{B1}$. Block $B2$ has only one instruction $m5$, and is mapped to cache line c_0. Thus, the previous instruction in line c_0 will be replaced by $m5$. Since the $arcs_{B2} = [\{m1\}, \{m2\}, \{m3\}, \{m4\}]$, after executing the instructions in $B2$, the resulting abstract leaving cache state $alcs_{B2} = [\{m5\}, \{m2\}, \{m3\}, \{m4\}]$.

Now, the control reaches a branch due to which, it is possible to execute either block $B6$ or block $B7$. In this case, $arcs_{B6} = alcs_{B2} = arcs_{B7}$. After executing blocks $B6$ or $B7$, the state of the cache is either $[\{m5\}, \{m2\}, \{m3\}, \{m4\}]$ or $[\{m5\}, \{m6\}, \{m7\}, \{m8\}]$.

Block $B8$ has two incoming edges: from blocks $B6$ and $B7$. To compute $arcs_{B8}$, we need to *join* $alcs_{B6}$ and $alcs_{B7}$. In this case, the join function is a pair-wise union over the vector elements. Thus, $arcs_{B8}[i] = alcs_{B6}[i] \cup alcs_{B7}[i]$, resulting in $arcs_{B8} = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m4, m8\}]$. After execution of block $B8$, the resulting leaving cache state is $[\{m5\}, \{m6\}, \{m3\}, \{m4\}]$.

4.2 The Absint join function

The Absint join function computes the abstract reaching cache state of a basic block when the block has more than one incoming edges. E.g., in Figure 7, the abstract reaching cache state of Block $B8$ has two incoming edges: from blocks $B6$ and $B7$. To compute $arcs_{B8}$ we need to *join* $alcs_{B6}$ and $alcs_{B7}$. It is possible that during the computation of the abstract reaching cache state of a block b, we may need to *join* two abstract leaving cache states that contain unknown instructions for some cache lines. E.g., $alcs_1 = [\{m1\}, \{\bot\}, \{m2\}, \{\bot\}]$ and $alcs_2 = [\{m1\}, \{\bot\}, \{\bot\}, \{\bot\}]$. The join function could be described as an union over the abstract reaching cache states, i.e., $arcs_b = alcs_1 \cup alcs_2 = [\{m1\}, \{\bot\}, \{m2, \bot\}, \{\bot\}]$. However, as described in Section 4.1, in the Absint approach an unknown instruction $\bot$ is subsumed by any instruction. E.g., $[\{m1\}, \{\bot\}, \{m2, \bot\}, \{\bot\}]$ becomes $[\{m1\}, \{\bot\}, \{m2\}, \{\bot\}]$. This join

function for the Absint approach is defined in Definition 11.

Definition 11 (Absint join function)**.** *We define the join function* $J_{Absint} : ACS \times ACS \rightarrow ACS$ *where for any two abstract cache states* $acs_1 \in ACS$ *and* $acs_2 \in ACS$*,*

$$J_{Absint}(acs_1, acs_2) = acs_3$$

where for all cache lines $c_i \in C$ *where* $i \in [0, N-1]$

$$acs_3[i] = \begin{cases} acs_1[i] & : if \quad acs_2[i] = \bot \\ acs_2[i] & : if \quad acs_1[i] = \bot \\ acs_1[i] \cup acs_2[i] & : otherwise \end{cases}$$

Given two abstract cache states $acs_1 = [\{\bot\}, \{m2\}, \{m3\}, \{\bot\}]$ and $acs_2 = [\{m5\}, \{m6\}, \{m7\}, \{\bot\}]$, then, $J_{Absint}(acs_1, acs_2) = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{\bot\}]$. The union of sets that correspond to the same cache lines.

4.3 The Absint transfer function

The Absint transfer function describes how an abstract reaching cache state of a basic block is transformed into an abstract leaving cache state, after executing the instructions in the basic block. E.g., in Figure 7, the abstract reaching cache state of block $B2$ is $arcs_{B2} = [\{m1\}, \{m2\}, \{m3\}, \{m4\}]$ and after the execution of block $B2$, where the instructions of the block is described by $BI(B2) = [m5, \mp, \mp, \mp]$, the abstract leaving cache state of $B2$ is $alcs_{B2} = [\{m5\}, \{m2\}, \{m3\}, \{m4\}]$. This transformation can be described as follows. Firstly, if a block b does not have an instructions mapped to a particular cache line c_i ($BI(b)[i] = \mp$), then after execution of the block, the contents of the cache corresponding to that cache line remains the same. E.g., basic block $B2$ has no instruction mapped to the cache line c_1, ($BI(B2)[1] = \mp$), thus, after execution, the abstract leaving cache state has the same set of instructions as in the abstract reaching cache state $alcs_{B2}[1] = \{m2\} = arcs_{B2}$. Secondly, if the block has an instruction mapped to a particular cache line ($BI(b)[i] \neq \mp$), then, after execution of the block, the content of the cache corresponding to that cache line is changed to the instruction executed by the block. E.g., basic block $B2$ has the instruction $m5$ mapped to the cache line c_0, ($BI(B2)[0] \neq \mp$), thus, after execution, the abstract leaving cache state has the instruction $m5$, $alcs_{B2}[0] = \{m5\} = \{BI(B2)[0]\}$. The transfer function, T_{Absint}, formalises this transformation as follows.

Definition 12 (Absint transfer function)**.** *The Absint transfer function is* $T_{Absint} : B \times ACS \rightarrow ACS$ *where, for a given basic block* $b \in B$ *and an abstract cache states* $acs \in ACS$*,*

$$T_{Absint}(b, acs) = acs'$$

where for all cache lines $c_i \in C$ *where* $i \in [0, N-1]$ *then,*

$$acs'[i] = \begin{cases} acs[i] & : if \quad BI(b)[i] = \mp \\ BI(b)[i] & : otherwise \end{cases}$$

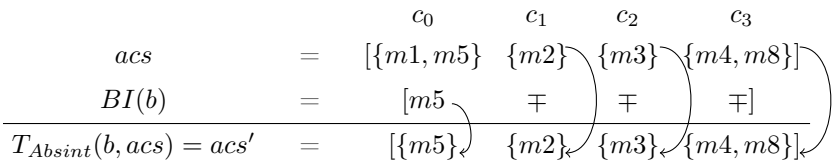


Figure 8: Illustration of the Absint transfer function

Illustration of the transfer function T_{Absint}

Figure 8 illustrates the operation of the transfer function T_{Absint}. Given block b, with $BI(b) = [m5, \mp, \mp, \mp]$, and an abstract reaching cache state $acs = [\{m1, m5\}, \{m2\}, \{m3\}, \{m4, m8\}]$, the abstract leaving cache state $cs' = T_{Absint}(b, acs)$ is computed as follows.

For cache line c_0, block b executes an instruction $m5$ on cache line c_0 ($BI(b)[0] = m5$). Thus, irrespective of the reaching cache state, the leaving cache (acs') contains only the instruction $m5$ on its cache line c_0, i.e., $cs'[0] = \{m5\} = \{BI(b)[0]\}$.

For the cache line c_1, block b has no instruction mapped to cache line c_1 ($BI(b)[1] = \mp$). Thus, the instructions set $\{m2\}$ on cache line c_1 in acs ($cs[1] = \{m2\}$), is copied to the cache line c_1 of the leaving cache state acs' i.e., $acs'[1] = \{m2\} = acs[1]$. Similarly, block b has no instructions mapped to cache lines c_2 and c_3, resulting in $acs'[2] = \{m3\} = cs[2]$ and $acs'[3] = \{m4, m8\} = acs[3]$. Thus, given block b, with $BI(b) = [m5, \mp, \mp, \mp]$ and a reaching cache state $acs = [\{m1, m5\}, \{m2\}, \{m3\}, \{m4, m8\}]$, $T_{Absint}(b, acs) = acs' = [\{m5\}, \{m2\}, \{m3\}, \{m4, m8\}]$.

4.4 Fixed-point computation

Cache analysis starts by the computation of abstract reaching cache state for every block $b \in B$. The Absint approach [4] solves this problem using a fixed-point computation algorithm presented in Algorithm 2.

In Algorithm 2, we first initialise the abstract reaching cache states for all blocks (lines 3 to 9, Algorithm 2). Since we assume that initially the state of the cache is empty, on line 5 for the initial block b_0 we set its reaching as $arcs^1_{b_0} = \{acs_\top\}$. Here, the notation $arcs^i_b$ represents the abstract reaching cache state of block b in iteration i. E.g., $arcs^1_{b_0}$ represents the reaching cache states of block b_0 for iteration 1. For the rest of the blocks, the execution of the CFG will impact the state of the cache, the state of the cache is unknown. Thus, on line 7, we set their reaching cache states as $arcs^1_b = \{acs_\bot\}$.

After initialisation, we compute the abstract leaving cache states of each block, on lines 11 to 14. We apply the transfer function T_{UoA} to every block and its corresponding abstract reaching cache states.

The iteration index (i) is incremented (line 15) to signal the start of the next iteration. Next, on lines 16 to 26, the abstract reaching cache states of each block are computed. For the initial block b_0, we know that the abstract reaching cache state is always empty. Thus, on line 19, we always set its reaching as $arcs^i_{b_0} = \{acs_\top\}$. For rest of the blocks, on lines 22 to 24, the reaching cache states are computed by looking at the leaving cache states of the predecessors (b') of the block b and using the Absint join function.

The iterative process, repeat-until loop on lines 10 to 27, is repeated until a fixed point is reached, i.e., if two consecutive iterations have the same sets of abstract reaching cache states for all blocks (line 27).

Algorithm 2 Fixed-point computation for the Absint approach

Input: A cache model $CM = \langle I,\ C,\ CI,\ G,\ BI \rangle$
Output: $arcs_b$ for every block $b \in B$.

1: $i = 1$ {iteration counter}
2: {Initialise the arcs of all blocks}
3: **for each** $b \in B$ **do**
4: **if** $b = b_0$ **then**
5: $arcs_b^i = acs_\top$
6: **else**
7: $arcs_b^i = acs_\bot$
8: **end if**
9: **end for**

10: **repeat**
11: {Compute the set of leaving cache states for all blocks for iteration i}
12: **for each** $b \in B$ **do**
13: $alcs_b^i = T_{Absint}(b, arcs_b^i)$
14: **end for**

15: $i = i + 1$; {Next iteration}
16: {Compute $arcs$ }
17: **for each** $b \in B$ **do**
18: **if** $b = b_0$ **then**
19: $arcs_b^i = acs_\top$
20: **else**
21: $arcs_b^{i+1} = [\emptyset, \emptyset, \ldots, \emptyset]$ {Empty vector}
22: **for each** $alcs_{b'}^i$, where $(b', b) \in E$ **do**
23: $arcs_b^{i+1} = J_{Absint}(arcs_b^{i+1}, alcs_{b'})$
24: **end for**
25: **end if**
26: **end for**

27: **until** $\forall b \in B$, $arcs_b^i = arcs_b^{i-1}$ {Termination condition}
28: **return** $arcs_b^i$ for all $b \in B$

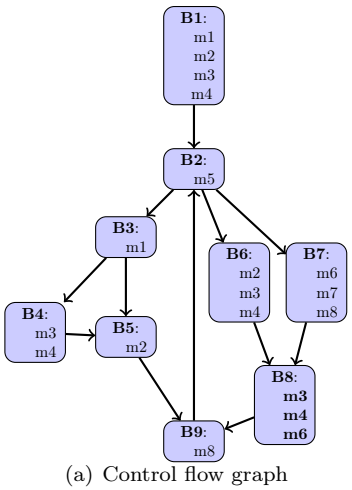
4.4.1 Illustration of the fixed-point algorithm

We illustrate the fixed-point algorithm using the Figure 9. The CFG and the mapping of the instructions to cache lines are presented in Figure 9(a) and Figure 9(b) respectively. Figure 9(c) presents a table showing the abstract reaching and abstract leaving cache states for all blocks, during each iteration of the algorithm.

We first initialise the abstract reaching cache states for all blocks (lines 3 to 9). For the CFG in Figure 9(a), the initial block is b_0 is $B1$. Since we assume that initially the state of the cache is empty, we set $arcs_{B1}^1$ as $acs_\top = [\{\top\}, \{\top\}, \{\top\}, \{\top\}]$. For the rest of the blocks, the state of the cache is unknown. Thus, we set their reaching cache states as $acs_\bot = [\{\bot\}, \{\bot\}, \ldots, \{\bot\}]$.

After initialisation, to compute the abstract leaving cache state of each block, we apply the transfer function T_{Absint} (lines 11 to 14). For example, in iteration 1 ($i = 1$), $alcs_{B1}^1$ $= T_{Absint}(B1, arcs_{B1}^1) = T_{Absint}(B1, [\{\top\}, \{\top\}, \{\top\}, \{\top\}]) = [\{m1\}, \{m2\}, \{m3\}, \{m4\}]$. For block $B2$, which has one instruction $m5$ that is mapped to line c_0, the leaving cache states $alcs_{B2}^1 = T_{NUS}(B2, [\{\bot\}, \{\bot\}, \{\bot\}, \{\bot\}]) = [\{m5\}, \{\bot\}, \{\bot\}, \{\bot\}]$. Similarly, the leaving cache states (column 4) for rest of the blocks are computed for iteration 1 as shown in Figure 9(c).

The iteration index (i) is incremented (line 15) to signal the start of the next iteration ($i = 2$).



(a) Control flow graph

Cache line	Instructions
c_0	m1,m5
c_1	m2,m6
c_2	m3,m7
c_3	m4,m8

(b) Instructions to cache line mapping

May analysis			
Ite. (i)	**Block** (b)	**Abstract reaching cache states** ($arcs_b^i$)	**Abstract leaving cache states** ($alcs_b^i$)
1	B1	$[\{\top\},\{\top\},\{\top\},\{\top\}]$	$[\{m1\},\{m2\},\{m3\},\{m4\}]$
	B2	$[\{\bot\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{m5\},\{\bot\},\{\bot\},\{\bot\}]$
	B3	$[\{\bot\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{m1\},\{\bot\},\{\bot\},\{\bot\}]$
	B4	$[\{\bot\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{\bot\},\{\bot\},\{m3\},\{m4\}]$
	B5	$[\{\bot\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{\bot\},\{m2\},\{\bot\},\{\bot\}]$
	B6	$[\{\bot\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{\bot\},\{m2\},\{m3\},\{m4\}]$
	B7	$[\{\bot\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{\bot\},\{m6\},\{m7\},\{m8\}]$
	B8	$[\{\bot\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{\bot\},\{m6\},\{m3\},\{m4\}]$
	B9	$[\{\bot\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{\bot\},\{\bot\},\{\bot\},\{m8\}]$
2	B1	$[\{\top\},\{\top\},\{\top\},\{\top\}]$	$[\{m1\},\{m2\},\{m3\},\{m4\}]$
	B2	$[\{m1\},\{m2\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m2\},\{m3\},\{m4,m8\}]$
	B3	$[\{m5\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{m1\},\{\bot\},\{\bot\},\{\bot\}]$
	B4	$[\{m1\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{m1\},\{\bot\},\{m3\},\{m4\}]$
	B5	$[\{m1\},\{\bot\},\{m3\},\{m4\}]$	$[\{m1\},\{m2\},\{m3\},\{m4\}]$
	B6	$[\{m5\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{m5\},\{m2\},\{m3\},\{m4\}]$
	B7	$[\{m5\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{m5\},\{m6\},\{m7\},\{m8\}]$
	B8	$[\{\bot\},\{m2,m6\},\{m3,m7\},\{m4,m8\}]$	$[\{\bot\},\{m6\},\{m3\},\{m4\}]$
	B9	$[\{\bot\},\{m2,m6\},\{m3\},\{m4\}]$	$[\{\bot\},\{m2,m6\},\{m3\},\{m8\}]$
3	B1	$[\{\top\},\{\top\},\{\top\},\{\top\}]$	$[\{m1\},\{m2\},\{m3\},\{m4\}]$
	B2	$[\{m1\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$
	B3	$[\{m5\},\{m2\},\{m3\},\{m4,m8\}]$	$[\{m1\},\{m2\},\{m3\},\{m4,m8\}]$
	B4	$[\{m1\},\{\bot\},\{\bot\},\{\bot\}]$	$[\{m1\},\{\bot\},\{m3\},\{m4\}]$
	B5	$[\{m1\},\{\bot\},\{m3\},\{m4\}]$	$[\{m1\},\{m2\},\{m3\},\{m4\}]$
	B6	$[\{m5\},\{m2\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m2\},\{m3\},\{m4\}]$
	B7	$[\{m5\},\{m2\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m6\},\{m7\},\{m8\}]$
	B8	$[\{m5\},\{m2,m6\},\{m3,m7\},\{m4,m8\}]$	$[\{m5\},\{m6\},\{m3\},\{m4\}]$
	B9	$[\{m1\},\{m2,m6\},\{m3\},\{m4\}]$	$[\{m1\},\{m2,m6\},\{m3\}\{m8\}]$
4	B1	$[\{\top\},\{\top\},\{\top\},\{\top\}]$	$[\{m1\},\{m2\},\{m3\},\{m4\}]$
	B2	$[\{m1\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$
	B3	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m1\},\{m2,m6\},\{m3\},\{m4,m8\}]$
	B4	$[\{m1\},\{m2\},\{m3\},\{m4,m8\}]$	$[\{m1\},\{m2\},\{m3\},\{m4\}]$
	B5	$[\{m1\},\{m2\},\{m3\},\{m4,m8\}]$	$[\{m1\},\{m2\},\{m3\},\{m4,m8\}]$
	B6	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m2\},\{m3\},\{m4\}]$
	B7	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m6\},\{m7\},\{m8\}]$
	B8	$[\{m5\},\{m2,m6\},\{m3,m7\},\{m4,m8\}]$	$[\{m5\},\{m6\},\{m3\},\{m4\}]$
	B9	$[\{m1,m5\},\{m2,m6\},\{m3\},\{m4\}]$	$[\{m1,m5\},\{m2,m6\},\{m3\},\{m8\}]$
5	B1	$[\{\top\},\{\top\},\{\top\},\{\top\}]$	$[\{m1\},\{m2\},\{m3\},\{m4\}]$
	B2	$[\{m1,m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$
	B3	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m1\},\{m2,m6\},\{m3\},\{m4,m8\}]$
	B4	$[\{m1\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m1\},\{m2,m6\},\{m3\},\{m4\}]$
	B5	$[\{m1\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m1\},\{m2\},\{m3\},\{m4,m8\}]$
	B6	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m2\},\{m3\},\{m4\}]$
	B7	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m5\},\{m6\},\{m7\},\{m8\}]$
	B8	$[\{m5\},\{m2,m6\},\{m3,m7\},\{m4,m8\}]$	$[\{m5\},\{m6\},\{m3\},\{m4\}]$
	B9	$[\{m1,m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	$[\{m1,m5\},\{m2,m6\},\{m3\},\{m8\}]$
6	B1	$[\{\top\},\{\top\},\{\top\},\{\top\}]$	
	B2	$[\{m1,m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	
	B3	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	
	B4	$[\{m1\},\{m2,m6\},\{m3\},\{m4,m8\}]$	
	B5	$[\{m1\},\{m2,m6\},\{m3\},\{m4,m8\}]$	
	B6	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	
	B7	$[\{m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	
	B8	$[\{m5\},\{m2,m6\},\{m3,m7\},\{m4,m8\}]$	
	B9	$[\{m1,m5\},\{m2,m6\},\{m3\},\{m4,m8\}]$	

(c) Reaching and leaving cache states

Figure 9: Computing all possible abstract reaching cache states using the Absint approach.

