Analysis and Verification of Cyber-Physical System Software Using Static Analysis

By

Zhenkai Zhang

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December, 2015

Nashville, Tennessee

Approved:

Professor Xenofon D. Koutsoukos

Professor Gabor Karsai

Professor Akos Ledeczi

Professor William H. Robinson

Professor Janos Sztipanovits

ACKNOWLEDGMENTS

I would like to sincerely thank my advisor, Dr. Koutsoukos. It is impossible to finish this work without his help, guidance, and encouragement. Besides, he acts as a strong role model of a good researcher that I can look up to.

I would also like to express my gratitude to the rest of the committee members – Dr. Karsai, Dr. Ledeczi, Dr. Robinson, and Dr. Sztipanovits. I really appreciate their generous time, comments and suggestions.

My thanks also go to all of my fellow labmates and collaborators – Joe Porter, Emeka Eyisi, Siyuan Dai, Heath LeBlanc, and so forth. Thank them for their contributions to my work, but most importantly, thank them for being such good friends and putting up with my silly jokes.

Last but not the least, I would like to thank my parents and my wife. I appreciate their everlasting support and endless love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

Since the modern notion of Cyber-Physical Systems (CPS) was formulated during the last decade, the CPS research area has attracted considerable attention from multiple research communities [1]. It has been well-recognized that every corner of human life is or will be reached by CPS, for example, power grids, automotive vehicles, aerospace crafts, robotics, and medical devices. CPS will have great economical and societal impact in the near future, and major investments are being made worldwide to develop the required technology [2]. However, in spite of those positive effects on our lives, there are also great technical challenges in developing efficient and dependable CPS.

CPS can be characterized as integration of computation, communication, and physical processes. The cyber part of the system typically monitors and/or controls the physical part, while the physical part affects the cyber part through feedback loops [2], [3]. Therefore, there are tight interactions between the physical dynamics, computational platforms, communication networks, and CPS software [4], [5], [6]. Different from traditional standalone embedded systems, CPS must interact with the physical world in a safe, dependable, secure, efficient, real-time, and sometimes distributed way, since most of the CPS are safety-critical control systems, such as automotive vehicles, aircraft, and industrial processes. Therefore, formal methods for systematic analysis can be valuable when analyzing/verifying system properties.

Due to the tight interactions in CPS, it is very difficult to have a unified method or framework to formally analyze/verify the whole system. Moreover, economic factors, such as persistent effort for low production costs and tight time-to-market, further complicate the process of CPS analysis and verification. Since software can be the most error-prone

part in a system, CPS can benefit greatly by analysis and verification methods focusing on CPS software.

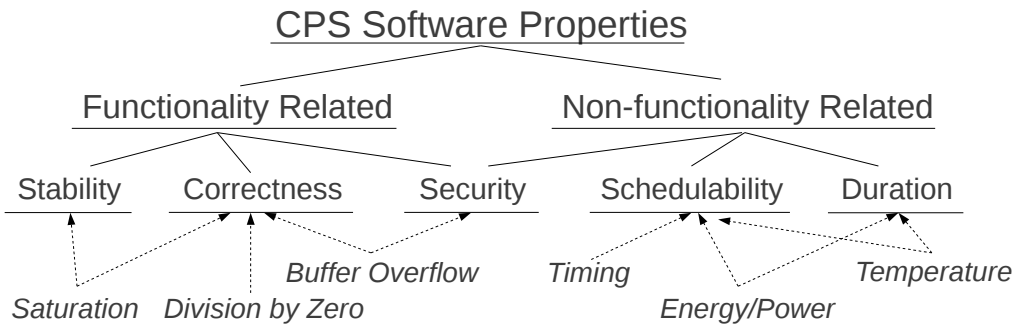


Figure 1.1: Taxonomy of CPS software properties

CPS software properties are diverse, and can be classified in different ways. A taxonomy of system properties related to our research is shown in Fig. 1.1. Properties can be categorized into one group that affect the computational behavior of the system and another group that impose constraints on the design of the system. Many of these properties are not decidable in general. For instance, if we could estimate the worst-case execution time (WCET) for any given program, then the halting problem (which is proven undecidable) would be solved. However, since we confine the concerned domain to CPS software, which uses more strict design methodologies, e.g. all the loops and recursions should be bounded, the properties are more decidable.

In this thesis, we focus on using **static analysis** to analyze and verify both functional and non-functional properties (in general, there are two main frameworks for static analysis [7]: data-flow analysis [8] and abstract interpretation [9]). Specifically, we perform **value analysis** to find what values the variables of interest may have in order to verify useful functional properties such as buffer overflow, division-by-zero, and saturation. We also perform **timing analysis** to analyze the timing behavior, for example, the WCET and preemption delay, which can be used for schedulability analysis.

1.2 Challenges

In this work, we argue that CPS software must be analyzed at a level of abstraction that captures low-level details of the computational platform. One direct challenge is that any software in its low-level form (e.g. machine-code) loses many apparent properties which are not dispensable to static analysis. For instance, the control-flow structure becomes less obvious in the low-level code. Therefore, there are challenges related to how to reconstruct the necessary information lost in transforming CPS software from its high-level form (e.g. source-code) to its low-level form, for example, control flow graph reconstruction.