Next, the abstract reaching cache states of each block,for iteration $i = 2$, are computed (lines 16 to 26). For the initial block $B1$, we know that the abstract reaching cache state is always $acs_\top$. Thus, on line 19, we always set its abstract reaching cache state as $RCS^2_{B1} = \{cs_\top\}$. For rest of the blocks, on lines 22 to 24, the reaching cache states are computed by looking at the abstract leaving cache states of the predecessors (b') of the block b and using the Absint join function. E.g., the predecessors of $B2$ are $B1$ and $B9$, $b' \in \{B1, B9\}$, and from the previous iteration ($i = 1$) the $alcs^1_{B1} = [\{m1\}, \{m2\}, \{m3\}, \{m4\}]$ and $alcs^1_{B9} = [\{\bot\}, \{\bot\}, \{\bot\}, \{m8\}]$. The process for computing $arcs^2_{B2}$ is as follows.

1. Initialise $arcs^2_{B2} = [\emptyset, \emptyset, \ldots, \emptyset]$ (line 21).
2. Using the Absint join function we process the *alcs* of each predecessor block ($B1, B9$), on lines 22 to 24, on at a time (in no particular order). For illustration we will first analyse $alcs^1_{B1}$ followed by $alcs^1_{B9}$ as follows.

$$\begin{aligned} arcs^2_{B2} &= J_{UoA}(arcs^2_{B2}, alcs^1_{B1}) \\ &= J_{UoA}([\emptyset, \emptyset, \ldots, \emptyset], [\{m1\}, \{m2\}, \{m3\}, \{m4\}]) \\ &= \{[\{m1\}, \{m2\}, \{m3\}, \{m4\}]\} \\ arcs^2_{B2} &= J_{UoA}(arcs^2_{B2}, alcs^1_{B9}) \\ &= J_{UoA}([\{m1\}, \{m2\}, \{m3\}, \{m4\}], [\{\bot\}, \{\bot\}, \{\bot\}, \{m8\}]) \\ &= \{[\{m1\}, \{m2\}, \{m3\}, \{m4, m8\}]\} \end{aligned}$$

The iterative process, repeat-until loop on lines 10 to 27, is repeated until a fixed point is reached, i.e., if two consecutive iterations have a same set of abstract reaching cache state for all blocks (line 27). For our example, it happens in the 6^{th} iteration, where the same set of abstract reaching cache states are computed for all blocks. As a result, the reaching cache states of the 6^{th} iteration represents all the possible abstract reaching cache states of the program. Note that while the NUS approach reached a fixed point after 9 iterations, the Absint approach achieved a fixed point after 6 iterations, when applied over the same example.

Also, note that during the first iteration, we assumed the reaching cache states for blocks $B2$ to $B9$ as unknown ($[\{\bot\}, \{\bot\}, \ldots, \{\bot\}]$), but when we reach the fixed-point, all the unknown cache states are resolved.

Thus, we have presented how to computes all possible abstract reaching cache states *arcs* for every block in a given cache model. However, to solve the first sub problem (Definition 6) which involves the computation all the possible reaching cache states (RCS) for every basic block in the given cache model, we need to translate the *arcs* in to corresponding RCS.

4.4.2 Mapping an abstract cache state to cache states

An abstract cache state (*acs*) contains the possible set of instructions that can be present in the line c_i ($acs[i]$). If we simply compute the cross product of these sets, we can compute the set of all cache states. For example, given the $arcs = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m8\}]$, the cross product is,

$$\{[m5, m2, m3, m8], [m5, m2, m7, m8], [m5, m6, m3, m8], [m5, m6, m7, m8]\}$$

Using Definition 13, we define the function that describes the above mapping.

Definition 13 (Abstract cache state to cache states mapping)**.** *We define a mapping function* $MAP_{Absint} : ACS \rightarrow 2^{CS}$ *where, for a given abstract reaching cache states* $acs \in ACS$ *is,*

$$MAP_{Absint}(acs) = acs[0] \times acs[1] \times \ldots \times acs[N-1]$$

For example, given an $arcs = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m8\}]$, then

$$\begin{aligned} MAP_{Absint}(arcs) &= arcs[0] \times arcs[1] \times arcs[2] \times arcs[3] \\ &= \{m5\} \times \{m2, m6\} \times \{m3, m7\} \times \{m8\} \\ &= \{[m5, m2, m3, m8], [m5, m2, m7, m8], [m5, m6, m3, m8], [m5, m6, m7, m8]\} \end{aligned}$$

By mapping an abstract cache sate to a set of cache states, we have solved the first sub problem (Definition 6) which involves the computation all the possible reaching cache states (RCS) for every basic block in the given cache model.

4.5 Calculating cache misses

The second sub problem of cache analysis (Definition 6) involves the computation of the number of cache misses for the best case and the worst case of every basic block. We present two solutions to this problem.

First, after translating $arcs$ (computed using Algorithm 2) into RCS (as described in Definition 13), using the functions wmc and bmc (presented in Definitions 4 and Definition 5), the number of cache misses in the worst and the best case can be computed.

Second, the number of cache misses could be computed by directly analysing the abstract cache state $arcs$. An abstract cache state $arcs$ describes the possible combination of instructions in the cache at some instance in time. $arcs[i]$ describes the possible instructions that can be present on the cache line c_i. For example, consider the abstract reaching cache state $arcs = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m8\}]$. For the cache line c_1, $arcs[1] = \{m2, m6\}$ describes that the cache either contains the instruction $m2$ or $m6$. $BI(b)[i]$ represents the instruction that block b executes cache line c_i. E.g., consider the block $B8$ with $BI(B8) = [\mp, m6, m3, m4]$. Then, $BI(B8)[1] = m6$ describes that block $B8$ needs the instruction $m6$ on the cache line c_1. We now describe how to compute the number of cache misses for the worst case and the best case.

Worst case analysis: A cache miss can occur on cache line c_i, if b has an instruction ($BI(b)[i] \neq \mp$), and $BI(b)[i]$ is not the only instruction contained in the abstract reaching cache state on cache line c_i ($\{BI(b)[i]\} \neq arcs[i]$). For the above example, the block $B8$ executes the instruction $m6$ on cache line $c1$, whereas the cache may contain either $m2$ or $m6$. Hence, for a cache line c_1, in the worst case, we have a cache *miss* ($\{BI(B8)[1]\} = \{m6\} \neq \{m2, m6\} = arcs[1]$).

Best case analysis: A cache miss can occur on cache line c_i, if b has an instruction ($BI(b)[i] \neq \mp$), and $BI(b)[i]$ is not present in the abstract reaching cache state on cache line c_i ($BI(b)[i] \notin arcs[i]$). For the above example, the block $B8$ executes the instruction $m4$ on cache line $c8$, whereas the cache contains only $m8$. Hence, for a cache line c_3, in the best case, we have a cache *miss* ($BI(B8)[3] = m4 \notin \{m8\} = arcs[3]$).

Using Definitions 14 and 15, we define the functions that compute the number of cache misses for the worst and the best case for a block directly from an abstract (reaching) cache state.

Definition 14 (Absint worst miss count)**.** *We define the Absint worst miss count function* $wmc_{Absint} : B \times ACS \to \mathbb{N}^0$ *where, for a given block* $b \in B$*, an abstract cache state* $acs \in ACS$ *is,*

$$wmc_{Absint}(b, acs) = \sum_{i=0}^{N-1} wmiss_i$$

where,

$$wmiss_i = \begin{cases} 1 & : if \quad BI(b)[i] \neq \mp \; and \; \{BI(b)[i]\} \neq acs[i] \\ 0 & : otherwise \end{cases}$$

Definition 15 (Absint best miss count)**.** *We define the Absint best miss count function* $bmc_{Absint} : B \times ACS \rightarrow \mathbb{N}^0$ *where, for a given block* $b \in B$ *and an abstract cache state* $acs \in ACS$ *is,*

$$bmc_{Absint}(b, acs) = \sum_{i=0}^{N-1} bmiss_i$$

where,

$$wmiss_i = \begin{cases} 1 & : if \quad BI(b)[i] \neq \mp \; and \; BI(b)[i] \notin acs[i] \\ 0 & : otherwise \end{cases}$$

Illustration

Given a $acs = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m8\}]$ and $BI(B8) = [\mp, m6, m3, m4]$, using Fig-

c_i		c_0	c_1	c_2	c_3	
acs	=	$[\{m5\}$	$\{m2, m6\}$	$\{m3, m7\}$	$\{m8\}]$	
$BI(B8)$	=	$[\ \mp$	$m6$	$m3$	$m4]$	
$wmiss_i$	=	0	1	1	1	$= 3 = wmc_{Absint}(B8, acs)$
$bmiss_i$	=	0	0	0	1	$= 1 = bmc_{Absint}(B8, acs)$

Figure 10: Illustration of the functions wmc_{Absint} and bmc_{Absint}

ure 10 we now illustrate the functions wmc_{Absint} and bmc_{Absint}.

During a worst case analysis, for cache line c_0, we do not have a cache miss, $wmiss_0 = 0$, because the block $B8$ does not have any instruction mapped to c_0 ($BI(B8)[0] = \mp$). For cache line c_1, we have a cache miss, $wmiss_1 = 1$, because the block needs to execute $m6$ and it is not the only possible instruction in the cache ($\{BI(B8)[1]\} = \{m6\} \neq \{m2, m6\} = acs[1]$). Similarly, $wmiss_2 = 0$, because $m3$ is not the only possible instruction in the cache ($\{BI(B8)[2]\} = \{m3\} \neq \{m3, m7\} = acs[2]$). Finally, for the cache line c_3, we have a cache miss, $wmiss_3 = 1$, because the block needs to execute $m4$, while the cache contains $m8$ ($\{BI(B8)[3]\} = \{m4\} \neq \{m8\} = acs[3]$). Thus, for the four cache lines, in the worst case, we get three misses ($wmc_{Absint}(B8, acs) = 0 + 1 + 1 + 1 = 3$).

During a best case analysis, for cache line c_0, we do not have a cache miss, $bmiss_0 = 0$, because the block $B8$ does not have any instruction mapped to c_0 ($BI(B8)[0] = \mp$). For cache line c_1, we have a possible cache hit, $wmiss_1 = 1$, because the block needs to execute $m6$ and it is present in the set of possible instruction in the cache ($\{BI(B8)[1]\} = \{m6\} \in \{m2, m6\} = acs[1]$). Similarly, for the cache line c_1, we have a possible cache hit $wmiss_2 = 1$, because $m3$ is present in the set of possible instruction in the cache ($\{BI(B8)[2]\} = \{m3\} \in \{m3, m7\} = acs[2]$). Finally, for the cache line c_3, we have a cache miss, $wmiss_3 = 1$, because the block needs to execute $m4$ and the instruction is not in the cache ($\{BI(B8)[3]\} = \{m4\} \notin \{m8\} = acs[3]$). Thus, for the four cache lines, in the best case, we get three misses ($bmc_{Absint}(B8, acs) = 0 + 0 + 0 + 1 = 1$).

In summary, we have solved the first sub problem of cache analysis (Definition 6) using the fixed-point computation (Algorithm 2) and the abstract cache state to cache states mapping function (Definition 13). For the second sub problem (Definition 6), we define two solutions (directly over $arcs$, or indirectly by mapping to CS) to compute the number of cache misses for the worst and the best case.

5 Comparison between NUS and Absint approaches

In this section, using Table 4, we present a comparison between the NUS and the Absint approaches. For the example (presented in Figure 2), column 2 presents the number of iterations required for reaching the fixed point computation. Column 3 presents the reaching cache states for block $B8$ computed during the NUS approach, and the abstract reaching cache states for block $B8$ computed during the Absint approach. Finally, the number of cache misses in the worst case and the best case are presented in columns 5 and 6 respectively. Approaches for solving the cache analysis problem may be compared on the basis of precision and the analysis time.

Approach	**Ite.**	$\boldsymbol{RCS_{B8}/arcs_{B8}}$	**Worst case**	**Best case**
NUS	9	$[m5, m2, m3, m4]$ $[m5, m6, m7, m8]$	2	1
Absint	6	$[\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m4, m8\}]$	3	0

Table 4: Comparing the precision between the NUS and the Absint approaches.

Precision
The NUS approach is more precise than the Absint approach, for both the worst case ($2 < 3$) and the best case ($1 > 0$) analysis.

Analysis time: The NUS approach is slower than the Absint approach. It takes 9 iterations to reach the fixed point, while the Absint approach takes only 6 iterations.

In general, *Precision* and *Analysis time* are complementary opposite to each other, i.e., If an approach yields high precision it is most likely will have high analysis time, and vice versa. The Absint approach, abstracts the relation between the cache lines during the *join* function (Definition 11), by doing a pair-wise union over elements of abstract reaching cache states. This, reduces the precision of the analysis. In contrast, the NUS approach during the *join* function (Definition 7), accumulates all cache states and does not lose any precision, but is much slower than the Absint approach.

In general, due to the scalability, the Absint approach is more practical and adopted by industry for successfully analysing large programs. However, some applications could significantly gain from more precise analysis. We present this idea in Figure 11. The ideal cache analysis approach should be as closer to the desired region as possible. Using this as a objective, in Section 6 we present our new cache analysis approach.

In Table 5 we present a summary of the symbols and the definitions that are presented in this section.

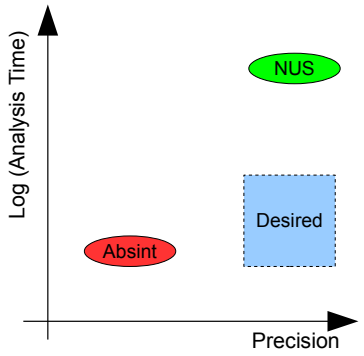


Figure 11: NUS and Absint approaches on precision vs analysis time.

Symbol/ Definition	**Description**	**Example**
$J_{NUS} : 2^{CS} \times 2^{CS} \rightarrow 2^{CS}$	NUS join function	$CS_1 = \{[m5, m2, m3, m4]\}$ and $CS_2 = \{[m5, m6, m7, m8]\}$, $J_{NUS}(CS_1, CS_2) = \{[m5, m2, m3, m4], [m5, m6, m7, m8]\}$
$T_{NUS} : B \times 2^{CS} \rightarrow 2^{CS}$	NUS transfer function	$cs_1 = [m5, m2, m3, m4]$, $cs_2 = [m5, m6, m7, m8]$, $RCS_{B8} = \{cs_1, cs_2\}$, $T_{NUS}(B8, \{cs_1, cs_2\}) = \{[m5, m6, m3, m4]\}$
acs	Abstract cache state	$[\{m1\}, \{m2\}, \{m3\}, \{m4\}]$
$arcs_b$	Abstract reaching cache state of block b	In Figure 7, $arcs_{B2} = [\{m1\}, \{m2\}, \{m3\}, \{m4\}]$
$alcs_b$	Abstract leaving cache state of block b	In Figure 7, $alcs_{B2} = [\{m5\}, \{m2\}, \{m3\}, \{m4\}]$
$J_{Absint} : ACS \times ACS \rightarrow ACS$	Absint join function	$acs_1 = [\{\perp\}, \{m2\}, \{m3\}, \{\perp\}]$, $acs_2 = [\{m5\}, \{m6\}, \{m7\}, \{\perp\}]$, then, $J_{Absint}(acs_1, acs_2) = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{\perp\}]$
$T_{Absint} : B \times ACS \rightarrow ACS$	Absint transfer function	$BI(b) = [m5, \mp, \mp, \mp]$, $acs = [\{m1\}, \{m2\}, \{m3\}, \{m4\}]$, $T_{Absint}(b, acs) = acs' = [\{m5\}, \{m2\}, \{m3\}, \{m4\}]$
$MAP_{Absint} : ACS \rightarrow 2^{CS}$	Abstract cache state to cache states mapping	$acs = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m8\}]$, then $MAP_{Absint}(acs) = \{[m5, m2, m3, m8], [m5, m2, m7, m8], [m5, m6, m3, m8], [m5, m6, m7, m8]\}$
$wmc_{Absint} : B \times ACS \rightarrow \mathbb{N}^0$	Absint worst miss count	$acs = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m8\}]$ and $BI(B8) = [\mp, m6, m3, m4]$ then, $wmc_{Absint}(B8, acs) = 0 + 1 + 1 + 1 = 3$
$bmc_{Absint} : B \times ACS \rightarrow \mathbb{N}^0$	Absint best miss count	$acs = [\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m8\}]$ and $BI(B8) = [\mp, m6, m3, m4]$ then, $bmc_{Absint}(B8, acs) = 0 + 0 + 0 + 1 = 1$

Table 5: Some of the symbols and the definitions presented in this section.

6 Proposed approach

6.1 Overview

The NUS approach (see Section 3) and the Absint approach (see Section 4) solve the cache analysis problem (Definition 6) by analysing all basic blocks in a single fixed point computation. However, if we analyse only one block (called the reference block) at a time, we can ignore blocks that execute instructions which do not affect the precision (number of cache hits/misses) of the reference block. Secondly, we can abstract the instructions in the cache state w.r.t. the instructions executed by the reference block. This may significantly reduce the number of possible cache states, and may reduce the number of iterations required during cache analysis, resulting in faster analysis time compared to the NUS approach. These are the basis of our proposed approach. Importantly, our abstraction is unlike the Absint approach, and the precision is the same as the NUS approach. Our intuition for analysing each block individually is based on the following two observations.

Observation 1 : *A reference block b_{ref} may not refer to all the cache lines during its execution.* The unrefereed cache lines, can be ignored during the analysis of b_{ref}. Further, we can remove vacuous blocks from the graph, if all their instruction(s) are mapped to the unrefereed cache lines (w.r.t b_{ref}). For example, given $BI(B8) = [\mp, m6, m3, m4]$, $BI(B1) = [m1, m2, m3, m4]$ and $BI(B2) = [m5, \mp, \mp, \mp]$, during the analysis of block $B8$ ($b_{ref} = B8$), we observe that $B8$ does not execute any instruction on the cache line c_0 ($BI(B8)[0] = \mp$). Thus, we can ignore the instructions $m1$ and $m5$ that are executed by blocks $B1$ and $B2$. This leaves block $B2$ with its only instruction being mapped to the unrefereed cache line c_0. Hence, $B2$ is vacuous, and can be removed from the graph. By reducing the number of cache lines to be analysed, and by removing vacuous blocks, we can perform more efficient (faster) analysis of a block.

Observation 2 : *For a cache line c_i, a reference block b_{ref} can experience a cache hit (or miss) only when the instruction executed by b_{ref} is the same (or different) as the instruction in the reaching cache state.* During the analysis of block b_{ref}, all instructions that can be executed on cache line c_i (for every block b in the CFG) can be abstracted as *same* (0) when $BI(b)[i] = BI(b_{ref})[i]$, or *different* (1) when $BI(b)[i] \neq BI(b_{ref})[i]$, or as *no instruction* ($\mp$) when there is no instruction on c_i in b ($BI(b)[i] = \mp$), or as *not of interest* ($\times$) when there is no instruction on c_i in b_{ref} ($BI(b_{ref})[i] = \mp$).

For example, given $BI(B8) = [\mp, m6, m3, m4]$ and $BI(B1) = [m1, m2, m3, \mp]$, during the analysis of block $B8$ (reference block = $B8$ =b_{ref}), the instructions in block $B1$ can be abstracted as the vector $[\times, 1, 0, \mp]$, and is computed as follows:

1. The first element of the vector is $\times$ because the instruction is *not of interest* as the reference block does not have any instruction on cache line c_0 ($BI(B8)[0] = \mp$).
2. The second element of the vector is 1. This captures that for cache lines c_1 the instruction in $B1$ is *different* to the instruction in $B8$ ($BI(B1)[1] = m2 \neq m6 = BI(B8)[1]$).
3. The third element of the vector is 0. This captures that for cache lines c_2 the instruction in $B1$ is *same* as the instructions in $B8$ ($BI(B8)[2] = m3 = BI(B1)[2]$).
4. Finally, the fourth element of the vector is $\mp$. This captures that for cache lines c_3 there is *no instruction* in $B1$ ($BI(B1)[3] = \mp$).

This abstraction reduces the number of instructions ($|I|$) in the CFG to just four values ($\times, \mp, 0, 1$), which reduces the memory foot print and could reduces the analysis time. We refer to these four abstract instructions as *relative* instructions and the instructions (I) as *concrete* instructions.

An overview of the proposed approach is presented using Algorithm 3. In our approach, as explained earlier, we analyse each block individually. This is described using the for-loop on lines 1 to 8. For each reference block b_{ref} in B, on line 2, we first reduce the CFG and compute relative instructions w.r.t the reference block (using Algorithm 44, explained later in Section 6.2). Due to the relative instructions, cache states also needs to be represented using relative instructions (explained later in Section 6.5). On line 3, using a fixed point computation, we compute all the possible reaching relative cache states of the reference block (using Algorithm 5, explained later in Section 6.5). Finally, on line 6 to 7, the number of cache misses in the worst/best case are computed (using Definitions 21 and 22, explained later in Section 6.5).

Algorithm 3 Overview of the proposed approach

Input: A cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$.
Output: Compute the worst/best miss count (wmc/bmc) for all basic blocks in B.

1: **for each** $b_{ref} \in B$ **do**
2: $(G^r, B^r) = Reduce(CM, b_{ref})$ {Reduced graph and relative instructions (Section 6.2)}
3: $RCS^r_{b_{ref}} = FP_{UoA}(CM, G^r, BI^r)$ {Compute reaching relative cache states (Section 6.5)}
4: $wmc = wmc_{UoA}(RCS^r_{b_{ref}})$ {Cache miss in the worst case (Section 6.6)}
5: $bmc = bmc_{UoA}(RCS^r_{b_{ref}})$ {Cache miss in the best case (Section 6.6)}
6: $P(b_{ref}) = (wmc, bmc)$ {Solution for the second sub problem (Definition 6)}
7: **end for**
8: **return** $P(b)$ (wmc, bmc) for all basic blocks b in B

We now present each of the algorithms and definition in detail.

6.2 Reducing the CFG and abstracting the instructions

We now formalise Observations 1 and 2 using Algorithm 4. Given a cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$ and the reference block $b_{ref} \in B$, the objective of the algorithm is: (1) to compute a new reduced graph $G^r = \langle B^r, b_{init}, E^r\rangle$ which contains only these blocks that are relevant for the analysis of block b_{ref} (described in Observation 1) and, (2) to compute the function BI^r which describes the relative instructions executed by the blocks in B^r w.r.t b_{ref}, referred as the *relative instruction mapping* function. The algorithm contains the following three steps.

Step 1: Initialise (lines 2 to 4). On line 2, we initialise G^r to have the same content as G, i.e., same set of blocks ($B^r = B$), initial block (b_{init}), edges ($E^r = E$). On lines 3 to 4, we initialise the function BI^r such that it does not contain any relative instructions for any block. For the example CFG (presented in Figure 2(a)), the graph G^r is presented in Figure 12(a).

Step 2: Relative instruction mapping (lines 7 to 22) Based on Observation 2, given a reference block $b_{ref} \in B$ and a cache line c_i, the instruction of any blocks $b_1 \in B^r$ can be expressed as *different* (1) when $BI(b_1)[i] \neq BI(b_{ref})[i]$ (checked on line 12), or *same* (0) when $BI(b_1)[i] = BI(b_{ref})[i]$ (checked on line 15), or *no instruction* ($\mp$) when there is no instruction in b_1 on cache line c_i, $BI(b_1)[i] = \mp$ (checked on line 18), or *not of*

interest ($\times$) when there is no instruction in b_{ref} on cache line c_i, $BI(b_{ref})[i] = \mp$ (checked on line 9). This relation is captured using the relative instruction mapping $BI^r(b_1)$.

For example, given $BI(b_{ref}) = [\mp, m6, m3, m4]$ and $BI(b1) = [m1, m2, m3, m4]$, we illustrate the *relative instruction mapping* function BI^r using Figure 13,

- For cache line c_0, $BI^r(b_1)[0] = \times$ because, the reference block b_{ref} does not have an instruction ($BI(b_{ref})[0] = \mp$).
- For cache line c_1, $BI^r(b_1)[1] = 1$ because, the reference block b_{ref} has an instruction $m6$ which is different to the instruction $m2$ of the basic block b_1 ($BI(b_{ref})[1] = m6 \neq m2 = BI(b_1)[1]$).
- For cache line c_2, $BI^r(b_1)[2] = 0$ because, the reference block b_{ref} has the instruction $m3$ which is the same as the instruction in the basic block b_1 ($BI(b_{ref})[2] = m3 = BI(b_1)[2]$).
- For cache line c_3, $BI^r(b_1)[3] = \mp$ because, the basic block b_1 does not have an instruction ($BI(b_1)[3] = \mp$).

For the graph G^r in Figure 12(a), the relative mapping (w.r.t $B8$) for every block is shown in Figure 12(b). Note that for all blocks b in B^r, if the reference block (b_{ref}) does not have an instruction on cache line c_i ($BI(b_{ref})[i] = \mp$), then for cache line c_i, the relative instruction for all blocks is $\times$ ($BI^r(b)[i] = \times$). This shows that during the analysis of b_{ref}, the cache line c_i is not considered. For example, in Figure 12(b), the reference block $B8$ does not have an instruction on cache line c_0 ($BI(B8)[0] = \mp$). Thus, for cache line c_0, the relative instruction for all blocks is $\times$ ($BI^r(b)[i] = \times$).

Step 3: Remove vacuous blocks and update edges (lines 24 to 44) Based on Observation 1, we can remove vacuous blocks which do not affect the precision of the reference block b_{ref}. We define a block $b_1 \in B^r$ to be vacuous, if $BI^r(b_1) \in (\{\times, \mp\})^N$. This check is done on line 25. It is possible for the initial block (b_{init}) to be vacuous. However, if the initial block has more than one successors, removing the initial block may result in multiple initial blocks. Thus, to simplify the analysis, we do not remove the initial block even when it is vacuous (line 25).

If a block b_1 is vacuous, we first compute the predecessors (line 27) and the successors (line 28) of b_1. Secondly, we remove the incoming edges to b_1 from each predecessor block (on lines 30 to 32) and, we remove the outgoing edges from b_1 to each successor block (on lines 33 to 35). Thirdly, we create a transition from each predecessor to each successor of b_1. This is achieved using the nested loop on lines 37 to 41. Finally, on line 42, the vacuous block b_1 is removed.

For the graph shown in Figure 12(b), blocks $B2$ and $B3$ do not contain any relative instructions ($BI^r(B2) = [\times, \mp, \mp, \mp]$ and $BI^r(B3) == [\times, \mp, \mp, \mp]$, thus, they are removed for the graph and the updated graph G^r is presented in Figure 12(c). This is the *reduced graph* G^r (w.r.t to $b_{ref} = B8$).

So far, we have presented how to reduce the graph size ($Reduce(CM, b_{ref}) = (G^r, BI^r)$), and compute the *relative instruction mapping* function BI^r. Due to the relative instructions, cache states also needs to be represented using relative instructions. The next step describes the behaviour of the cache.

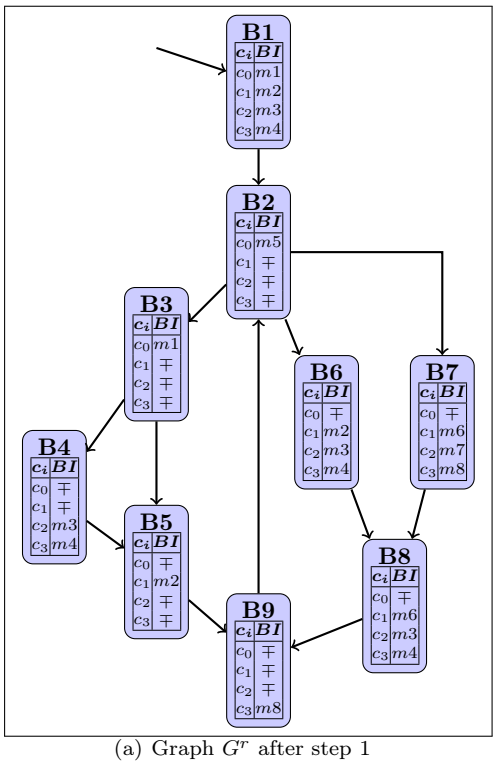
Algorithm 4 *Reduce*: Reduce the CFG and abstract instructions.

Input: Cache model $CM = \langle I,\ C,\ CI,\ G,\ BI\rangle$ and a reference block $b_{ref} \in B$.
Output: $G^r = \langle B^r, b_{init}, E^r\rangle$ and $BI^r : B^r \rightarrow (\{\times, \mp, 1, 0\})^N$

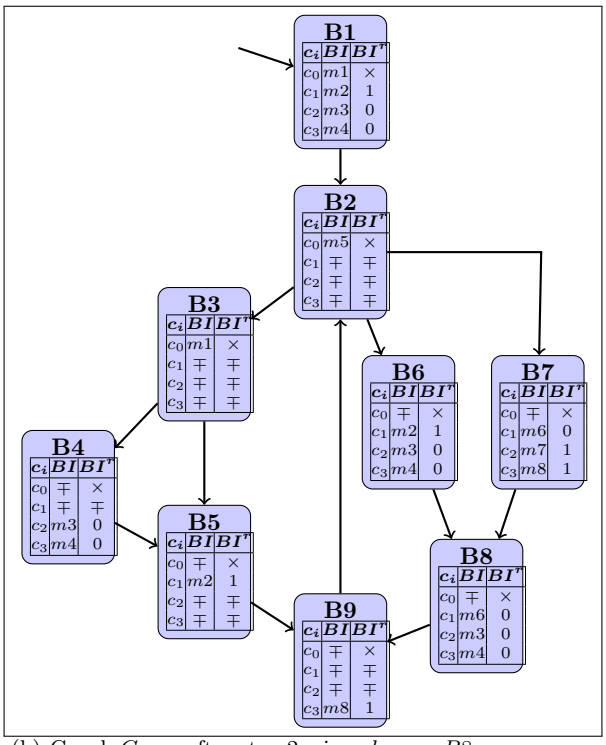
1: {Step 1: Initialise}
2: $B^r = B$, $E^r = E$, $G^r = \langle B^r, b_{init}, E^r\rangle$ {Copy all blocks and edges}
3: **for each** $b_1 \in B^r$ **do**
4: $BI^r(b_1) = \emptyset$ {Initialise the vector $BI^r(b_1)$}
5: **end for**

6: {Step 2: Relative instruction mapping}
7: **for each** $b_1 \in B^r$ **do**
8: **for each** $c_i \in C$ **do**
9: **if** $(BI(b_{ref})[i] = \mp)$**{**not of interest**}** **then**
10: $BI^r(b_1)[i] = \times$
11: **end if**
12: **if** $(BI(b_{ref})[i] \neq BI(b_1)[i])$ {different instruction} **then**
13: $BI^r(b_1)[i] = 1$
14: **end if**
15: **if** $(BI(b_{ref})[i] = BI(b_1)[i])$**{**same instruction**}** **then**
16: $BI^r(b_1)[i] = 0$
17: **end if**
18: **if** $(BI(b_1)[i] = \mp)$**{**no instruction**}** **then**
19: $BI^r(b_1)[i] = \mp$
20: **end if**
21: **end for**
22: **end for**

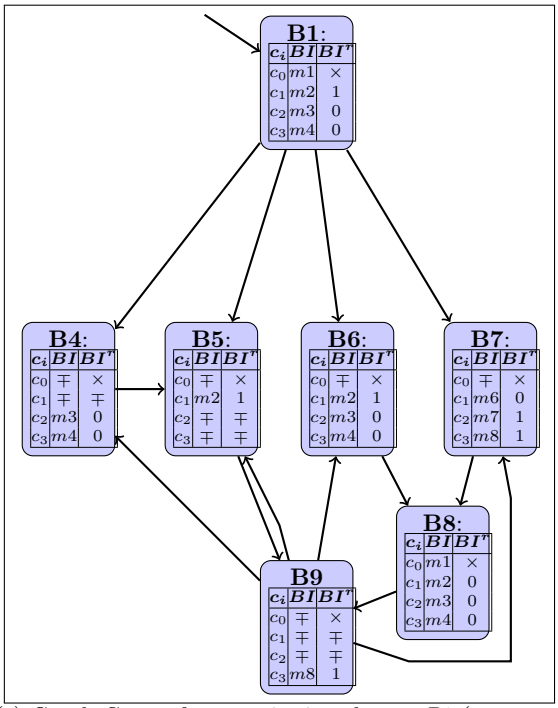
23: {Step 3: Remove vacuous blocks and update edges}
24: **for each** $b_1 \in B^r$ **do**
25: **if** $(BI(b_1) \in (\{\times, \mp\})^N) \wedge (b_1 \neq b_{init})$ {check for all vacuous blocks, excluding initial block} **then**
26: {Compute predecessors and successors of b_1}
27: $Preds = \{b_2 | b_2 \rightarrow b_1 \in E^r\}$
28: $Succ = \{b_2 | b_1 \rightarrow b_2 \in E^r\}$
29: {Remove incoming and outgoing edges of b_1}
30: **for each** $b_2 \in Preds$ **do**
31: $E^r = E^r \setminus \{b_2 \rightarrow b_1'\}$
32: **end for**
33: **for each** $b_2 \in Succ$ **do**
34: $E^r = E^r \setminus \{b_2 \rightarrow b_1'\}$
35: **end for**
36: {Add new edge from each predecessor to each successor }
37: **for each** $b_p' \in Preds$ **do**
38: **for each** $b_s' \in Succ$ **do**
39: $E^r = E^r \cup \{b_p' \rightarrow b_s'\}$
40: **end for**
41: **end for**
42: $B^r = B^r \setminus \{b_1\}$ {Remove block b_1}
43: **end if**
44: **end for**



(a) Graph G^r after step 1



(b) Graph $G_{b_{ref}}$ after step 2, given $b_{ref} = B8$



(c) Graph $G_{b_{ref}}$ after step 3, given $b_{ref} = B8$ (reduced graph).

Figure 12: Illustration of Algorithm 4 with $b_{ref} = B8$.

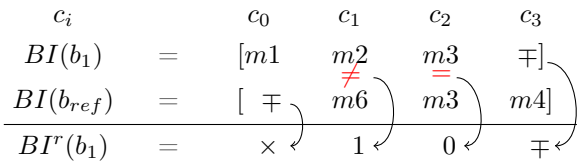


Figure 13: Illustration of the function $BI^r(b)$

6.3 Relative cache states

We describe the state of a cache using the notion of *relative cache states*. A relative cache state is described as a vector $[inst_0^r, inst_1^r, \ldots, inst_{N-1}^r]$, where each element $inst_i^r$ is described w.r.t the instruction $(BI(b_{ref})[i])$ in the block b_{ref}. For a cache line $c_i \in C$:

- $inst_i^r = 1$ represents that the instruction in the cache is *different* to the instruction executed in the reference block b_{ref}. E.g., $BI(b_{ref})[i] = m1$ and $inst_i^r = 1$ represents that the instruction on the cache is not $m1$.
- $inst_i^r = 0$ represents that the instruction in the cache is *same* as the instruction executed in the reference block b_{ref}. E.g., $BI(b_{ref})[i] = m1$ and $inst_i^r = 0$ represents that the instruction on the cache is $m1$.
- $inst_i^r = \top$ represents that the cache does not contain any instruction i.e., it is *empty*.
- $inst_i^r = \bot$ represents that the cache has an *unknown* instruction.
- $inst_i^r = \times$ represents that the instruction on this cache line is *not of interest* during the analysis of b_{ref}.