- The main challenges in value analysis of CPS software are:
  - Selection of appropriate numeric domains which should be precise enough in order to derive useful value properties but allow operations with proper computational complexity in order to make the analysis scalable;
  - Accurate representation of the program semantics since value analysis needs to conform to the program semantics which are encoded in a machine-dependent way in the low-level code.
- The main challenges in timing analysis of CPS software are:
  - Incorporating in the analysis different micro-architectural components such as caches and cache hierarchies since these components have a huge impact on the variation of execution time;
  - Estimation of the preemption effect on a task since many task sets are only schedulable under preemptive scheduling strategies that lead to additional delays.

1.3 Contributions

In the following, we briefly state the contributions made in this thesis. The corresponding details can be found in later chapters.

1. We have studied how to perform generic value-set analysis on low-level code [10]. The main contributions of our work include:

   (a) We extend the strided-interval domain and also define operations on the domain to more precisely track the set of structured number.

   (b) We propose an intermediate language to capture the semantics of the instructions of different architectures.

   (c) We define the abstract semantics with respect to value-set analysis for the intermediate language.

2. We have studied how to safely and precisely derive timing information in the context of single-level and multi-level caches [11, 12, 13]. The main contributions of our work include:

   (a) We identify some sources of pessimism in two recent cache persistence analysis for WCET estimation and propose two methods to eliminate these sources of pessimism.

   (b) We propose a *top-down* and *bottom-up* approach that can more precisely analyze the behavior of a cache hierarchy maintaining the inclusion property.

   (c) We propose an approach which analyzes a multi-level inclusive cache as a whole based on the abstract interpretation of a concrete operational semantics defined for multi-level inclusive caches.

   (d) We investigate the cache-related preemption delay due to the "pollution" of the states of inclusive cache hierarchies and propose a method to bound this preemption delay.

1.4 Organization

The rest of this thesis is organized as follows: Chapter 2 describes the related work; Chapter 3 presents how to analyze value ranges of low-level code in a generic way; Chapter 4 gives the work on cache persistence analysis that improves state-of-the-art approaches; Chapter 5 describes the work on multi-level inclusive cache analysis that uses a novel idea to tighten the WCET; Chapter 6 describes the work on how to further improve the precision of multi-level inclusive cache analysis; Chapter 7 investigates the multi-level inclusive cache-related preemption delay analysis; and Chapter 8 concludes this thesis and lists future work.

Chapter 2

Related Work

The main concentrations of this thesis is related to how to analyze and verify properties of interest for CPS software in its low-level form. In order to represent the program for analysis, control-flow graph (CFG) needs to be extracted from the low-level code. Based on the CFG representation, various properties are analyzed/verified which are related to data value ranges, execution time variation due to caches, and execution time overhead due to preemptions.

Most of the CPS software properties can be precisely derived/verified if all concrete states of the system are available, which means it has to explore all possible executions under all possible environments. However, in general, the set of all concrete states is an infinite mathematical object which is not computable. In order to predict these properties, static analysis can be used to compute abstract states that over-approximate possible concrete states. By examining these abstract states, we can safely conclude the system has a property of interest if the abstract states satisfy it (although the opposite is not necessarily true: if the abstract states reject a property, the system may still have the property).

2.1 Control Flow Graph Reconstruction

Most static analyses are based on CFG of a program. The exact reconstruction of a CFG from a binary executable is very difficult due to (1) indirect control transfers (indirect jumps and indirect calls) typically generated due to uses of switch statements and function pointers; (2) exception handling where control is indirectly transferred to a handler which may belong to other binaries (e.g. RTOS); (3) ambiguous usage of instructions (e.g. MIPS's *jr* instruction can be used as *return* or *unconditional jump*); (4) data or padding bytes that may be contained in the text section.

In [14], Cifuentes *et al.* propose an approach to recover jump tables and their target addresses in a machine and compiler independent way. When an indirect jump is encountered in the process of disassembling of the binary, the approach first utilizes intraprocedural backward slicing technique which is described in [15] to acquire the smallest set of instructions that affect this indirect jump instruction. The slice is translated into a form called register transfer list which describes the effects of machine instructions in terms of register transfers. Then, forward expression substitution is performed to transform the indirect jump instruction to a normal form, which may belong to a set of normal forms for n-conditional jumps. If the derived form is one of the normal forms, the jump table base address and corresponding index can be derived.

Similar to the method in [14], Kastner *et al.* in [16] use program slicing to obtain the part of the program that affects the computed jump/call targets. They encode the operation semantics of a specific processor's instructions using an intermediate language called TDL (Target Description Language). By evaluation of the slice in the form of TDL, they can resolve the possible branch targets in a generic way. In [17], De Sutter *et al.* use a fictitious node called *hell* node to capture the conservativeness of unresolved indirect branches. The outgoing and incoming *hell* edges are refined by analyzing the sliced program.

In [18], Theiling argues that traditional top-down CFG reconstruction algorithms may suffer from the dependency of the information about procedure boundaries (start and end addresses), some data or padding mixed into text section, and interlocked procedure code parts. Thus, they propose a bottom-up algorithm to handle mixed data portions and interlocked code parts, and the approach does not need to know the boundaries of the procedures. The basic idea is to maintain a stack of procedure start addresses, and for each procedure, to maintain a queue of next possible instruction addresses. The program entry address is pushed into the stack of procedure start addresses. The reconstruction pops a procedure start address up as the current work address, and starts a loop: (1) disassemble the content of the current address into an instruction; (2) analyze the instruction to

discover next possible addresses according to the type of the instruction: enqueue falling through addresses or/and branch target addresses into the procedure-wise queue of possible instruction addresses for non-procedure-call instructions, and push call target addresses into the stack of procedure start addresses for procedure-call instructions; (3) exit the loop if the queue is empty, otherwise assign a dequeued address to the current work address and iterate again. The process ends when all discovered procedure start addresses are handled.

In [19], Kinder *et al.* propose an abstract interpretation-based framework for resolving indirect jumps which safely combines a pluggable abstract domain with the notion of partial control flow graphs. They propose a simple low level assembly-like language and assign this language a concrete semantics. They construct a *resolve* operator, based on conditions imposed on the provided abstract domain, for calculating branch targets. They also present their disassembly tool Jakstab which produces the most precise over-approximation of the control flow graph with respect to the used abstract domain. Following the approach proposed in [19], Flexeder *et al.* extend it with a treatment of procedure calls in [20]. One particular problem this work addresses is *abort* and *exit* functions, which do not return to the corresponding call site, but terminate the whole program whenever they are called.

Since approaches based on abstract interpretation produce over-approximation of the indirect branch targets, many spurious control flow edges and basic blocks are added into the reconstructed CFG, which makes the subsequent analysis or verification imprecise or even useless. An interesting idea is presented by Kinder *et al.* in [21] to improve the precision of CFG reconstruction. This method alternates over-approximation and under-approximation when resolving indirect branches: as soon as a predefined condition is met (the current indirect branch is unresolvable), it switches to try deriving an under-approximation of the branch targets and skips over this unresolvable indirect branch by substituting branch targets detected by the under-approximation. The under-approximation can be derived by simply executing the program with random input. Later, the analysis returns to use over-approximation.

In [22], Reinbacher *et al.* propose a SAT-based control flow graph reconstruction approach. This approach uses the bit-blasting technique to encode the semantics of each instruction i into a boolean formula $encode(i)$ (each register/memory location is encoded as a bit-vector and the semantics of i is encoded as the bit-wise changes), so the semantics of a basic block b is represented by $\phi(b) = \bigwedge_{i \in b}(encode(i))$. By using the value-set abstraction proposed in [23], the set of values that a bit vector can have during the execution are derived. For each basic block, the incremental SAT solving is applied to derive the pre-conditions on the bit vectors of the entry of the basic block and the post-conditions of the exit of the basic block. Given the boolean formulas which encode the semantics of the basic blocks and the input/output conditions, the program can be analyzed forward and backward to derive the invariants and resolve the indirect branches.

2.2 Value Analysis

Since many interesting CPS software properties are related to numerical computations at run-time, such as division-by-zero, variable saturation, and buffer overflow, a sound and precise static value analysis is essential for verifying these properties. Moreover, when reconstructing the CFG, the indirect branch target addresses also need to be resolved by value analysis. In order to make the value analysis sound and scalable, abstract interpretation of the concrete computational semantics defined on an abstract value domain becomes necessary.

Many numerical abstract domains are proposed and corresponding operations on these domains are defined. These domains are defined in the trade-offs between expressiveness and computational complexity. Intervals are the most straightforward numerical abstractions. For example, in [24], Cousot *et al.* use integer intervals to perform numerical analysis. In [25], Hickey *et al.* define the arithmetic operations on floating-point intervals. The computational complexity of operations on interval abstractions is acceptable, but the over-approximation of the operations can be very large. Some more precise domains are also

proposed, for instance, the octagon domain is proposed in [26] and the pentagon domain is proposed in [27]. The Parma Polyhedra Library is an excellent library implementation for using many convex polyhedral domains [28].

In [29], Balakrishnan *et al.* propose value-set analysis (VSA) which combines numerical analysis and pointer analysis together. The goal of VSA is to determine an over-approximation of the set of numerical values and addresses at each program point. It is a flow-sensitive, context-sensitive, and interprocedural static analysis approach based on abstract interpretation. For a set of values, VSA uses an abstract domain called strided-interval (*SI*) to over-approximate this set. A strided-interval is an abstract object used to represent a set of structured integers with a fixed stride. A k-bit strided-interval $s[l,u]$ where $-2^{k-1} \leq l \leq u \leq 2^{k-1}-1, 0 \leq s \leq 2^k-1$ represents the set $\{i | i = l + n \times s \wedge n \geq 0 \wedge i \leq u\}$. A set of operations including arithmetic, shift, and bit-wise operations on *SI* is given in [30]. VSA is used in a tool called CodeSurfer/x86 for analyzing Intel x86 binary code [31].