Definition 16 (Relative cache state)**.** *Given a reference block b_{ref}, a relative cache state $cs^r \in (\{1, 0, \top, \bot, \times\})^N$, is a vector $[inst_0^r, inst_1^r, \ldots, inst_{N-1}^r]$, where each element $inst_i^r \in \{1, 0, \top, \bot, \times\}$. Also, the set of all possible relative cache states (w.r.t b_{ref}) is denoted as CS^r.*

Before we illustrate relative cache states, we introduce two key terms essential to cache analysis. For any basic block b, the *reaching relative cache states* represent the set of relative cache states prior to the execution of a basic block and the *relative leaving cache states* represent the set of cache states after the execution of the basic block. We denote the reaching relative cache states of a basic block b as RCS_b^r and the relative leaving cache states of a basic block b as LCS_b^r.

Illustration

A fragment of the reduced graph (Figure 12(c)) is presented in Figure 14. Using this graph we illustrate relative reaching and leaving cache states of a block. Given the reference block $b_{ref} = B8$ with $BI(b_{ref}) = [\mp, m6, m3, m4]$ and the relative instruction mapping of each block (described by the function BI^r), we illustrate the relative cache state as we start executing the initial block $B1$. Initially, the cache is empty and the reference block does not have an instruction on cache line c_0 $(BI^r(b_{ref})[0] = \mp)$. Thus the instruction on cache line c_0 is *not of interest* $(\times)$ resulting in the initial cache state of $[\times, \top, \top, \top]$, denoted as $cs_\top^r$. Note this initial cache state is unlike the NUS approach, where the initial state is represented using the vector $[\top, \top, \top, \top]$. In our approach, given the reference block b_{ref} (in this case $b_{ref} = B8$), we are only interested in the cache line c_1, c_2, c_3. Thus, we ignore the cache state w.r.t cache line c_0, as it does not

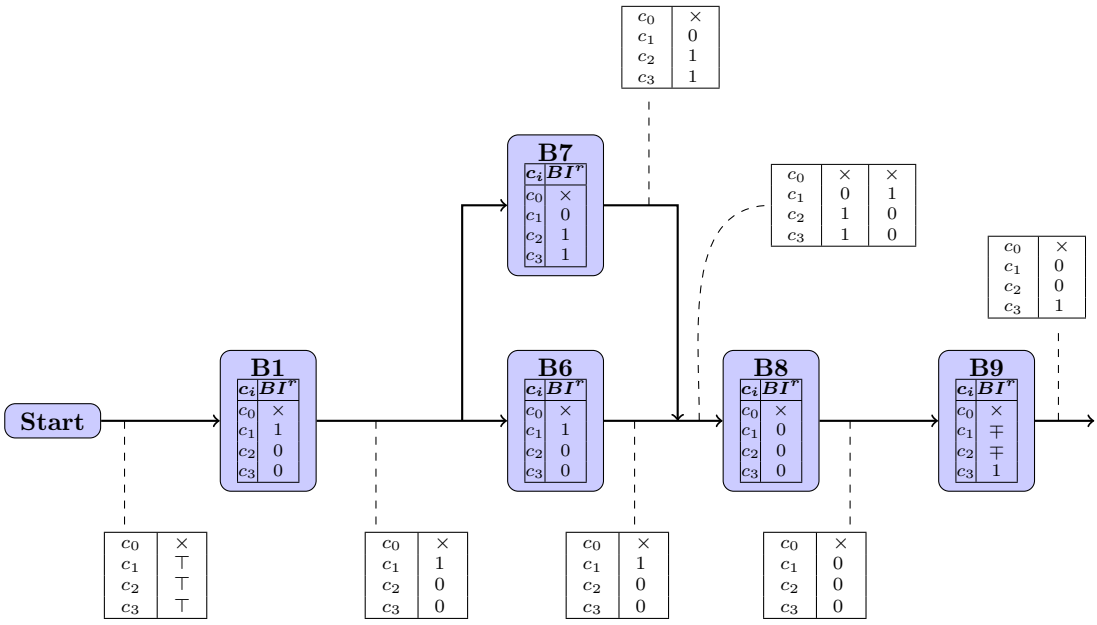


Figure 14: Illustration of the relative cache states.

affect the precision of the reference block. Also, by ignoring c_0, allows for a memory efficient implementation. In this case, the relative cache state can be represented using a vector with only three elements, instead of four elements, saving 25% of memory.

After execution of block $B1$, where the relative instructions are described by $BI^r(B1) = [\times, 1, 0, 0]$, cache lines c_1, c_2, c_3 will contain relative instructions 1, 0, 0. Hence after $B1$ executes, the relative cache state $cs^r = [\times, 1, 0, 0]$. Here, $cs^r[0] = \times$ represents the instruction on cache line c_0 is *not of interest*. $cs^r[1] = 1$ represents that the instruction in the cache is not $m6$, because $BI(b_{ref})[1] = m6$. $cs^r[2] = 0$ represents that the instruction in the cache is $m3$, because $BI(b_{ref})[2] = m3$. Similarly, $cs^r[3] = 0$ represents the instruction in the cache is $m4$ ($BI(b_{ref})[3] = m4$).

The relative cache state $cs^r_\top$ is the state of the cache prior to the execution of the basic block $B1$. Thus, $RCS^r_{B1} = \{[\times, \top, \top, \top]\}$ is the set of reaching relative cache states of block $B1$. Similarly, $LCS^r_{B1} = \{[\times, 1, 0, 0]\}$ is the set of relative leaving cache states of block $B1$.

Now, the control reaches a branch due to which, it is possible to execute either block $B6$ or block $B7$. In this case, $RCS^r_{B6} = LCS^r_{B1} = RCS^r_{B7}$. After executing blocks $B6$ (or $B7$), the state of the cache is $[\times, 1, 0, 0]$ (or $[\times, 0, 1, 1]$).

Block $B8$ has two incoming edges: from blocks $B6$ and $B7$. To compute RCS^r_{B8}, we need to *join* LCS^r_{B6} and LCS^r_{B7}. In this case, the join function is a union over the set of relative cache states. Thus, $RCS^r_{B8} = LCS^r_{B6} \cup LCS^r_{B7}$, resulting in $RCS^r_{B8} = \{[\times, 0, 1, 1], [\times, 1, 0, 0]\}$.

The UoA join function is different to a simple union operator, due to its handling of unknown instructions ($\perp$). During the computation of the reaching relative cache states of a block b, we may need to *join* two leaving relative cache states that may contain unknown ($\perp$) instructions for some cache lines. E.g., $LCS^r_1 = \{cs^r_1\}$ where $cs^r_1 = [1, \perp, \perp, \perp]$ and $LCS^r_2 = \{cs^r_2\}$ where $cs^r_2 = [1, \perp, 0, \perp]$. In this case, given two relative cache states cs^r_1 and cs^r_2, the cache state cs^r_1 is *subsumed* by the cache state cs^r_2, if for all cache lines ($c_i \in C$), when the relative instruction in cs^r_2 is same as the instruction in cs^r_1 ($cs^r_2[i] = cs^r_1[i]$) or, the relative instruction in cs^r_1 is unknown

($cs_1^r[i] = \perp$). In this case, we can remove cs_1^r, as it does not contain any extra information compared to cs_2^r.

For example, let $RCS_3^r = LCS_1^r \cup LCS_2^r$ which contains two relative cache states $cs_1^r = [1, \perp, \perp, \perp]$ and $cs_2^r = [1, \perp, 0, \perp]$. Since cs_1^r is subsumed by cs_2^r, we can reduce the reaching relative cache states of b to $RCS_b = \{[1, \perp, 0, \perp]\}$. This idea of subsumed relative cache states is further explained during the fixed point computation.

Intuitively, when joining two sets of reaching relative cache states, the join function retains only those relative cache states that can not be subsumed. The join function for our approach is defined in Definition 17.

Definition 17 (UoA join function)**.** *The UoA join function* $J_{UoA} : 2^{CS^r} \times 2^{CS^r} \rightarrow 2^{CS^r}$ *where for any two sets of relative cache states* $CS_1^r \in 2^{CS^r}$ *and* $CS_2^r \in 2^{CS^r}$*,*

$$J_{UoA}(CS_1^r, CS_2^r) = CS_1^r \cup CS_2^r \setminus \{cs_1^r \in (CS_1^r \cup CS_2^r) | \exists cs_2^r \in (CS_1^r \cup CS_2^r) \wedge S_{UoA}(cs_1^r, cs_2^r)\}$$

where, $S_{UoA} : CS^r \times CS^r \rightarrow \{true, false\}$ *where, for any* $cs_1^r, cs_2^r \in CS^r$*,*

$$S_{UoA}(cs_1^r, cs_2^r) = \begin{cases} true & : if \ \ \forall i \in [0, N-1], cs_2^r[i] = cs_1^r[i] \ or \ cs_1^r = \perp \\ false & : otherwise \end{cases}$$

$\boldsymbol{cs_1^r}$	$\boldsymbol{cs_2^r}$	$\boldsymbol{S_{UoA}(cs_1^r, cs_2^r)}$
$[\times, 1, \perp, \perp]$	$[\times, 1, \perp, 0]$	$true$
$[\times, 1, \perp, 0]$	$[\times, 1, \perp, \perp]$	$false$

Table 6: Illustrate the subsumed function S_{UoA}

$\boldsymbol{CS_1^r}$	$\boldsymbol{CS_2^r}$	$\boldsymbol{CS_1^r \cup CS_2^r}$	$\boldsymbol{J_{UoA}(CS_1^r, CS_2^r)}$
$\{[\times, 1, \perp, \perp],$ $[\times, 1, 1, 1]\}$	$\{[\times, 1, \perp, 0],$ $[\times, 1, 1, 1]\}$	$\{[\times, 1, \perp, \perp],$ $[\times, 1, \perp, 0],$ $[\times, 1, 1, 1]\}$	$\{[\times, 1, \perp, 0],$ $[\times, 1, 1, 1]\}$

Table 7: Illustrate the join function J_{UoA}

Using Table 6, we first illustrate the UoA subsumed function (S_{UoA}). In the first row, given two relative cache states $cs_1^r = [\times, 1, \perp, \perp]$ and $cs_2^r = [\times, 1, \perp, 0]$, $S_{UoA}(cs_1^r, cs_2^r) = true$. This shows the case when cs_1^r is subsumed by cs_2^r. In the second row, given two cache states $cs_1^r = [\times, 1, \perp, 0]$ and $cs_2^r = [\times, 1, \perp, \perp]$, $S_{UoA}(cs_1^r, cs_2^r) = false$. This shows the case when cs_1^r is not subsumed by cs_2^r.

Using Table 7, we illustrate the UoA join function. Given two sets of relative cache states $CS_1^r = \{[\times, 1, \perp, \perp], [\times, 1, 1, 1]\}$ and $CS_2^r = \{[\times, 1, \perp, 0], [\times, 1, 1, 1]\}$, $CS_1^r \cup CS_2^r = \{[\times, 1, \perp, \perp], [\times, 1, \perp, 0], [\times, 1, 1, 1]\}$. As illustrated in first row of Table 6, relative cache state $[\times, 1, \perp, \perp]$ is subsumed by $[\times, 1, \perp, 0]$. Thus, $J_{UoA}(CS_1^r, CS_2^r) = \{[\times, 1, \perp, 0], [\times, 1, 1, 1]\}$.

6.4 UoA transfer function

The UoA transfer function describes how a reaching relative cache state $cs_1^r = [inst_0^1, inst_1^1, \ldots, inst_{N-1}^1]$ of a basic block is transformed into a relative leaving cache state $cs_2^r = [inst_0^2, inst_1^2, \ldots,$

$inst^2_{N-1}]$, after executing the relative instructions in the basic block b with $BI^r(b) = [inst^b_0, inst^b_1, \ldots, inst^b_{N-1}]$. The transformation can be described using the following four cases:

(1) If a block b does not have an instruction mapped to a cache line c_i ($inst^b_i = \mp$), or (2) a cache line c_i is not of interest ($inst^b_i = \times$) then after execution of the block, the contents of the cache corresponding to that cache line remains unchanged ($inst^2_0 = inst^1_i$). (3) If a block b has the same ($inst^b_i = 0$), or (4) different ($inst^b_i = 1$) relative instruction, then after execution of the block, the content of the relative leaving cache is changed to the relative instruction executed by the block ($cs^r_2[i] = inst^b_i$]).

For example, given a reaching relative cache state $cs^r_1 = [\times, 1, 1, 1]$ of block b, with $BI^r(b) = [\times, \mp, 0, 0]$, the relative leaving cache state $cs^r_2 = [\times, 1, 0, 0]$. Here, for cache line c_0 the instruction in block b is not of interest ($BI^r(b)[0] = \times$). Thus, the state of the cache is unchanged ($cs^r_2[0] = \times = cs^r_1[0]$). For cache line c_1 the block b does not have a relative instruction ($BI^r(b)[0] = \mp$). Hence, the state of the cache is also unchanged ($cs^r_2[1] = 0 = cs^r_1[1]$). In contrast, for cache lines c_2 and c_3, block b has relative instructions ($BI^r(b)[2] = 0$ and $BI^r(b)[3] = 0$). Thus, the state of the cache is updated ($cs^r_2[2] = 0 = BI^r(b)[2]$ and $cs^r_2[3] = 0 = BI^r(b)[3]$).

We now define the transfer function using Definition 18.

Definition 18 (UoA transfer function)**.** *The UoA transfer function $T_{UoA} : B \times CS^r \to CS^r$ where, for a given basic block $b \in B$ and a relative cache state $cs^r_1 \in CS^r$,*

$$T_{UoA}(b, cs^r_1) = cs^r_2$$

where for all cache lines $c_i \in C$ where $i \in [0, N-1]$,

$$cs^r_2[i] = \begin{cases} BI^r(b)[i] & : \; if \; (BI^r(b)[i] = 0) \vee (BI^r(b)[i] = 1) \\ cs^r_1[i] & : \; otherwise \end{cases}$$

Illustration

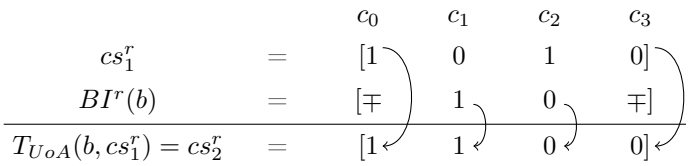


Figure 15: Illustration of the UoA transfer function

Figure 15 illustrates the operation of the transfer function T_{UoA}. Given a block b, with $BI^r(b) = [\mp, 1, 0, \mp]$, and a reaching relative cache state $cs^r_1 = [1, 0, 1, 0]$, the relative leaving cache state $cs^r_2 = T_{UoA}(b, cs^r_1)$ is computed as follows:

For cache lines c_0 and c_3, $cs^r_2[0] = cs^r_1[0]$ and $cs^r_2[3] = cs^r_1[3]$ because the block b has no instructions ($BI^r(b)[0] = \mp$ and $BI^r(b)[3] = \mp$) mapped to the cache lines. Thus, the content of the relative cache state is unchanged.

For the cache lines c_1 and c_2, the relative instructions are 1 ($BI^r(b)[1]$) and 0 ($BI^r(b)[2]$) respectively. Thus, the relative leaving cache state contains 1 on cache line c_1 ($cs^r_2[1] = BI^r(b)[i]$) and 0 on cache line c_2 ($cs^r_2[2] = BI^r(b)[2]$). Hence, given a block b, with $BI^r(b) = [\mp, 1, 0, \mp]$ and a reaching relative cache state $cs^r_1 = [1, 0, 1, 0]$, $T_{UoA}(b, cs^r_1) = cs^r_2 = [1, 1, 0, 0]$.

Given a reaching relative cache state of a basic block, the transfer function (Definition 18) computes its leaving relative cache state. During cache analysis, a block may have a set of

reaching relative cache states. E.g., in Figure 14, the block $B8$ has two reaching cache states, $RCS^r_{B8} = \{[\times, 0, 1, 1], [\times, 1, 0, 0]\}$. Thus, we extend the transfer function to compute a set of relative leaving cache states for a given set of reaching relative cache states for any block.

Definition 19 (Extended UoA transfer function)**.** *We define the transfer function* $T_{UoA} : B \times 2^{CS^r} \rightarrow 2^{CS^r}$ *where, for a given basic block* $b \in B$ *and a set of cache states* $CS^r_1 \subseteq 2^{CS^r}$*,*

$$T_{UoA}(b, CS^r_1) = CS^r_2$$

where for all cache states $cs^r_i \in CS^r_1$ *is,*

$$CS^r_2 = \bigcup_{i=1}^{|CS^r_1|} \{T_{UoA}(b, cs^r_i)\}$$

In Figure 14, block $B8$ has two reaching cache states ($RCS^r_{B8} = \{cs^r_1, cs^r_2\}$) where $cs^r_1 = [\times, 0, 1, 1]$ and $cs^r_2 = [\times, 1, 0, 0]$. In this case, $T_{UoA}(B8, \{cs^r_1, cs^r_2\}) = \{T_{UoA}(B8, cs^r_1)\} \cup \{T_{UoA}(B8, cs^r_2)\}$. In this example, $T_{UoA}(B8, cs^r_1) = [\times, 0, 0, 1]$ and also $T_{UoA}(B8, cs^r_2) = [\times, 0, 0, 1]$. Thus, $T_{UoA}(B8, \{cs_1, cs_2\}) = \{[\times, 0, 0, 1]\} \cup \{[\times, 0, 0, 1]\} = \{[\times, 0, 0, 1]\}$.

6.5 Computing all possible reaching relative cache states of the reference block

The first step for cache analysis involves the computation of all possible reaching relative cache states of b_{ref} ($RCS^r_{b_{ref}}$), using the fixed point computation algorithm presented in Algorithm 5.

As illustrated in Figure 14, the initial state of the cache is empty($cs^r_\top$), and is based on the instructions of the reference block. Similarly, like the NUS and the Absint approaches, we must also introduce the unknown cache state ($cs^r_\bot$), which is explained later during this algorithm. In general, the empty/unknown cache state is different for each $b_{ref} \in B$. Thus, for a given reference block b_{ref}, we first compute the empty cache state $cs^r_\top$, and unknown cache state $cs^r_\bot$ on lines 2 to 8.

For each cache line c_i, if the reference block b_{ref} does not have an instruction (in this case, $BI^r(b_{ref}) = \times$), then the cache state on c_i is *not of interest* during the analysis of the reference block b_{ref}. Thus, the relative instruction on cache line c_i of $cs^r_\top$ and $cs^r_\bot$ is set to $\times$ (line 4). Otherwise ($BI^r(b_{ref}) \neq \times$), on line 6, for cache line c_i, the empty cache state is set to be $\top$ ($cs^r_\top[i] = \top$), and the unknown cache state is set to be $\bot$ ($cs^r_\bot[i] = \bot$).