One problem of the original *SI* representation is that: if l is required to be less than or equal to u (i.e. $\forall s[l,u], l \leq u$), it loses the ability to precisely track the set of numbers when some of the numbers in the set lead to an overflow with respect to two's complement representation. In [32], Sen *et al.* introduce a numerical abstract domain called *CLP* (circular linear progression). *CLP* does not require $l \leq u$, so it can represent a set of structured integers which cross the signed computation overflow boundary. The corresponding operations are also defined in [32].

In order to realize generic value analysis on low-level code of different architectures, semantically translating the low-level code to a generic intermediate form becomes necessary. Intermediate languages are actually often used in many compiler frameworks, like GCC and LLVM, to enable the support of different programming languages at the front end and of different platforms at the back end. In order to make it possible to develop analysis tools and algorithms for generic value analysis on binary code, an intermediate language called REIL is proposed in [33]. REIL has a very compact set of intermediate

instructions while being able to achieve semantic closure, but the translated code in REIL may have a very long representation. More recently in [34], an extension to REIL called RREIL is made mainly with an addition of several comparison instructions to take into account relational information. Based on RREIL, a toolkit called GDSL to specify semantics to machine languages is presented in [35]. In [36], a binary analysis platform (BAP) is introduced which is a successor of Vine that is used in the BitBlaze project [37]. Both BAP and Vine use a formally specified intermediate language to enable generic static analysis. In [38], a binary code analysis framework called BINCOA is proposed that is based on a formal automaton model to capture the low-level code semantics. In order to achieve adaptable value analysis, intermediate representations are also used in [39]. Incremental SAT solving is used to derive sets of values for registers taking into account the relationship between registers and status flags.

2.3 Timing Analysis

Since most of the CPS are hard real-time systems which have stringent timing constraints, a single deadline violation may cause significant damages. Thus, when designing a CPS, a schedulability analysis must be carried out to guarantee all its timing constraints will be met. However, only when each task's WCET is known can schedulability be analyzed. Therefore, timing analysis is an essential job in the CPS design process.

WCET estimation problem has been under research for more than twenty years, and many methods have been used in CPS, e.g. AbsInt's aiT WCET analyzer has been used to analyze A380's flight control software [40]. It is worth to mention that Whilhelm *et al.* give an excellent survey on WCET estimation techniques in [41]. Different timing analysis methods are reviewed and different tools are compared from a bird's-eye-perspective.

Since the exact WCET is almost impossible to acquire, we have to perform analysis to derive a sound estimation, as shown in Fig. 2.1. A valid and useful WCET estimation has to satisfy safety and precision criteria: (1) the estimation must be safe, i.e. it is an

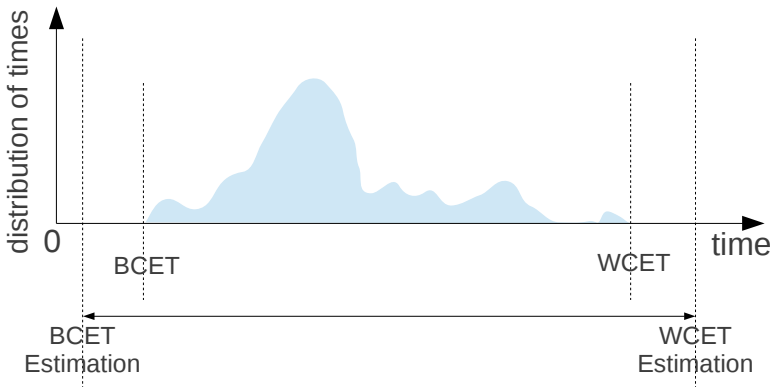


Figure 2.1: WCET/BCET and WCET/BCET estimations

upper bound of the real WCET; (2) the estimation should be as tight as possible in order to maximize the processor utilization.

There are three timing analysis techniques: static approach, measurement-based approach, and hybrid approach [42]. While the measurement-based approach cannot guarantee the estimation is safe, the static approach considers all possible executions and can provide the bounds on the execution time. The hybrid approach combines the measurement-based method and static method together, but it still suffers from the deficiencies of the measurement-based approach, and instrumentation code may be needed which, however, is not allowed in some systems.

In general, the static approach starts with CFG reconstruction from the statically linked binary executables (since only can statically linked binary executables provide all the needed information and are not affected by compilation). Each node of the CFG is a basic block which is a sequence of instructions without branches, except possibly at the end, and without branch targets, except possibly at the beginning. Taking into account the processor's micro-architecture (e.g. pipeline and cache organizations), each basic block's WCET can be estimated. Techniques such as integer linear programming or model checking can be used to find the path that leads to the worst case execution time of the program.

If the processing unit of the system is just a simple microcontroller (single-core without performance enhancement features, such as caches, pipelines, and branch predictors),

the WCET analysis becomes much easier. However, as more and more computation is required, a lot of features are added into the processor to improve the performance but at the same time these features make the timing behavior very hard to predict. Therefore, WCET estimation approaches greatly depend on the processor's micro-architecture [43].

Our work mainly focuses on cache analysis for WCET estimation[1]. The performance gap between processors and main memories is increasingly large. As a result, caches are introduced into the memory system to reduce this gap by leveraging the *principle of locality* (spatial and temporal localities). Nowadays, many embedded processors even contain multi-level caches. The presence of caches improves the average-case performance, but their dynamic behavior makes the execution time change considerably so that the worse-case performance is very hard to predict. Thus, a sound cache analysis becomes very important in WCET estimation.