Using relative cache states $cs^r_\top$ and $cs^r_\bot$, we initialise the reaching relative cache states for all blocks (lines 11 to 17). Since we assume that initially the state of the cache is empty, on line 13 for the initial block b_{init} we set its reaching as $RCS^{r^1}_{b_{init}} = \{cs^r_\top\}$. Here, the notation $RCS^{r^i}_b$ represents the reaching relative cache states of block b in iteration i. E.g., $RCS^{r^1}_{b_{init}}$ represents the reaching relative cache states of block b_{init} for iteration 1. For rest of the blocks, the initial state of the cache is unknown. Thus, on line 15, we set their reaching cache states as $\{cs^r_\bot\}$. After initialisation, we compute the relative leaving cache states of each block, on lines 19 to 22. We apply the transfer function T_{UoA} to every block and its corresponding reaching relative cache states.

The iteration index (i) is incremented (line 23) to signal the start of the next iteration. Next, on lines 25 to 34, the reaching relative cache states of each block are computed. For the initial block b_{init}, we know that the reaching relative cache state is always empty. Thus, on line 27, we

Algorithm 5 FP_{UoA}: Fixed point computation for UoA approach

Input: A cache model $CM = \langle I,\ C,\ CI,\ G,\ BI \rangle$, reduced graph $G^r = \langle B^r, b_{init}, E^r \rangle$ and $BI^r : B^r \rightarrow (\{\times, \mp, 1, 0\})^N$.

Output: Reaching relative cache states of block b_{ref} ($RCS^r_{b_{ref}}$).

1: {Initialise $cs^r_\top$ and $cs^r_\bot$}
2: **for each** $c_i \in C$ **do**
3: **if** $BI^r(b_{ref})[i] = \times$ **then**
4: $cs^r_\top[i] = \times, \quad cs^r_\bot[i] = \times$ {When b_{ref} has no instruction (in this case, $BI^r(b_{ref}) = \times$) on cache line c_i, initialise empty/unknown cache states as *not of interest.*}
5: **else**
6: $cs^r_\top[i] = \top, \quad cs^r_\bot[i] = \bot$ {When b_{ref} has an instruction on cache line c_i, initialise empty/unknown cache states as $\top/\bot$.}
7: **end if**
8: **end for**
9: $i = 1$ {iteration counter}
10: {Initialise RCS^r for all blocks }
11: **for each** $b \in\ B$ **do**
12: **if** $b = b_{init}$ **then**
13: $RCS^{r^1}_{b_{init}} = \{cs^r_\top\}$ {for the initial block, the initial state of the cache is always empty}
14: **else**
15: $RCS^{r^1}_b = \{cs^r_\bot\}$ {for rest of the blocks, the initial state of the cache is unknown}
16: **end if**
17: **end for**

18: **repeat**
19: {Compute LCS^r for all blocks}
20: **for each** $b \in\ B^r$ **do**
21: $LCS^{r^i}_b = T_{UoA}(b, RCS^{r^i}_b)$
22: **end for**

23: $i = i + 1$; {Next iteration}
24: {Compute RCS^r for the next iteration $i + 1$}
25: **for each** $b \in\ B^r$ **do**
26: **if** $b = b_{init}$ **then**
27: $RCS^{r^i}_{b_{init}} = \{cs^r_\top\}$
28: **else**
29: $RCS^{r^i}_b = \emptyset$
30: **for each** $LCS^{r^i}_{b'}$, where $(b', b) \in E^r$ **do**
31: $RCS^{r^i}_b = J_{UoA}(RCS^{r^i}_b, LCS^{r^i}_{b'})$
32: **end for**
33: **end if**
34: **end for**

35: **until** $\forall b \in\ B^r$, $RCS^{r^i}_b = RCS^{r^{i-1}}_b$ {Termination condition}
36: **return** $RCS^r_{b_{ref}}$

always set its reaching as $RCS^{r^i}_{b_{init}} = \{cs_\top\}$. For rest of the blocks, we first initialise the reaching relative cache state as empty set (line 29), and on lines 30 to 32, the reaching cache states are computed by looking at the relative leaving cache states of the predecessors (b') of the block b and using the UoA join function.

The iterative process, repeat-until loop on lines 18 to 35, is repeated until a fixed point is reached, i.e., if two consecutive iterations have the same sets of reaching relative cache states for all blocks (line 35).

Illustration of the fixed point algorithm

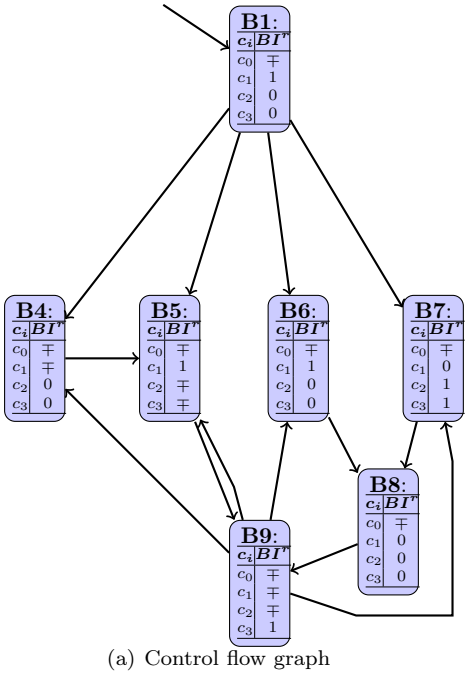
We describe the fixed point algorithm using the Figure 16. The reduced graph G^r w.r.t the reference block, ($b_{ref} = B8$) is presented in Figure 16(a). Figure 16(b) presents a table showing the reaching and leaving relative cache states for all blocks, during each iteration of the algorithm.

We first initialise the empty cache state $cs^r_\top$ and the unknown cache state $cs^r_\bot$ on lines 2 to 8. The reference block $B8$ does not have an instruction on cache line c_0 ($BI^r(b_{ref})(0) = \mp$), and has an instruction for rest of the cache lines (c_1, c_2, c_3). Thus, we get $cs^r_\top = [\times, \top, \top, \top]$ and $cs^r_=[\times, \bot, \bot, \bot]$.

Using relative cache states $cs^r_\top$ and $cs^r_\bot$, we initialise the reaching relative cache states for all blocks (lines 11 to 17). For the CFG in Figure 16(a), the initial block (b_{init}) is $B1$. Since we assume that initially the state of the cache is empty, on line 13, we set $RCS^{r^i}_{b_{init}}$ as $\{cs^r_\top\} = \{[\times, \top, \top, \top]\}$. For rest of the blocks, on line 7, the state of the cache is unknown. Thus, we set their reaching cache states as $\{cs^r_\bot\} = \{[\times, \bot, \bot, \bot]\}$, as presented during the first iteration in column 3 of Figure 16(b).

After initialisation, to compute the relative leaving cache states of each block, we apply the transfer function T_{UoA} (lines 19 to 22). For example, in iteration 1, $LCS^{r^1}_{B1} = T_{UoA}(B1, RCS^{r^1}_{B1}) = T_{UoA}(B1, \{[\times, \top, \top, \top]\}) = \{[\times, 1, 0, 0]\}$. Similarly, the relative leaving cache states for rest of the blocks are computed.

The iteration index (i) is incremented (line 23) to signal the start of the next iteration ($i = 2$). Next, the reaching relative cache states of each block, for iteration $i = 2$, are computed (lines 25 to 34). For the initial block $B1$, we know that the reaching relative cache state is always $cs^r_\top$. Thus, on line 27, we set its reaching relative cache state as $RCS^{r^2}_{B1} = \{cs_\top\}$. For the remaining blocks, on lines 29 to 32, the reaching relative cache states are computed by joining the leaving relative cache states of its predecessor blocks (b') using the UoA join function. E.g., the predecessors of $B4$ are $B1$ and $B9$, such that $b' \in \{B1, B9\}$. From the previous iteration ($i = 1$), the $LCS^{r^1}_{B1} = \{[\times, 1, 0, 0]\}$ and $LCS^{r^1}_{B9} = \{[\times, \bot, \bot, 1]\}$. The process for computing the reaching relative cache states $RCS^{r^2}_{B4}$, in the next iteration $i = 2$, is as follows.



(a) Control flow graph

Iteration (i)	Block (b)	Reaching Cache States (RCS_b^i)	Leaving Cache States (LCS_b^i)
1	B1	$\{[\times, \top, \top, \top]\}$	$\{[\times, 1, 0, 0]\}$
	B4	$\{[\times, \bot, \bot, \bot]\}$	$\{[\times, \bot, 0, 0]\}$
	B5	$\{[\times, \bot, \bot, \bot]\}$	$\{[\times, 1, \bot, \bot]\}$
	B6	$\{[\times, \bot, \bot, \bot]\}$	$\{[\times, 1, 0, 0]\}$
	B7	$\{[\times, \bot, \bot, \bot]\}$	$\{[\times, 0, 1, 1]\}$
	B8	$\{[\times, \bot, \bot, \bot]\}$	$\{[\times, 0, 0, 0]\}$
	B9	$\{[\times, \bot, \bot, \bot]\}$	$\{[\times, \bot, \bot, 1]\}$
2	B1	$\{[\times, \top, \top, \top]\}$	$\{[\times, 1, 0, 0]\}$
	B4	$\{[\times, 1, 0, 0], [\times, \bot, \bot, 1]\}$	$\{[\times, 1, 0, 0], [\times, \bot, 0, 0]\}$
	B5	$\{[\times, 1, 0, 0], [\times, \bot, \bot, 1]\}$	$\{[\times, 1, 0, 0], [\times, 1, \bot, 1]\}$
	B6	$\{[\times, 1, 0, 0], [\times, \bot, \bot, 1]\}$	$\{[\times, 1, 0, 0]\}$
	B7	$\{[\times, 1, 0, 0], [\times, \bot, \bot, 0]\}$	$\{[\times, 0, 1, 1]\}$
	B8	$\{[\times, 1, 0, 0], [\times, 0, 1, 1]\}$	$\{[\times, 0, 0, 0]\}$
	B9	$\{[\times, 1, \bot, \bot], [\times, 0, 0, 0]\}$	$\{[\times, 1, \bot, 1], [\times, 0, 0, 1]\}$
3	B1	$\{[\times, \top, \top, \top]\}$	$\{[\times, 1, 0, 0]\}$
	B4	$\{[\times, 1, 0, 0], [\times, 1, \bot, 1], [\times, 0, 0, 1]\}$	$\{[\times, 1, 0, 0], [\times, 0, 0, 0]\}$
	B5	$\{[\times, 1, 0, 0], [\times, 1, \bot, 1], [\times, 0, 0, 1]\}$	$\{[\times, 1, 0, 0], [\times, 1, \bot, 1], [\times, 1, 0, 1]\}$
	B6	$\{[\times, 1, 0, 0], [\times, 1, \bot, 1], [\times, 0, 0, 1]\}$	$\{[\times, 1, 0, 0]\}$
	B7	$\{[\times, 1, 0, 0], [\times, 1, \bot, 1], [\times, 0, 0, 1]\}$	$\{[\times, 0, 1, 1]\}$
	B8	$\{[\times, 1, 0, 0], [\times, 0, 1, 1]\}$	$\{[\times, 0, 0, 0]\}$
	B9	$\{[\times, 1, 0, 0], [\times, 0, 0, 0], [\times, 1, \bot, 1]\}$	$\{[\times, 1, 0, 1], [\times, 0, 0, 1], [\times, 1, \bot, 1]\}$
4	B1	$\{[\times, \top, \top, \top]\}$	$\{[\times, 1, 0, 0]\}$
	B4	$\{[\times, 1, 0, 0], [\times, 1, 0, 1], [\times, 0, 0, 1]\}$	$\{[\times, 1, 0, 0], [\times, 0, 0, 0]\}$
	B5	$\{[\times, 1, 0, 0], [\times, 1, 0, 1], [\times, 0, 0, 1], [\times, 0, 0, 0]\}$	$\{[\times, 1, 0, 0], [\times, 1, \bot, 1], [\times, 1, 0, 1]\}$
	B6	$\{[\times, 1, 0, 0], [\times, 1, 0, 1], [\times, 0, 0, 1]\}$	$\{[\times, 1, 0, 0]\}$
	B7	$\{[\times, 1, 0, 0], [\times, 1, 0, 1], [\times, 0, 0, 1]\}$	$\{[\times, 0, 1, 1]\}$
	B8	$\{[\times, 1, 0, 0], [\times, 0, 1, 1]\}$	$\{[\times, 0, 0, 0]\}$
	B9	$\{[\times, 1, 0, 0], [\times, 0, 0, 0], [\times, 1, 0, 1]\}$	$\{[\times, 1, 0, 1], [\times, 0, 0, 1], [\times, 1, \bot, 1]\}$
5	B1	$\{[\times, \top, \top, \top]\}$	
	B4	$\{[\times, 1, 0, 0], [\times, 1, 0, 1], [\times, 0, 0, 1]\}$	
	B5	$\{[\times, 1, 0, 0], [\times, 1, 0, 1], [\times, 0, 0, 1], [\times, 0, 0, 0]\}$	
	B6	$\{[\times, 1, 0, 0], [\times, 1, 0, 1], [\times, 0, 0, 1]\}$	
	B7	$\{[\times, 1, 0, 0], [\times, 1, 0, 1], [\times, 0, 0, 1]\}$	
	B8	$\{\mathbf{[\times, 1, 0, 0]}, \mathbf{[\times, 0, 1, 1]}\}$	
	B9	$\{[\times, 1, 0, 0], [\times, 0, 0, 0], [\times, 1, 0, 1]\}$	

(b) Reaching and leaving relative cache states

Figure 16: Computing all possible reaching relative cache states of the reference block b_{ref} (B8) using the UoA approach.

1. Initialise $RCS^{r^2}_{B4} = \emptyset$ (line 29).
2. Using the UoA join function we process the LCS^r of each predecessor block ($B1, B9$), on lines 30 to 32, one at a time (in no particular order). For illustration we will first analyse $LCS^{r^1}_{B1}$ followed by $LCS^{r^1}_{B9}$ as follows.

$$\begin{aligned} RCS^{r^2}_{B4} &= J_{UoA}(RCS^{r^2}_{B4}, LCS^{r^1}_{B1}) \\ &= J_{UoA}(\emptyset, \{[\times, 1, 0, 0]\}) \\ &= \{[\times, 1, 0, 0]\} \\ RCS^{r^2}_{B4} &= J_{UoA}(RCS^{r^2}_{B4}, LCS^{r^1}_{B9}) \\ &= J_{UoA}(\{[\times, 1, 0, 0]\}, \{[\times, \perp, \perp, 1]\}) \\ &= \{[\times, 1, 0, 0], [\times, \perp, \perp, 1]\} \end{aligned}$$

The iterative process, repeat-until loop on lines 18 to 35, is repeated until a fixed point is reached, i.e., if two consecutive iterations have a same sets of reaching relative cache states for all blocks (line 35). For our example, it happens at the 5^{th} iteration, where the same set of reaching relative cache states are computed for all blocks. As a result, the reaching relative cache states of the 5^{th} iteration represents all the possible reaching relative cache states of the program.

Note that during the first iteration, we assumed the reaching relative cache states for blocks $B4$ to $B9$ as unknown ($\{[\times, \perp, \perp, \perp]\}$), but when we reach the fixed point, all the unknown cache states are resolved. Thus, we have

In summary, we have presented how to compute all possible reaching relative cache states of the reference block $RCS^r_{b_{ref}}$. This corresponds to the function call on line 3 of Algorithm 6 (reproduced from Algorithm 3, which shows the overall cache analysis approach). Given the set of reaching relative cache states of the reference block ($RCS^r_{b_{ref}}$), the next step (lines 4 to 5, Algorithm 6) in our approach is to compute the number of cache misses for the worst case and the best case.

Algorithm 6 Overview of the proposed approach (reproduced from Algorithm 3)

Input: A cache model $CM = \langle I,\ C,\ CI,\ G,\ BI \rangle$.
Output: Compute the worst/best miss count (wmc/bmc) for all basic blocks in B.

```
1: for each b_ref ∈ B do
2:    (G^r, B^r) = Reduce(CM, b_ref) {Reduced graph and relative instructions (Section 6.2)}
3:    RCS^r_{b_ref} = FP_UoA(CM, G^r, BI^r) {Compute reaching relative cache states (Section 6.5)}
4:    wmc = wmc(b_ref, RCS_{b_ref}) {Cache miss in the worst case (Section 6.6)}
5:    bmc = bmc(b_ref, RCS_{b_ref}) {Cache miss in the best case (Section 6.6)}
6:    P(b_ref) = (wmc, bmc) {Solution for the second sub problem (Definition 6)}
7: end for
8: return P(b) (wmc, bmc) for all basic blocks b in B
```

6.6 Calculating cache misses for the reference block

The second sub problem of cache analysis (Definition 6) involves the computation of the number of cache misses for the worst case and the best case. The number of cache misses could be computed by directly analysing the relative cache state $cs^r = [inst^r_1, inst^r_2, \ldots, inst^r_{N-1}]$. For a cache line $c_i \in C$, $inst_i = 1$ represents a cache miss. Thus, to compute the number of cache misses for a given cs^r we count the number of instances when $inst_i = 1$. E.g., given $cs^r = [\times, 0, 1, 1]$, we have two cache misses ($inst^r_2 = 1$ and $inst^r_3 = 1$). We define this computation in the following definition.

Definition 20 (UoA miss count)**.** *The UoA miss count function, $mc_{UoA} : CS^r \rightarrow \mathbb{N}^0$, where for any relative cache state $cs^r \in CS^r$ and a cache line $c_i \in C$ with $i \in [0, N-1]$ is defined as follows:*

$$mc_{UoA}(cs^r) = \sum_{i=0}^{N-1} miss_i$$

where,

$$miss_i = \begin{cases} 1 & : if \quad cs^r[i] = 1 \\ 0 & : otherwise \end{cases}$$

For example, given $cs^r = [\times, 0, 1, 1]$ then, $miss_0 = 0$, $miss_1 = 0$, $miss_2 = 1$, $miss_3 = 1$ and $mc_{UoA}(cs^r) = 0 + 0 + 1 + 1 = 2$.

The mc_{UoA} function computes the number of cache misses for a given reaching relative cache state. However, if there is more than one reaching relative cache states (CS^r), the number of cache misses can vary. For cache analysis, we are interested in finding the worst/best case number of cache misses that may be experienced by a basic block. This can be computed by calculating the maximum/minimum number of cache misses. It is defined using the following two functions.