2.3.1 Cache Analysis for A Single Processor

In terms of a uni-core processor, the state-of-the-art single-level cache behavior analysis using abstract interpretation is proposed in [53]. It is assumed that the cache uses least recently used (LRU) replacement policy[2] and can have different associativities. The method aims to classify each cache access into four categories under all possible execution scenario, i.e. cache hit/miss classification (CHMC), as described in Tab. 2.1. Three separate static analyses are used to derive the CHMC for every instruction cache access. At each program point, each analysis computes an "abstract cache state" (ACS) according to this

[1] In addition to caches, other performance enhancement micro-architectural features of processors also have an important role in affecting the execution time of a task. Two common ones in some advanced embedded processors are pipelines and branch predictors. Pipelines are analyzed in terms of in-order (e.g. in [44], [45], [46], and [47]) or out-of-order (e.g. in [48]). Branch predictors are mainly analyzed to bound the misprediction effects e.g. in [49], [50], and [51]. An interesting phenomenon called timing anomaly can arise from interactions between caches, pipelines, and branch predictors in a dynamically scheduled processor [52].

[2] Other replacement policies are also studied by many researchers, for instance, PLRU is studied in [54] and [55]; FIFO is studied in [56], [57], and [58]; and MRU is studied in [59]. However, as stated in [60], LRU replacement policy is the most predictable one.

analysis's semantics. The semantics is actually embodied into two functions - *update* and *join*. The *update* function is used to change the ACS before a program point into the ACS after the point; The *join* function is used to combine two or more ACS into one ACS when two or more control flows converge (e.g. the convergence point of a conditional branch).

must analysis determines which blocks are definitely in the cache at the cache access time: if the accessed block is in this analysis's ACS, the cache access is a *AH*.

may analysis determines which blocks may be in the cache at the cache access time: if the accessed block is not in its ACS, the cache access is a *AM*.

persistence analysis determines if a block will not be evicted after it has been loaded: if the accessed block is in its ACS, the cache access is a *PS*. If a cache access can not be classified as neither *AH*, *AM*, nor *PS*, it will be classified as *NC*.

Classification	Semantics
AH (always hit)	the access always causes a cache hit
AM (always miss)	the access always causes a cache miss
PS (persistent)	the access may be a miss for the 1st time, but always hits later on
NC (not classified)	the access cannot be classified into neither AH, AM, nor PS

Table 2.1: Cache Hit/Miss Classification (CHMC)

The techniques stated above can only safely provide single-level instruction cache analysis but not for multi-level instruction caches. The first correct multi-level non-inclusive instruction cache behavior analysis is proposed in [61], and later extended in [62] for data caches and in [63] for shared instruction caches. The idea behind the method is to classify if a memory reference will access each cache level. Three cache access classifications (CAC) are introduced for categorizing a memory reference r at a cache level l, as described in Tab. 2.2. The CAC for r at l depends on the results of cache analysis of this reference at the level l-1 (CHMC and CAC). Obviously, if r at a level before l is *AH*, then the CAC for r at l is *N*; r at the first level is definitely *A*, and if r at level l-1 has the CHMC *AM* and CAC *A*, r at level l is definitely *A* as well; otherwise, the CAC for r is *U*. In order

to take into account the extensions, the *update* function in [53] also needs to be extended depending on the CAC for r at cache level l: if CAC is A, the original *update* function in [53] is used; if CAC is N, the *update* function is an identity function which makes the ACS of level l unchanged; if CAC is U, two sub-cases are considered – one assumes the access is performed, so the *update* function is the original one, and the other one assumes the access is not performed, so the *update* function is an identity function, and after having these two sub-ACSs, the original *join* function is applied to them to get the final updated ACS. Thus, we have

$$update(r,l,ACS) = \begin{cases} update_{original}(r,ACS) & if\ CAC_{r,l} = A \\ ACS & if\ CAC_{r,l} = N \\ join_{original}(ACS, update_{original}(r,ACS)) & if\ CAC_{r,l} = U \end{cases}$$

Classification	Semantics
N (never)	the access to r is never performed at cache level l
A (always)	the access to r is always performed at cache level l
U (uncertain)	the access to r may be performed at cache level l

Table 2.2: Cache Access Classification (CAC)

In [64], Hardy *et al.* extend/modify non-inclusive cache hierarchy analysis method to take into account inclusive and exclusive cache hierarchies. For inclusive cache hierarchies, some accesses that are always hits under the situation of non-inclusive multi-level caches may not be always hits anymore, since in order to maintain inclusion property, when a cache block is evicted on a lower level, its corresponding copies on the higher levels need to be invalidated. Thus, the method classifies all memory accesses to all cache levels, except the first level, as U (uncertain), which makes the analysis of each cache level independent from other levels. Similar to the approach in [63], U access classification makes the analysis take into account two possible ACSs due to a memory reference and join the two into one ACS. This mechanism introduces great pessimism. Also, the CHMC of each reference is changed according to the influence of possibly invalidated blocks, which can be derived

from a persistence analysis. If an *AH* or *PS* reference on a cache level without considering invalidation has its referenced block in any of the sets of possibly invalidated blocks of higher cache levels, it needs to change to *NC* to take into account the possible influence of invalidation. For exclusive cache hierarchies, an ingenious modeling of this hierarchy is proposed to ignore whether a reference accesses to each cache level but to only consider the impact of exclusion enforcement on the contents of multi-level caches. Under this modeling technique, the ACSs of an exclusive cache hierarchy can be transformed into an ACS of a single level cache, which then can be calculated by using the approach in [53].