Definition 21 (UoA worst miss count)**.** *We define the function $wmc_{UoA} : 2^{CS^r} \rightarrow \mathbb{N}^0$ where, for a set of relative cache states $CS_1^r \in 2^{CS^r}$ such that $CS_1^r = \{cs_1^r, cs_2^r, \ldots, cs_m^r\}$ and $m = |CS_1^r|$,*

$$wmc_{UoA}(CS_1^r) = MAX(mc_{UoA}(cs_1^r), \ldots, mc_{UoA}(cs_m^r))$$

Definition 22 (UoA best miss count)**.** *We define the function $bmc_{UoA} : 2^{CS^r} \rightarrow \mathbb{N}^0$ where, for a set of relative cache states $CS_1^r \in 2^{CS^r}$ such that $CS_1^r = \{cs_1^r, cs_2^r, \ldots, cs_m^r\}$ and $m = |CS_1^r|$,*

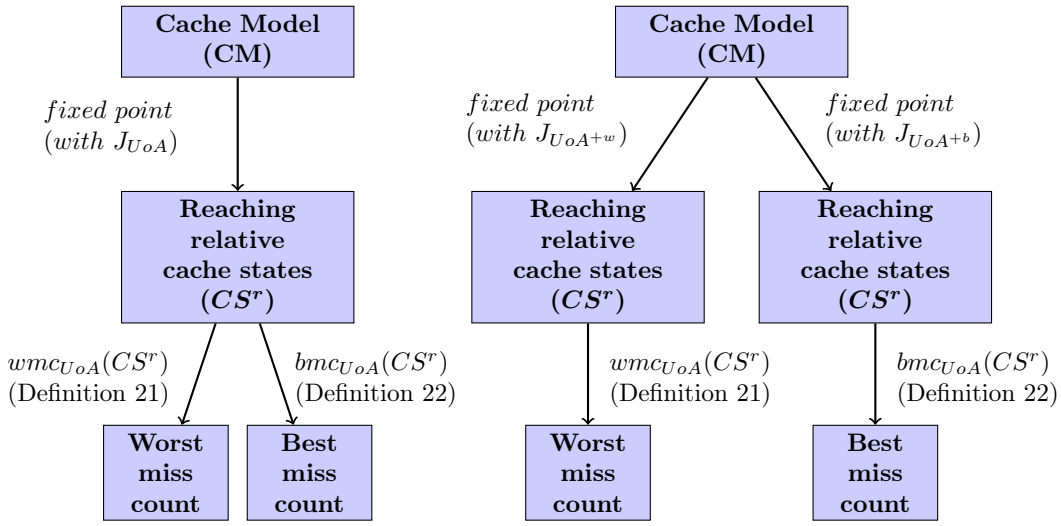
$$bmc_{UoA}(CS_1^r) = MIN(mc_{UoA}(cs_1^r), \ldots, mc_{UoA}(cs_m^r))$$

Illustration
For example, given the reference block $b_{ref} = B8$ and $RCS_{B8}^r = \{cs_1^r, cs_2^r\}$ where $cs_1^r = [\times, 1, 0, 0]$, $cs_2^r = [\times, 0, 1, 1]$ then, $mc_{UoA}(cs_1^r) = 1$ and $mc_{UoA}(cs_2^r) = 2$. For the worst case analysis, the reference block has 2 misses, $wmc_{UoA}(CS_1^r) = MAX(mc_{UoA}(cs_1^r), mc_{UoA}(cs_2^r)) = MAX(1, 2) = 2$. For the best case analysis, the reference block has 1 misses, $bmc_{UoA}(CS_1^r) = MIN(mc_{UoA}(cs_1^r), mc_{UoA}(cs_2^r)) = MIN(1, 2) = 1$.

In summary, for a given reference block $b_{ref} \in B$, we have computed the number of cache misses for the worst case and the best case. To complete the cache analysis, we repeat this process for each basic block in B (lines 1 to 6, Algorithm 6). In the next section, we present an optimisation that can be applied to our approach to reduce the analysis time while maintaining the precision.

6.7 Reducing the analysis time

Figure 17(a) presents our current approach (presented in the previous section). Here, given a cache model CM and a reference block b_{ref}, using a fixed point computation (Algorithm 5) all possible reaching relative cache states of the reference block are computed ($CS_{b_{ref}}^r$). Then using the the functions wmc_{UoA} and bmc_{UoA}, the worst and best case cache misses are computed.



(a) Current UoA approach (b) Optimised UoA approach

Figure 17: UoA approaches for analysing a reference block b_{ref}.

$CS^r = \{cs_1^r, cs_2^r, cs_3^r, cs_4^r\}$	Worst case	Best case
$cs_1^r = [0,1,0,0]$	remove	keep
$cs_2^r = [0,1,0,1]$	keep	remove
$cs_3^r = [0,1,1,0]$	keep	remove
$cs_4^r = [0,1,0,1]$	remove	keep
$cs_5^r = [1,0,0,0]$	keep	keep

Table 8: Reducing cache states based on the worst/best case analysis.

In this section, we propose to customise our current approach to analyse either for the worst case or the best case. Our intuition is that by analysing only the worst case (or the best case) allows us to ignore cache states that are not relevant, i.e, during the worst case analysis, we can ignore cache states that represent the best case. Using this customisation, we may be able to reduce the number of iterations required to reach the fixed point, and may result in faster analysis time, while not effecting the precision.

For example, given $CS^r = \{[1,1,1,1],[0,1,0,1],[0,0,0,0]\}$, during the fix-point computation if we are only analysing for worst case, we can reduce CS^r to $\{[1,1,1,1]\}$ because it captures the worst case (four misses) relative cache state and the rest can be removed. Similarly, for the best case we can reduce CS^r to $\{[0,0,0,0]\}$ because it captures the best case (zero misses) relative cache state and the rest can be removed. Now, consider the CS^r with five possible relative cache states ($CS^r = \{cs_1^r, cs_2^r, cs_3^r, cs_4^r\}$) presented in column 1 of Table 8. To decide on what cache states can be *subsumed* (removed) is not very trivial. The decision on how to remove a relative cache state is based on rest of the relative cache states in the set and the type of the analysis (worst/best). We explain the decision process as follows:

Worst case A cache state cs_k^r is subsumed by cs_j^r, if for all cache lines ($c_i \in C$), when the relative instruction in cs_j^r is same as the instruction in cs_k^r ($cs_j^r[i] = cs_k^r[i]$) or, the relative instruction in cs_j^r captures a cache miss ($cs_j^r[i] = 1$).

For the example in Table 8, for the first three cache line (c_0, c_1, c_2) the relative instructions in cs_2^r and cs_1^r are same. However, for the cache line c_3, $cs_2^r[3] = 1$ and $cs_1^r[3] = 0$. Thus, cs_1^r is subsumed by cs_2^r.

Best case A cache state cs_k^r is subsumed by cs_j^r, if for all cache lines ($c_i \in C$), when the relative instruction in cs_j^r is same as the instruction in cs_k^r ($cs_j^r[i] = cs_k^r[i]$) or, the relative instruction in cs_j^r captures a cache hit ($cs_j^r[i] = 0$).

For the example in Table 8, for the first three cache line (c_0, c_1, c_2) the relative instructions in cs_2^r and cs_1^r are same. However, for the cache line c_3, $cs_2^r[3] = 1$ and $cs_1^r[3] = 0$. Thus, cs_2^r is subsumed by cs_1^r.

The overview of the new approach is presented in Figure 17(b). Given a cache model CM, we optimise the current UoA join function (Definition 17) by creating two new join functions to analyse the best and worst cases. These two functions are described by the following definitions.

Definition 23 (UoA join function for worst case)**.** *The worst case UoA join function* $J_{UoA^{+w}} : 2^{CS^r} \times 2^{CS^r} \rightarrow 2^{CS^r}$ *where for any two relative cache states* $CS_1^r \in 2^{CS^r}$ *and* $CS_2^r \in 2^{CS^r}$, $J_{UoA^{+w}}(CS_1^r, CS_2^r) = CS_3^r$ *where,*

$$CS_3^r = J_{UoA}(CS_1^r, CS_2^r) \setminus \{cs_1^r \in (J_{UoA}(CS_1^r, CS_2^r)) | \exists cs_2^r \in (J_{UoA}(CS_1^r, CS_2^r)) \wedge S_{UoA^{+w}}(cs_1^r, cs_2^r)\}$$

where, $S_{UoA^{+w}} : CS^r \times CS^r \rightarrow \{true, false\}$ *where, for any* $cs_1^r, cs_2^r \in CS^r$,

$$S_{UoA^{+w}}(cs_1^r, cs_2^r) = \begin{cases} true & : if \quad \forall i \in [0, N-1], cs_2^r[i] = cs_1^r[i] \text{ or } cs_2^r = 1 \\ false & : otherwise \end{cases}$$

Definition 24 (UoA join function for best case)**.** *The best case UoA join function* $J_{UoA^{+b}} : 2^{CS^r} \times 2^{CS^r} \rightarrow 2^{CS^r}$ *where for any two relative cache states* $CS_1^r \in 2^{CS^r}$ *and* $CS_2^r \in 2^{CS^r}$, $J_{UoA^{+b}}(CS_1^r, CS_2^r) = CS_3^r$ *where,*

$$CS_3^r = J_{UoA}(CS_1^r, CS_2^r) \setminus \{cs_1^r \in (J_{UoA}(CS_1^r, CS_2^r)) | \exists cs_2^r \in (J_{UoA}(CS_1^r, CS_2^r)) \wedge S_{UoA^{+b}}(cs_1^r, cs_2^r)\}$$

where, $S_{UoA^{+b}} : CS^r \times CS^r \rightarrow \{true, false\}$ *where, for any* $cs_1^r, cs_2^r \in CS^r$,

$$S_{UoA^{+b}}(cs_1^r, cs_2^r) = \begin{cases} true & : if \quad \forall i \in [0, N-1], cs_2^r[i] = cs_1^r[i] \text{ or } cs_2^r = 0 \\ false & : otherwise \end{cases}$$

CS_1^r	CS_2^r	$J_{UoA}(CS_1^r, CS_2^r)$	$J_{UoA^{+w}}(CS_1^r, CS_2^r)$	$J_{UoA^{+b}}(CS_1^r, CS_2^r)$
{[×, 0, ⊥, 1], [×, 1, 0, 1]}	{[×, 0, 0, 1], [×, 0, 1, 0]}	{[×, 0, 0, 1], [×, 1, 0, 1], [×, 0, 1, 0]}	{[×, 1, 0, 1], [×, 0, 1, 0]}	{[×, 0, 0, 1] [×, 0, 1, 0]}

Table 9: Illustration of functions $J_{UoA^{+w}}$ and $J_{UoA^{+b}}$

Using Table 9, we illustrate the functions $J_{UoA^{+w}}$ and $J_{UoA^{+b}}$. Given CS_1^r = {[×, 0, ⊥, 1], [×, 1, 0, 1]}, CS_2^r = {[×, 0, 0, 1], [×, 0, 1, 0]}, to compute the $J_{UoA^{+w}}(CS_1^r, CS_2^r)$ and $J_{UoA^{+b}}(CS_1^r, CS_2^r)$ we first check for subsumed unknown (⊥) relative cache states using $J_{UoA}(CS_1^r, CS_2^r)$. In this case, $J_{UoA}(CS_1^r, CS_2^r)$ = {[×, 1, 0, 1], [×, 0, 0, 1], [×, 0, 1, 0]}. For worst case analysis, $J_{UoA^{+w}}(CS_1^r, CS_2^r) = J_{UoA^{+w}}$ {[×, 1, 0, 1], [×, 0, 0, 1], [×, 0, 1, 0]} = {[×, 1, 0, 1], [×, 0, 1, 0]}. Here,

we remove relative cache state $[\times, 0, 0, 1]$ as it is subsumed by the cache state $[\times, 1, 0, 1]$. Similarly, for best case analysis, $J_{UoA^{+b}}(CS_1^r, CS_2^r) = J_{UoA^{+b}}\{[\times, 1, 0, 1], [\times, 0, 0, 1], [\times, 0, 1, 0]\} = \{[\times, 0, 0, 1], [\times, 0, 1, 0]\}$. Here, we remove relative cache state $[\times, 1, 0, 1]$ as it is subsumed by the cache state $[\times, 0, 0, 1]$.

As illustrated in Figure 17(b), for the worst (or best) case analysis, using the function $J_{UoA^{+w}}$ (or $J_{UoA^{+b}}$) we should be able to reduce the analysis time and the memory footprint, and most importantly there should be no change in the precision compared to the non optimised approach (Figure 17(a)). For example, if we analyse the set $CS_3^r = J_{UoA}(CS_1^r, CS_2^r) = \{[\times, 1, 0, 1], [\times, 0, 0, 1], [\times, 0, 1, 0]\}$ for the worst case we get two misses (using $wmc_{UoA}(CS_3^r) = 2$, Definition 21) and for the best case we get one miss (using $bmc_{UoA}(CS_3^r) = 1$, Definition 22). This precision (worst/best miss count) is the same as $J_{UoA^{+w}}([\times, 1, 0, 1], [\times, 0, 1, 0]) = 2$ and $J_{UoA^{+b}}([\times, 0, 0, 1], [\times, 0, 1, 0]) = 1$. Thus we illustrate the precision is the same for both approaches (Figure 17(a) and Figure 17(b)).

In summary, using the fixed point computation (Algorithm 5) we have computed the reaching relative cache states of the reference block. During the fixed point computation, the optimisations presented in Section 6.7 reduces the analysis time, while not effecting the precision. Finally, as described in Section 6.6, we compute the number of cache misses for the worst case and the best case. In Table 10, we present a summary of the symbols and the definitions that were used in this section.

Symbol/ Definition	**Description**	**Example**
Reduce	Given a CM and block b_{ref}, computes the reduced graph G^r	See Section 6.2
$cs^r \in \{1, 0, \top, \bot, \times\}^N$	Relative cache state w.r.t to b_{ref}	$cs^r = [\times, 0, 1, \bot]$
$cs^r[i] = \times$	*not of interest*	$cs^r[0] = \times$
$cs^r[i] = 1$	*different*	$cs^r[0] \neq BI(b_{ref})[0]$
$cs^r[i] = 0$	*same*	$cs^r[0] = BI(b_{ref})[0]$
$cs^r[i] = \top$	*no instruction*	$cs^r[0] = \top$
$cs^r[i] = \bot$	*unknown instruction*	Given $CI(0) = \{m1, m5\}$, $cs^r[0] = \{m1, m5\top\}$
RCS^r_b	Set of relative cache states prior to the execution of block b	In Figure 14, $RCS^r_{B1} = \{[\times, \top, \top, \top]\}$
LCS^r_b	Set of relative cache states after the execution of block b	In Figure 14, $LCS^r_{B1} = \{[\times, 1, 0, 0]\}$
$J_{UoA} : 2^{CS^r} \times 2^{CS^r} \to 2^{CS^r}$	The UoA join function	Given $CS^r_1 = \{[\times, 0, 1, 1]\}$ and $CS^r_2 = \{[\times, 0, 0, 0]\}$, $J_{UoA}(CS^r_1, CS^r_2) = \{[\times, 0, 1, 1], [\times, 0, 0, 0]\}$
$T_{UoA} : B \times CS^r \to CS^r$	The UoA transfer function	Given $BI^r(b) = [\mp, 1, 0, \mp]$ and $RCS^r1 = \{[1, 0, 1, 0]\}$, $T_{UoA}(b, RCS^r1) = \{[1, 1, 0, 0]\}$
$mc_{UoA} : CS^r \to \mathbb{N}^0$	UoA miss count function	Given $cs^r = [\times, 0, 1, 1]$ then $mc_{UoA}(cs^r) = 0 + 0 + 1 + 1 = 2$.
$wmc_{UoA} : 2^{CS^r} \to \mathbb{N}^0$	UoA worst miss count	Given $wmc_{UoA}(\{[\times, 1, 0, 0], [\times, 0, 1, 1]\ \}) = 2$
$bmc_{UoA} : 2^{CS^r} \to \mathbb{N}^0$	UoA best miss count	Given $wmc_{UoA}(\{[\times, 1, 0, 0], [\times, 0, 1, 1]\ \}) = 1$
$J_{UoA^{+w}} : 2^{CS^r} \times 2^{CS^r} \to 2^{CS^r}$	UoA join function for worst case	$J_{UoA^{+w}}\{[\times, 1, 0, 1], [\times, 0, 0, 1], [\times, 0, 1, 0]\ \} = \{[\times, 1, 0, 1], [\times, 0, 1, 0]\}$
$J_{UoA^{+b}} : 2^{CS^r} \times 2^{CS^r} \to 2^{CS^r}$	UoA join function for best case	$J_{UoA^{+b}}\{[\times, 1, 0, 1], [\times, 0, 0, 1], [\times, 0, 1, 0]\ \} = \{[\times, 0, 0, 1], [\times, 0, 1, 0]\}$

Table 10: Some of the symbols and the definitions presented in this section.

7 Comparisons between the NUS, Absint and UoA approaches

In this section, we compare the three approaches by comparing their computed concrete reaching cache states and the precision. The NUS approach directly computes the concrete reaching cache states of a block, during its fixed point computation (Algorithm 1). The Absint approach computes the abstract reaching cache states and then maps them into the concrete cache states, using the mapping function MAP_{Absint} (Definition 13). Similarly, the proposed approach computes the reaching relative cache states of a block using the fixed point computation (Algorithm 5). However, we have not yet described how the relative cache states are mapped into the concrete cache states. We need this mapping to facilitate a better comparison between the three approaches.

We start by presenting a function that maps the relative cache states to the concrete cache states (in Section 7.1). Later, we present a detailed comparison between the three approaches (in Section 7.2).

7.1 Mapping relative cache states of b_{ref} to concrete cache states

In this section, we present how the reaching relative cache states for the reference block can be converted into concrete reaching cache states. The mapping function, called MAP_{UoA}, is defined in Definition 27. However, the following notions/definitions are essential to describe the mapping function.

A relative cache state cs^r is a vector $[inst_0^r, inst_1^r, \ldots, inst_{N-1}^r]$ and for each cache line $c_i \in C$, $inst_i^r \in \{\times, 0, 1, \top, \bot\}$. Here, each element $inst_i^r$ is described in relation to the instruction on cache line c_i ($BI(b_{ref})[i]$) in the block b_{ref}. These relations can be translated into a set of constraints. We explain these constraints as follows.