Compared to data caches, the behavior of instruction caches is relatively easy to predict since in a basic block (BB) the sequence of instruction memory access addresses is known but the sequence of data memory access addresses may be varying, i.e. depend on its run-time behavior. Local variables and procedure formal parameters are allocated on the stack, and in the binary code they are often referenced relative to the stack pointer. Thus, if a data memory access address is given by the pattern *stack pointer* + *offset*, the address can be easily calculated since the values of the stack pointer can be determined considering the calling contexts. In [65], Hur *et al.* extend the timing schema approach proposed in [45] to take into account many micro-architectural features including data caches. They conservatively assume if the load/store address is not given following the *base register (e.g. stack pointer)* + *offset* pattern (so-called dynamic load/store instruction), it will cause two cache misses (one is for the memory reference and the other is for the references to the replaced block). In [66], Kim *et al.* first use reaching definitions data-flow analysis to eliminate as many dynamic load/store instructions as possible, and then they employ the *pigeonhole principle* to reduce the possible cache misses of dynamic load/store instructions – if there are n memory references to m memory locations and $n > m$ there must be at least $n - m$ cache hits. However, they must guarantee these n memory references should not cause cache replacements (i.e. no conflict misses for accessing direct mapped cache).

There is no doubt that if we want to improve the precision of data cache analysis in the presence of dynamic load/store instructions, the set of possibly referenced addresses need to be resolved. Usually, array references within loops are the main source of dynamic load/store instructions, and they often exhibit certain reference patterns. In [67], Ghosh *et al.* propose using a set of linear Diophantine equations called cache miss equations (CME) to capture the relationships among loop indices, array sizes, base addresses, and cache parameters for cache misses in a loop nest. Reuse vectors proposed in [68] are used to capture the temporal and spatial locality information. For each data reference instruction and along every reuse vector for that reference, two kinds of CME are generated, which are cold miss equations and replacement miss equations respectively. By solving the equations, accurate cache misses can be derived. However, the method requires perfect loop nests with no data dependent conditionals (perfect loop nests have all array references contained within the inner-most loops). Furthermore, it also requires array indices and loop bounds to be affine combinations of loop induction variables. In [69], Chatterjee *et al.* propose an approach based on Presburger arithmetic formulas to precisely model data cache behavior. The approach has advantages because it can handle imperfect loop nests and a variety array layouts from row- and column-major to non-linear. However, the computational complexity of the approach is super-exponential in the worst case for satisfiability checking and quantifier elimination of Presburger formulas.

In [70], Ramaprasad *et al.* extend the CME method to derive exact cache hit/miss patterns. This work proposes a technique called forced loop fusion to concatenate iteration spaces of sequential loops and introduce conditionals based on loop induction variables to maintain correctness. Using this technique, it can deal with imperfect loop nests and non-rectangle loops (non-rectangle loops has a condition on the upper bound of an inner loop that is based on current value of an outer loop). The method analyzes all iteration points (an iteration point is a vector whose components give corresponding loop iteration) in order to get an exact data cache reference pattern. After solving the generated CME, the method

re-analyzes all the compulsory misses to eliminate pessimistic misses and also verify cache hits against the conditionals introduced at forced loop fusion. Apparently, methods like the one in [70] depend on affine reference patterns which can be too restrictive.

In [71], White *et al.* use address expansion to calculate relative addresses for each load-/store instruction. This relative address calculation actually can be achieved by a compiler automatically. The calculated relative addresses are relative to the beginning of a global variable or the stack frame. In order to get the set of virtual addresses a data reference may access, control flow information is combined with relative addresses to derive the ranges of virtual addresses. A static cache simulation approach which basically is an interprocedural data-flow analysis is used to categorize each scalar data reference into four categories similar to the categories described above for instruction references in Tab. 2.1. For a non-scalar data reference in a loop nest, the method introduces a new category called *calculated* indicating the maximum number of data cache misses at each loop nest level. When computing the maximum number of misses, the method also takes into account the spatial and temporal localities to increase the accuracy.

In [72], Ferdinand *et al.* propose using a ***persistence analysis*** to tighten the number of possible cache misses due to instructions referencing data caches. For a set of memory blocks $M = \{m_1, m_2, ..., m_x\}$, if each of memory block $m \in M$ can be guaranteed as persistent, and then the dynamic load/store instruction that possibly accesses these M blocks can cause at most $|M|$ data cache misses. Thus, a persistence analysis is used to bound the worst-case data cache conflicts. However, this persistence analysis method is not safe, as pointed out in [73] and [74].