- $inst_i^r = \times$ describes that the instruction on the cache line c_i is unconstrained. The instruction in the cache line c_i can be any possible instruction from the set $CI(c_i)$ or can be empty ($\top$). E.g., given $CI(c_0) = \{m1, m5\}$ and $inst_0^r = \times$, then the instruction on cache line c_i could represent any instruction from $\{\top\} \cup CI(c_0) = \{m1, m5, \top\}$.

- $inst_i^r = 0$ describes the constraint that the instruction in the cache must be $BI(b_{ref})[i]$. E.g., given $inst_0^r = 0$ and $BI(b_{ref})[0] = m1$, then the only possible instruction in cache line c_0 is $m1$.

- $inst_i^r = 1$ describes the constraint that the instruction in the cache is not $BI(b_{ref})[i]$. Thus, the instruction can only belong to $\{\top\} \cup CI(c_i) \setminus \{BI(b_{ref})[i]\}$. E.g., given $CI(c_0) = \{m1, m5\}$, $inst_0^r = 1$ and $BI(b_{ref})[0] = m1$, then the instruction in the cache belongs to $CI(c_0) \cup \{\top\} \setminus \{m1\} = \{\top, m5\}$.

- $inst_i^r = \top$ describes that there is no instruction ($\top$) on the cache line c_i. E.g., given $inst_0^r = \top$, then there is no instruction on cache line c_0. It is empty ($\top$).

- $inst_i^r = \bot$ describes that there is an instruction on the cache line c_i and it must belong to the set ($CI(c_i)$). E.g., given $CI(c_0) = \{m1, m5\}$ and $inst_0^r = \bot$, then the instruction in cache line c_0 could represent any instruction from $\{m1, m5\}$.

Hence, given a cache model CM, a cache line $c_i \in C$, an element of the relative cache state ($inst_i^r$) and, the reference block b_{ref}, we can compute the corresponding set of possible concrete instructions, for the cache line c_i. This computation is described using the following function.

Definition 25 (Relative instruction to concrete instructions mapping)**.** *Given a cache model CM and the reduced graph $G^r = \langle B^r, b_{init}, E^r \rangle$, the relative instruction to concrete instructions mapping function $MAPinst_{UoA} : C \times \{\times, 0, 1, \top, \bot\} \times B^r \rightarrow 2^{I \cup \{\top\}}$ where, for a given cache line $c_i \in C$, a relative instruction $inst_i^r \in \{\times, 0, 1, \top, \bot\}$, and a block $b \in B^r$.*

$$MAPinst_{UoA}(c_i, inst_i^r, b) = \begin{cases} \{\top\} \cup CI(c_i) & : if \quad inst_i^r = \times \\ \{BI(b)[i]\} & : if \quad inst_i^r = 0 \\ \{\top\} \cup CI(c_i) \setminus \{BI(b)[i]\} & : if \quad inst_i^r = 1 \\ \{\top\} & : if \quad inst_i^r = \top \\ CI(c_i) & : if \quad inst_i^r = \bot \end{cases}$$

Illustration

Using Table 11, we illustrate the function $MAPinst_{UoA}$. For the cache line c_0, given that the instruction executed by the block b_{ref} is $m1$ ($BI(b_{ref})[0] = m1$) and $CI(c_0) = \{m1, m5\}$, for each possible relative instruction $inst_0^r$, we describe the mapping as follows.

$\boldsymbol{inst_0^r}$	$\boldsymbol{MAPinst_{UoA}(c_0, inst_0^r, b_{ref})}$
$\times$	$\{\top\} \cup CI(c_0) = \{\top\} \cup \{m1, m5\} = \{\top, m1, m5\}$
0	$\{BI(b_{ref})[0]\} = \{m1\}$
1	$\{\top\} \cup CI(c_0) \setminus \{BI(b_{ref})[0]\} = \{\top\} \cup \{m1, m5\} \setminus \{m1\} = \{\top, m5\}$
$\top$	$\{\top\}$
$\bot$	$CI(c_0) = \{m1, m5\}$

Table 11: Illustration of the function $MAPinst_{UoA}$ w.r.t cache line c_0, $BI(b_{ref})[0] = m1$ and $CI(c_0) = \{m1, m5\}$

- $\boldsymbol{inst_0^r = \times}$ (row 1) describes all possible instructions $MAPinst_{UoA}(c_0, \times, b_{ref}) = \{\top\} \cup CI(c_0) = \{m1, m5, \top\}$.
- $\boldsymbol{inst_0^r = 0}$ (row 2) describes the instruction on the cache can only be $m1$, $MAPinst_{UoA}(c_0, 0, b_{ref}) = \{BI(b_{ref})[0]\} = \{m1\}$.
- $\boldsymbol{inst_0^r = 1}$ (row 3) describes the instruction on the cache can not be $m1$, $MAPinst_{UoA}(c_0, 1, b_{ref}) = \{\top\} \cup CI(c_0) \setminus \{BI(b_{ref})[0]\} = \{\top\} \cup \{m1, m5\} \setminus \{m1\} = \{\top, m5\}$.
- $\boldsymbol{inst_0^r = \top}$ (row 4) describes the instruction on the cache is empty, $MAPinst_{UoA}(c_0, \top, b_{ref}) = \{\top\}$.
- $\boldsymbol{inst_0^r = \bot}$ (row 5) describes the cache has an unknown instruction, $MAPinst_{UoA}(c_0, \bot, b_{ref}) = CI(c_0) = \{m1, m5\}$.

As discussed above, a relative instruction $inst_i^r$ could map to more than one instruction. For the cache model presented in Figure 2 (running example), given a relative cache state $cs^r = [\times, 1, 0, 0]$, the reference block with $BI(b_{ref}) = [\mp, m6, m3, m4]$, for cache lines c_0, c_1, c_2, c_3 using the mapping function $MAPinst_{UoA}$, we get the sets $\{\top, m1, m5\}$, $\{\top, m2\}$, $\{m3\}$, $\{m4\}$ respectively.

This represents that c_0 contains $\top$, $m1$, or $m5$, c_1 contains $\top$ or $m6$, c_2 contains $m3$, c_3 contains $\top$ or $m8$. We can describe this as the abstract cache state (described in Definition 10)

$acs = [\{\top, m1, m5\}, \{\top, m2\}, \{m3\}, \{\top, m8\}]$. Then using the abstract cache state to cache state mapping function (described in Definition 13), we get the set of cache states cs as,

$$\{[\top, \top, m3, m4], [\top, m2, m3, m4], [m1, \top, m3, m4],$$

$$[m1, m2, m3, m4], [m5, \top, m3, m4], [m5, m2, m3, m4]\}$$

This translation is described by the following function.

Definition 26 (Relative cache state to cache state mapping)**.** *Given a cache model CM, the UoA mapping function is $MAP_{UoA} : CS^r \times B^r \rightarrow 2^{CS}$, where for a given relative cache state $cs^r \in CS^r$ and $b \in B^r$.*

$MAP_{UoA}(cs^r, b) = set_0 \times set_1 \times \ldots \times set_{N-1}$ *where,* $set_i = MAPinst_{UoA}(c_i, cs^r[i], b)$.

Illustration
For example, given $cs^r = [\times, 1, 0, 0]$, and the reference block b_{ref} is $B8$ with $BI(b_{ref}) = [\mp, m6, m3, m4]$ then,

$$\begin{aligned} MAP_{UoA}(cs^r, b_{ref}) &= MAP_{UoA}([\times, 1, 0, 0], B8) \\ &= set_0 \times set_1 \times set_2 \times set_3 \\ &= \{\top, m1, m5\} \times \{\top, m2\} \times \{m3\} \times \{m4\} \\ &= \{[\top, \top, m3, m4], [\top, m2, m3, m4], [m1, \top, m3, m4], \\ &\quad [m1, m2, m3, m4], [m5, \top, m3, m4], [m5, m2, m3, m4]\} \end{aligned}$$

The mapping function, as defined in Definition 26, translates a relative cache state cs^r into a set of concrete cache state CS. However, a block my have more than one reaching relative cache states CS^r. In this case, the mapping of the cache states is an union of the possible mapping for each reaching relative cache state, i.e., given $CS^r = \{cs_1^r, cs_2^r\}$, then mapping to the cache sates is $MAP_{UoA}(cs_1^r, b_{ref}) \cup MAP_{UoA}(cs_2^r, b_{ref})$. We define this mapping using the following function.

Definition 27 (Extended UoA mapping function)**.** *The mapping function $MAP_{UoA} : 2^{CS^r} \times B^r \rightarrow 2^{CS}$ where for a given $b \in B^r$ and a set of relative cache states $CS^r \in 2^{CS^r}$. Then, for all relative cache states $cs_i^r \in CS^r$ where $i \in [1, |CS^r|]$,*

$$MAP_{UoA}(CS^r, b) = \bigcup_{i=1}^{|CS^r|} \{MAP_{UoA}(cs_i^r, b_{ref})\}$$

Illustration

For example, given the reference block b_{ref} is $B8$ with $BI(b_{ref}) = [\mp, m6, m3, m4]$, and after the fixed point $RCS_{b_{ref}}^r = \{cs_1^r, cs_2^r\}$ where $cs_1^r = [\times, 1, 0, 0]$, $cs_2^r = [\times, 0, 1, 1]$ then, $MAP_{UoA}(RCS_{b_{ref}}^r, b_{ref}) = MAP_{UoA}(cs_1^r, b_{ref}) \cup MAP_{UoA}(cs_2^r, b_{ref})$. We start by illustrating the mapping of individual relative cache states (cs_1^r, cs_2^r), and later present the mapping for the set of relative cache states $(RCS_{b_{ref}}^r)$.

$$
\begin{aligned}
MAP_{UoA}(cs_1^r, b_{ref}) &= MAP_{UoA}([\times, 1, 0, 0], B8) \\
&= \{\top, m1, m5\} \times \{\top, m2\} \times \{m3\} \times \{m4\} \\
&= \{[\top, \top, m3, m4], [\top, m2, m3, m4], [m1, \top, m3, m4], \\
&\quad [m1, m2, m3, m4], [m5, \top, m3, m4], [m5, m2, m3, m4]\} \\
MAP_{UoA}(cs_2^r, b_{ref}) &= MAP_{UoA}([\times, 0, 1, 1], B8) \\
&= \{\top, m1, m5\} \times \{m6\} \times \{\top, m7\} \times \{\top, m8\} \\
&= \{[\top, m6, \top, \top], [\top, m6, \top, m8], [\top, m6, m7, \top], [\top, m6, m7, m8], \\
&\quad [m1, m6, \top, \top], [m1, m6, \top, m8], [m1, m6, m7, \top], [m1, m6, m7, m8], \\
&\quad [m5, m6, \top, \top], [m5, m6, \top, m8], [m5, m6, m7, \top], [m5, m6, m7, m8]\}
\end{aligned}
$$

$$
\begin{aligned}
MAP_{UoA}(RCS^r_{b_{ref}}, b_{ref}) &= MAP_{UoA}(cs_1^r, b_{ref}) \cup MAP_{UoA}(cs_2^r, b_{ref}) \\
&= \{[\top, \top, m3, m4], [\top, m2, m3, m4], [m1, \top, m3, m4], \\
&\quad [m1, m2, m3, m4], [m5, \top, m3, m4], [m5, m2, m3, m4]\} \\
&\quad \cup \{[\top, m6, \top, \top], [\top, m6, \top, m8], [\top, m6, m7, \top], [\top, m6, m7, m8], \\
&\quad [m1, m6, \top, \top], [m1, m6, \top, m8], [m1, m6, m7, \top], [m1, m6, m7, m8], \\
&\quad [m5, m6, \top, \top], [m5, m6, \top, m8], [m5, m6, m7, \top], [m5, m6, m7, m8]\} \\
&= \{[\top, \top, m3, m4], [\top, m2, m3, m4], [m1, \top, m3, m4], [m1, m2, m3, m4], \\
&\quad [m5, \top, m3, m4], [m5, m2, m3, m4], [\top, m6, \top, \top], [\top, m6, \top, m8], \\
&\quad [\top, m6, m7, \top], [\top, m6, m7, m8], [m1, m6, \top, \top], [m1, m6, \top, m8] \\
&\quad [m1, m6, m7, \top], [m1, m6, m7, m8], [m5, m6, \top, \top], [m5, m6, \top, m8] \\
&\quad [m5, m6, m7, \top], [m5, m6, m7, m8]\}
\end{aligned}
$$

Finally, we have presented the mapping function MAP_{UoA} (Definition 27), which computes the corresponding concrete cache states from a given set of relative cache states. Next, we present a detailed comparison between the NUS, Absint and the proposed approaches.

7.2 Comparison between the approaches

In this section, we start by presenting a summary of the three approaches that were presented in this chapter.The NUS approach (see Section 3) and the Absint approach (see Section 4) solve the cache analysis problem (Definition 6) by analysing all basic blocks in a single fixed point computation. In contrast, the proposed (UoA) approach (see Section 6) analyses one basic block at a time. Given a cache model CM, using Figure 18, we discuss how the three approaches analyse a basic block (b). Before we discuss the approaches, we present the meaning of the different components of the figure.

A *domain* is represented as an ellipse. For example, the figure has four domains: (1) Domain 1 that represents all possible relative cache states, (2) Domain 2 that represents all possible abstract cache states, (3) Domain 3 that represents all possible concrete cache states, and (4) Domain 4 that represents all possible precisions (the pairs (wmc, bmc)) for block b. A *set* of elements inside a domain is represented as a circle. An *element* is represented as a point.

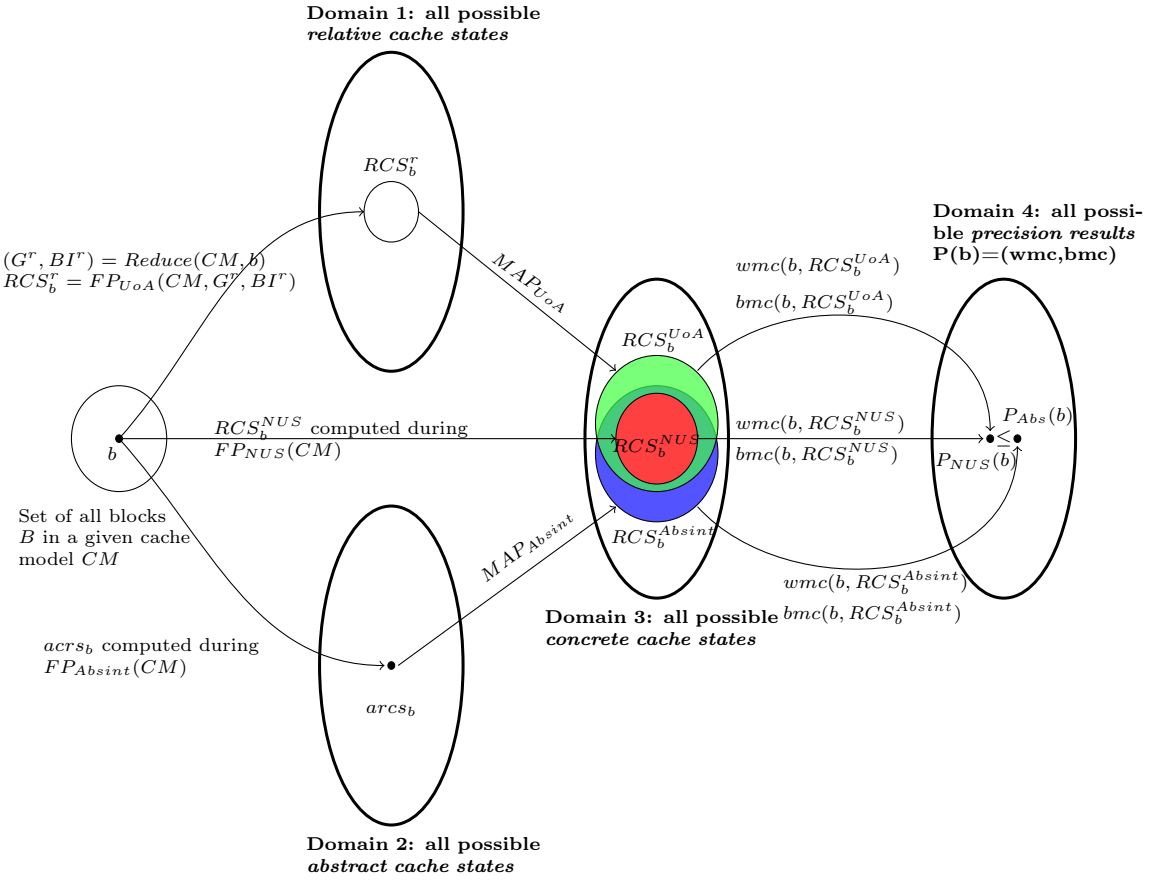


Figure 18: Overview of NUS, Absint and UoA approaches for computing the worst/best cache misses for a block b

NUS approach: Given a cache model CM,

1. Using the fixed point computation (Algorithm 1), the set of all possible *concrete* reaching cache states, of all basic blocks, is computed. For the basic block b, the set is represented by the circle RCS_b^{NUS} in Domain 3.
2. Then, using the functions *wmc* and *bmc* (Definitions 4 and 5), we compute the number of cache misses in the worst/best case. For the computed reaching cache states (RCS_b^{NUS}), the precision is represented by the point $P_{NUS}(b)$ in Domain 4.

Absint approach: Given a cache model CM,

1. Using the fixed point computation (Algorithm 2), the possible *abstract* reaching cache state, of all basic blocks, is computed. The abstract reaching cache state for the basic block b, is represented by the point $arcs_b$, in Domain 2.
2. Then, using the function MAP_{Absint} (Definition 13), the abstract cache state is mapped to a set of concrete cache states, represented by the circle RCS_b^{Absint}, in Domain 3.
3. Then, using the functions *wmc* and *bmc* (Definitions 4 and 5), we compute the number of cache misses in the worst/best case. For the concrete reaching cache states (RCS_b^{Absint}), the precision is represented by the point $P_{Abs}(b)$, in Domain 4.

UoA approach: Given a cache model CM,

1. For the reference basic block b, we first compute the reduced graph G^r and the relative instructions mapping function BI^r (Algorithm 4). Then, using the fixed point computation (Algorithm 5), we compute the set of all possible reaching *relative* cache states. The set is represented by the circle RCS_b^r, in Domain 1.
2. Then, using the function MAP_{UoA} (Definition 27), the set of relative cache states are mapped to a set of concrete cache states, represented by the circle RCS_b^{UoA}, in Domain 3.
3. Then, using the functions *wmc* and *bmc* (Definitions 4 and 5) we compute the number of cache misses in the worst/best case. For the set of reaching cache states (RCS_b^{UoA}), the precision is same as the NUS approach. Thus, the precision is also represented by the point $P_{NUS}(b)$, in Domain 4.

We now compare the three approaches by first comparing the computed concrete cache states (Domain 3) and then the precision (Domain 4).

Comparing the concrete cache states: In Domain 3, we illustrate that: (1) the concrete cache states corresponding to the NUS approach (RCS_b^{NUS}) is always a subset (or equal) of the concrete cache states corresponding to the Absint and the UoA approaches ($RCS_b^{NUS} \subseteq RCS_b^{Absint}$ and $RCS_b^{NUS} \subseteq RCS_b^{UoA}$) and, (2) the concrete cache states corresponding to the Absint and the UoA approaches may not be equal ($RCS_b^{Absint} \neq RCS_b^{UoA}$).

We illustrate this idea using the example cache model (Figure 2). For the basic block $B8$, Table 12 presents comparison of the three approaches. For the NUS approach (row 1), the concrete reaching cache states of $B8$ are presented in column 3. For the Absint approach (row 2), the abstract reaching cache state of $B8$ and the corresponding concrete cache states are presented in columns 2 and 3 respectively. It contains 8 possible concrete cache states. Similarly, for the UoA approach, the reaching relative cache states and the corresponding

concrete cache states are presented in columns 2 and 3 respectively. It contains 18 possible concrete cache states.

As stated earlier, we observe that the set of concrete cache states corresponding to the NUS approach (only 2 cache states) is a subset of the concrete cache states corresponding to the Absint (8 cache states) and the UoA (18 cache states) approaches. This shows that the NUS approach yields the smallest set of possible reaching cache states.

Comparing the precision: In Domain 4, we presented the precision of the NUS, Absint and UoA approaches. We illustrate that *always* the precision computed by the UoA approach is the same as the NUS approach and, the precision from the Absint approach is always less than or equal to the NUS/UoA approaches ($P_{NUS}(b) \leq P_{Abs}(b)$). We illustrate this idea using Table 12. For the block $B8$ ($BI(B8) = [\mp, m6, m3, m4]$), given the set of possible reaching concrete cache states for each approach (in column 3), the precision (wmc, bmc) are presented in columns 4 and 5 respectively. As stated earlier, the precision from the NUS approach $(2, 1)$ is the same as the UoA approach $(2, 1)$.

Also, we observe that the precision of the Absint approach is$(3, 0)$. For the worst case, the wmc for the Absint is greater than the NUS approach ($3 > 2$), making Absint approach less precise. Similarly, for the best case, the bmc for the Absint is smaller than the NUS approach ($0 < 1$), making Absint approach less precise. Thus, as illustrated in Domain 4, the NUS approach is more precise than the Absint approach ($P_{NUS}(B8) = (2, 1) \leq (3, 0) = P_{Abs}(B8)$).

Approach	$\boldsymbol{arcs_{B8}/RCS^{r}_{B8}}$	$\boldsymbol{RCS_{B8}}$	$\boldsymbol{wmc(B8)}$ (Worst)	$\boldsymbol{bmc(B8)}$ (Best)
NUS		$[m5, m2, m3, m4]$ $[m5, m6, m3, m4]$	2	1
Absint	$\{m5\}, \{m2, m6\}, \{m3, m7\}, \{m4, m8\}$	$[m5, m2, m3, m4]$ $[m5, m2, m3, m8]$ $[m5, m2, m7, m4]$ $[m5, m2, m7, m8]$ $[m5, m6, m3, m4]$ $[m5, m6, m3, m8]$ $[m5, m6, m7, m4]$ $[m5, m6, m7, m8]$	3	0
UoA	$\{[\times 100]\ [\times 011]\}$	$[\top, \top, m3, m4]$, $[\top, m2, m3, m4]$, $[m1, \top, m3, m4]$, $[m1, m2, m3, m4]$, $[m5, \top, m3, m4]$, $[m5, m2, m3, m4]$, $[\top, m6, \top, \top]$, $[\top, m6, \top, m8]$, $[\top, m6, m7, \top]$, $[\top, m6, m7, m8]$, $[m1, m6, \top, \top]$, $[m1, m6, \top, m8]$, $[m1, m6, m7, \top]$, $[m1, m6, m7, m8]$, $[m5, m6, \top, \top]$, $[m5, m6, \top, m8]$, $[m5, m6, m7, \top]$, $[m5, m6, m7, m8]$	2	1

Table 12: Comparing the cache states and the precision between the NUS, the Absint and the UoA approaches as we analyse block $B8$ ($BI(B8) = [\mp, m6, m3, m4]$).

Finally, using Table 13, we present a qualitative comparison between the three approaches. (1) The NUS approach analyses all blocks using a single fixed point computation. It directly computes all possible reaching concrete cache states of all blocks, because it does not introduce any abstractions. The lack of abstraction yields high precision, but longer analysis time. (2) The Absint approach analyses all blocks using a single fixed point computation. However, the notion of abstract cache states is introduced which abstracts the relation between cache lines. This allows for faster analysis time, but sacrifices precision. (3) Finally, the UoA approach analyses one block at a time. It introduces the notion of relative cache states. This abstraction does not sacrifice the precision (same as NUS). Later in Section 9, using experimental results, we quantitatively compare the three approaches in terms of precision and analysis time.

Approach	**Fixed point**	**Cache States**	**Precision**	**Analysis time**	**Optimisation**
NUS	all blocks	concrete (cs)	high	slow	none
Absint	all blocks	abstract (acs)	low	fast	merge cache states
UoA	one block at a time	relative (cs^r)	high (same as NUS)	medium	(1) reduce graph (2) reduce cache lines (3) relative instructions

Table 13: Qualitative comparing the precision between the NUS, the Absint and the UoA approaches.

8 Discussion

We started by formalising the cache model CM (presented in Definition 1). The model contains the instructions (I) and the set of cache lines (C), where each instruction is mapped to a single cache line. We assume that whenever there is a cache miss, only one instruction is fetched from the main memory. However, in practice, this assumption is not true. Whenever there is a cache miss, a set of instructions, called *memory block*, are fetched from the main memory and stored in a cache line. This execution model is identical to the analytical model presented by the NUS approach [12]. In order to map the execution model to the cache model presented in this chapter, we treated each memory block as an instruction. Then, each cache line stored one *memory block* instead of one instruction.

To fairly compare the three approaches, we presented the Absint approach in a slightly different manner to match the NUS and the UoA approaches. However, the differences do not affect the core technique of the Absint approach [4]. The Absint approach as described in [4], presents two different analyses. The *May/Must analysis* computes the instructions that may/must be in the reaching cache state of a basic block. This analysis is used to compute the number of cache misses in the best/worst case. However, given the results of the May analysis, both the worst case and the best case cache misses can be derived, avoiding the need for Must analysis. Thus, in this chapter, during the fixed point computation (Algorithm 2) we only illustrated the May analysis. Also, during the initialisation of the fixed point computation in the Absint approach, an empty set is used to represent an unknown instruction. However, to maintain consistent terminology w.r.t the NUS approach, we used the symbol $\perp$.

One cache analysis approach that we did not compare with is the Cache Conflict Graphs [7,8]. This approach analyses the number of cache misses in each block, one cache line at a time. For our example (Figure 2), the cache model has four cache lines c_0, c_1, c_2 and c_3. We would generate four graphs, one for each cache line c_i, where each graph captures the instructions that are mapped to the cache line c_i. Then, by traversing each graph and looking at the predecessor basic block, the cache hit (or miss) of each basic block can be computed for the particular cache line. A cache hit occurs when the instruction appears in the predecessor block, otherwise, it is a

cache miss. We believe, by analysing one cache line at a time, the relation between cache lines is abstracted away. This abstraction is similar to the *join* function (Definition 11) of the Absint approach. Thus, we would expect the precision of the Cache Conflict Graphs to be identical to the precision of the Absint approach. However, it would be interesting to compare the analysis time.

9 Benchmarking and Experimental Results

In this section, we compare the precision and computation time between the NUS, the Absint, and the UoA approaches. Here, we compare the WCRT, BCRT, and the analysis time over a set of benchmarks.

For the experiments, we have selected five examples from [6] that model control applications. These examples are developed using the model-driven engineering approach defined by the IEC 61499 international standard for describing industrial process-control systems [**?**]. It prescribes the use of function blocks to facilitate a component-oriented approach to software development for industrial process-control systems. Each example contains a collection of *function* blocks and, all blocks are executed in a round-robin fashion to emulate the semantics of scan cycles of programmable logic controllers (PLC). The program runs periodically in a sequence of *scan cycles*. This is similar to the execution of PRET-C program that runs periodically in a sequence of *ticks*. Using an automated tool, similar to the work presented in [6], we translate each function block into a PRET-C thread. *EOT* statements are then used to mark the computation boundaries for each scan cycle.

All five examples are presented in last five rows in Table 14. The first column presents the example followed by its description (column 2). The number of PRET-C lines of the program and its memory footprint is presented in columns 3 and 4 respectively. The largest example are CruiseController and RailRoadCrossing with more than 4000 lines of PRET-C code. We have also created two additional examples: a BubbleSort program and a Synthetic example. The latter example is introduced to highlight the difference between the NUS, Absint and UoA approaches.

Example	**Description**	**Number of PRET-C lines**	**Size (source file)**
BubbleSort	Bubble sort program.	128	2KB
Synthetic	Contains branching and loops	180	4KB
Flasher	Control for distributed lights	384	9KB
DrillStation	Controlling a drilling station	1800	62KB
ConvBelt	Airport baggage conveyor belt	1280	44KB
RailRoadCrossing	Rail road intersection cnt.	4613	163KB
CruiseController	Cruise control model	4194	146KB

Table 14: Benchmark programs and their characteristics.

9.1 Benchmarking

In our experiment, we compare WCRT, BCRT and analysis time of Absint, the UoA, and the NUS approaches. The results are presented in Table 15. For this experiment, we have configured the cache size to be 1% of the program size. We have chosen this cache size because, in practice, the size of the cache is significantly smaller that the program size. For example, the MicorBlaze

processor has a main memory of size 128 MB and, the cache size can be configured from anything between 128 bytes to 128 KB (0.001% to 1% of the mian memory) [1].

The WCRT, BCRT, and the analysis time (in seconds) for Absint is presented in columns 2, 3, and 4 respectively. Similarly, the following columns present results for the UoA and NUS approaches. For most examples, the NUS approach takes longer than 12 hours. In this case, we were not able to calculate the WCRT and the BCRT values. This was represented using "T.O" (Time Out). Finally, the last column of the table presents the gain in precision on the WCRT estimate of the UoA approach, compared to the Absint approach.

For WCRT analysis, across all the benchmarks, we observe that the estimates from the UoA approach is always less than or equal to the estimates from the Absint approach. Also, the estimates from the UoA approach are always equal to the estimates from the NUS approach. This shows that the UoA approach does not lose any precision.

For BCRT analysis, across all the benchmarks, we observe that the estimates from the UoA approach is always greater than or equal to the estimates from the Absint approach. Also, the estimates from the UoA approach are always equal to the estimates from the NUS approach. This shows that the UoA approach does not lose any precision.

The abstractions presented for UoA approach also significantly reduces the analysis time, when compared to the NUS approach (see earlier comparison in Section 7). For most of the examples, the NUS approach does not finish (more than 12 hours) its computation, while the UoA approach takes less than 3 minutes. However, the UoA approach is slightly longer than the Absint approach.

For the biggest example (RailRoadCrossing), the UoA approach is 24% (1.24 in the last column) tighter then the Absint approach. More importantly, this experiment shows that, on an average, the UoA approach is more precise (less abstract) than the Absint approach. At the same time, it is always as precise (i.e., no loss in precision) as the NUS approach, while being significantly faster than the NUS approach, and marginally longer than the Absint approach. This idea is illustrated using Figure 19. Overall, this is a significant result that supports the contribution (UoA approach as a new cache analysis technique) of this chapter.

	Absint			**UoA**			**NUS**			**Gain**
Example	**WCRT (clks)**	**BCRT (clks)**	**AT (sec)**	**WCRT (clks)**	**BCRT (clks)**	**AT (sec)**	**WCRT (clks)**	**BCRT (clks)**	**AT (sec)**	**(col2/ col5)**
RailRoadCrossing	307389	7090	4.0	249609	7270	142.0	T.O	T.O	T.O	1.23
Flasher	117292	1110	1.8	95692	1110	41.9	T.O	T.O	T.O	1.22
CruiseController	356270	5679	3.6	287438	5679	113.7	T.O	T.O	T.O	1.24
ConvBeltModel	21200	5716	1.7	17960	7660	35.1	T.O	T.O	T.O	1.18
DrillStation	31341	8564	1.9	26913	13028	45.1	T.O	T.O	T.O	1.16
BubbleSort	2571	44	0.7	2571	44	36.6	2571	44	0.4	1
Synthetic	78015	311	1.4	77835	311	8.2	77835	311	0.8	1.0
Average	**130583**	**4073**	**2.2**	**108288**	**5015**	**60.4**	**NA**	**NA**	**NA**	**1.15**

Table 15: Quantitative comparison between Absint, UoA, and NUS approaches

10 Conclusions

In this paper, we have formally described the cache analysis model and presented three approaches for solving it. First, the NUS approach, which does not introduce any abstractions resulting in a very precise, but very slow analysis. In contrast, the Absint approach abstracts the relation between the cache lines. This allows for faster analysis time, but sacrifices precision. Finally, we have presented a new approach, called UoA approach, which abstracts the instructions and cache states w.r.t. to a reference block. This does not sacrifice any precision. Experimental

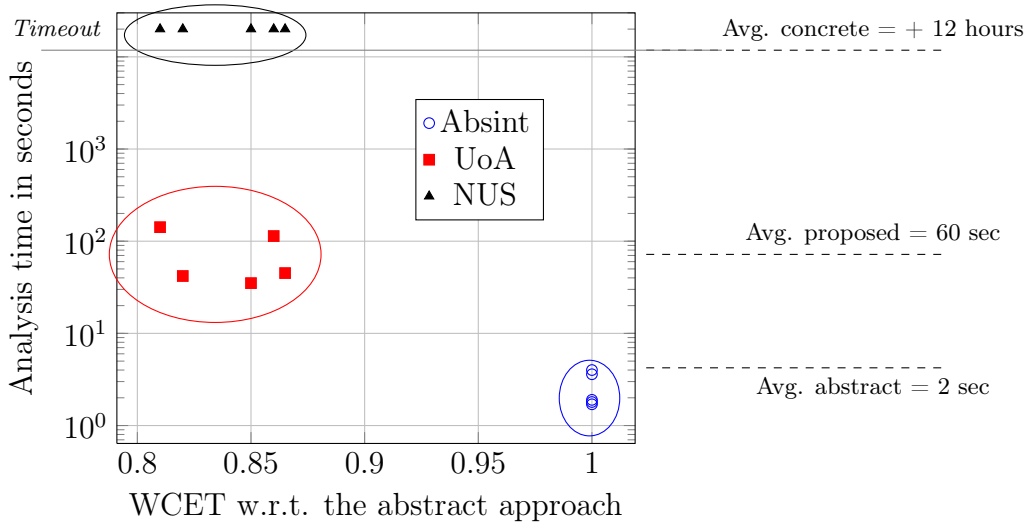


Figure 19: Comparing WCET and analysis time for the last five examples (1% relative cache size)

results reveal that the precision of the UoA approach is the same as the NUS approach, but more precise than the Absint approach. Also, the analysis time is much shorter than the NUS approach, but longer (on average by 1 minute) than the Absint approach.

References

[1] *MicroBlaze Processor Reference Guide*. `www.xilinx.com` (Last accessed: 01/10/2012).

[2] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static WCET analysis to Automotive Communication Software. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS'05, pages 249–258, Washington, DC, USA, 2005. IEEE Computer Society.

[3] H. Falk and J. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *Proceedings of the 46th ACM/IEEE Design Automation Conference (DAC)*, pages 732 –737, July 2009.

[4] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 35:163–189, November 1999.

[5] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[6] M. Kuo, L. H. Yoong, S. Andalam, and P. Roop. Determining the worst-case reaction time of IEC 61499 function blocks. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 1104 –1109, July 2010.

[7] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *Proceedings of the 40th annual Design Automation Conference (DAC)*, pages 466–471, Anaheim, CA, USA, 2003.

[8] Y.-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(3):257–279, July 1999.

[9] Y. Liang and T. Mitra. Static analysis for fast and accurate design space exploration of caches. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/-Software codesign and system synthesis (CODES+ISSS)*, pages 103–108, Atlanta, GA, USA, 2008.

[10] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable Programming on a Precision Timed Architecture. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*, Atlanta, GA, USA, October 2008.

[11] I. Liu. *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012.

[12] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate Estimation of Cache-Related Preemption Delay. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 201–206, Newport Beach, CA, USA, 2003.

[13] I. Puaut. WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, Dresden, Germany, July 2006.

[14] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, automation and test in Europe (DATE)*, pages 1484–1489, Nice, France, April 2007.

[15] J. Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008.

[16] M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004), volume 3292 of LNCS*, pages 371–382. Springer, 2004.

[17] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, ISOLA '06, pages 212–219, Washington, DC, USA, 2006. IEEE Computer Society.

[18] J. Souyris, E. L. Pavec, G. Himbert, V. Jégu, and G. Borios. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, Palma de Mallorca, Spain, July 2005.

[19] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad Allocation for Concurrent Embedded Software. *ACM Transactions on Programming Languages and Systems*, 32(4), 2010.

[20] H. Theiling. Extracting safe and precise control flow from binaries. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, RTCSA'00, pages 23–30, Washington, DC, USA, 2000. IEEE Computer Society.

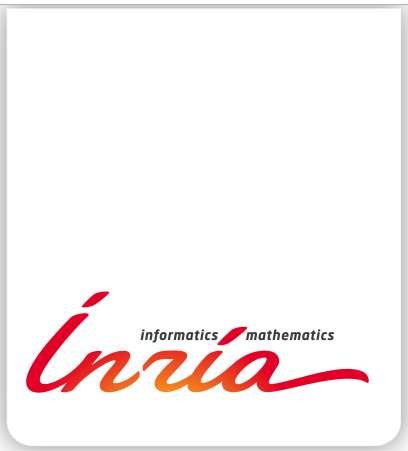
[21] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):966–978, July 2009.

[22] S. Wilhelm and B. Wachter. Symbolic state traversal for WCET analysis. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT'09, pages 137–146, Grenoble, France, 2009.

Contents

List of Figures

List of Tables



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399