In [75], Sen *et al.* first perform an address analysis to derive all the possible addresses an instruction may reference, and then perform a ***must analysis*** of the cache contents. The must analysis is extended from the well-established must cache analysis described in many papers (e.g. in [53]). Their cache model is also n-way set associative employing LRU replacement policy with write-through and write-no-allocate. The extension is to

safely handle a set A of possibly accessed addresses when updating the ACS, while the original method only deals with one address like when fetching an instruction. Since this is a must analysis, if a memory block of the set A is not in current ACS, it should not be added into the ACS; that is because "possibly accessed" is not "definitely accessed" and a must cache analysis only discovers the memory blocks that are definitely in the cache at a given program point. Given the set A by an address analysis, for each relative age l_i of each ASS $\hat{s}_x$ of the ACS, the method computes the number of ages $o(x,i)$ that the content of l_i should become "older" with. Let $\alpha(an\ address)$ denote the memory block this address references. The $o(x,i)$ will be an upper bound which gives the worst effect on aging each cache block by referencing the set A of possible addresses: $o(x,i) = |\{a \in A | a \text{ mapped to set } x \wedge (\alpha(a) \in \text{content of } l_j, i < j \leq n \vee \alpha(a) \notin \text{content of } l_k, 1 \leq k \leq n)\}|$. They also employ partial loop unrolling to sequentialize the set of addresses as much as possible such that an instruction that may reference many addresses in a loop will have several duplicates each of which references fewer addresses in the unrolled loop instances.

In [73], Huynh *et al.* observe that the abstract interpretation-based methods like those used in [72] and [75] suffer from large over-estimation because they ignore the temporal information of a data memory access when using address analysis to compute the set of possibly referenced memory blocks. As mentioned before, the paper points out the persistence analysis in [72] can underestimate the relative ages when updating the ACS. They fix the problem by introducing a younger set $ys(m)$ to each memory block m at each program point that keeps track of memory blocks that may be younger than m when reaching the program point along some possible execution path. Thus, for any memory block m under the fixed persistence analysis, its maximal relative age will be $|ys(m)| + 1$. Based on the persistence analysis, they propose a scope-aware data cache analysis method which captures the temporal information of a data reference over different loop iterations. A temporal scope for memory block m which is possibly accessed by a data reference d is a mapping: $ts(m)(d) = \{loop_i^d \mapsto [lw, up]\}$ where $loop_i^d$ is the i^{th}

loop d resides in and $[lw, up]$ is the corresponding loop's iteration range in which d may access m (derived through access patterns). Two temporal scopes of memory blocks m_i and m_j overlap at a loop nesting level if and only if the iteration intervals of this loop nesting level and all upper loop nesting levels have intersections. Of course, when a memory block m may be referenced by an instruction, for all memory blocks whose temporal scope overlaps with the temporal scope of m, their younger set needs to include m. The proposed method categorizes the persistence of memory blocks in the calculated temporal scopes. If a memory block m is not evicted during the iteration interval $ts(m)(d)(loop_i)$, then all accesses to m from d cause at most one cache miss within the corresponding loop execution. Given different loop nesting levels, the same memory block may have different classifications. Let $L_{ps}(m)$ be the outer-most loop nesting level where a memory block m is consistently persistent. The upper bound of cache misses caused by a data reference d will be $\sum(\prod(ts(m)(d)(loop_i).up - ts(m)(d)(loop_i).lw + 1))$ where $m \in$ d's possibly accessed blocks $\wedge$ $loop_i$ is outer than $L_{ps}(m)$.

Interestingly enough, the underestimation generated by the persistence analysis in [72] is also pointed out by Cullmann in [76]. A different approach is proposed to tackle this problem which is to take into account the information computed by the may analysis. In this proposed persistence analysis approach the abstract domain is augmented with a parallel may analysis ACS on the basis of the original ACS of persistence analysis. The new *join* function is just the parallel combination of the *join* function of may analysis and the *join* function of the original persistence analysis. The new *update* function has two cases to consider: (1) if the sum of the distinct memory blocks (compared to the currently accessed one) is greater than or equal to the set associativity, which means that the set is possibly full, then the *update* function updates the ACS of persistence analysis by shifting all positions to the right by one and making the youngest age contain the currently accessed one; (2) otherwise, the set is not full yet, so *update* function does the shifting as well but will not evict blocks. In [76], Cullmann also introduces a much simpler algorithm by using conflict

counting to compute persistence. It just keeps track of possibly contained memory blocks and uses the cardinality of this set to decide if a reference is persistent or not. Although it is safe (overestimation is guaranteed), it is not precise as the other approaches.

Multi-level non-inclusive instruction cache behavior analysis is proposed in [61], but this method cannot be directly applied to multi-level data caches. In [62], Lesage *et al.* extend the method in order to analyze multi-level non-inclusive set-associative data caches. As the method in [61], the approach also relies on the concept of *cache access classification* (CAC) at each cache level for a data reference. Address analysis is used to get a safe approximation of accessed addresses for each data reference. CHMC similar to the ones in Tab. 2.1 is used except here a data reference may access more than one addresses such that if a data reference is *AH* then all the possibly referenced blocks should be in the ACS (in the case of must analysis). A *content-independent* (*CI*) classification is also introduced for store instructions – since their cache model assumes (1) write-through and write-no-allocate policies are used; (2) writes go all the way through the memory hierarchy (3) write will not affect the corresponding relative age of cache blocks, a store instruction will not depend on the cache contents (only load instructions depend on and update ACS). When a load instruction may access multiple addresses, the ACS is duplicated as many times as the possible addresses and each of the duplicates is updated by the original *update* function for a possible address, and then the original *join* function is used to merge these duplicated ACSs together as the updated ACS. CAC similar to the ones in Tab. 2.2 are used except a new classification *U-N* is added which means no guarantee can be given for the first access to each possibly referenced blocks of the memory reference, but next accesses will never be performed at this cache level. The rest of multi-level data cache analysis is similar to the method in [61] with respect to the corresponding modified *update* functions of the analysis (must/may/persistence).

In [77], Chattopadhyay *et al.* consider analysis of a multi-level cache architecture with separate L1 instruction and data caches and a unified L2 cache which contains both in-

struction and data. The approach combines the techniques presented in [53], in [75], and in [61] for L1 instruction cache analysis, L1 data cache analysis, and CAC computation respectively, and extend some techniques for unified L2 cache analysis. The basic steps to compute the ACS of unified cache when executing a given instruction are: (1) update the ACS first with the memory block where the given instruction is stored by using the approach described in [61]; (2) if the given instruction is not a load/store instruction, return the updated ACS; (3) compute the set of possibly accessed addresses through address analysis; (4) update the ACS considering all the possible memory blocks of the data access also by using the approach described in [61]. For different analyses (must/may/persistence), the corresponding *update* and *join* functions are different and they are given in [72] for persistence analysis, in [75] for must analysis, and they extend the may analysis in [53] to take into account non-singletons.

The approaches above for data cache analysis all assume **write-through** and **write-no-allocate** schemes in favor of ease of analysis. However, **write-back** policy with **write-allocate** is more common in processors. In [78], Sondag *et al.* propose a novel abstract domain called "live caches" to handle multi-level instruction and data cache behavior analyses, and their approach deals with write-back and write-allocate policy. Since write-back with write-allocate is used, when a write happens, the corresponding block will be allocated on the lower levels (starting from the first level) if it is not there until it is found on some level, and the corresponding block on the first level also becomes dirty. A live cache is an abstract cache maintained for a pair of cache levels. If there are N cache levels, there will be $\binom{N}{2}$ live caches. A live cache is also set-associative, and each live cache set relates two corresponding sets of the two related cache levels with the larger associativity of the sets. A block will be in the live cache if it exists in at least one of the two related levels, so live caches preserve the information that accessing the blocks will hit either level of cache. Thus, when any of the related sets is updated, the corresponding live cache set needs to be updated as well. The position of a block is determined by taking the better case of the

same block's positions in the two related cache levels. Both *join* and *update* functions need to be modified to take into account the live cache and write-back policy. These modified functions are rather complicated and have many sub-cases to consider. It should be noted that when using *join* function, for must analysis, if the same block are both dirty in the ACSs, the block is dirty in the joined ACS, but if a block is dirty in only one ACS, mark it as maybe dirty; for may analysis, if a block is dirty in any one of the ACSs, it is dirty in the joined ACS. Keeping track of dirty/maybe dirty blocks makes the analysis safe in the case of using write-back policy.

2.3.2 Cache Analysis for A Multi-Core Processor

As many CPS require more and more computation, e.g. automotive control systems may need to process and analyze images/videos, multi-core or many-core processors emerge in CPS applications quickly. Although these technologies make high-performance and low energy consumption CPS possible, the interferences in shared resources between cores affect their timing behavior predictability [79].

A significant type of interference occurs in shared caches/memories. When several tasks/threads run on different cores, the states of shared caches can be perturbed in a non-deterministic manner since the access times of these tasks/threads are unpredictable. In order to address this problem, research focuses on deriving an upper bound of disturbance in shared instruction caches (shared data caches are much harder to analysis due to the presence of cache coherence).

In [80], Yan *et al.* make the first attempt to analyze the WCET of a task running on a multi-core processor. In the paper, only the memory accesses due to instruction references are taken into account, so it is assumed there are no cache misses due to data references (perfect L1 data cache). In the system model, each core has a private L1 cache and all the cores share the same L2 cache which is a direct-mapped cache, and tasks running on different cores are independent and non-preemptive. Since a L2 instruction cache access only

occurs when an instruction is not available in the L1 instruction cache, the method mainly tries to estimate the bound of interferences in the shared L2 instruction cache. It is observed that the worst-case instruction cache access interferences can be computed by examining the control flow information of tasks of different cores (distinguishing instructions that are in loops from instructions that are not in loops). The method classifies the L2 accesses into three categories (L2 hit, L2 miss, and L2 hit except once). Each L2 access is categorized considering the worst-case according to (1) whether or not the instruction leading to the L1 miss (so causing L2 access) is in a loop; (2) whether or not the corresponding instruction is in L2 cache; and (3) whether or not the direct-mapped location may be contended by tasks. However, the approach makes many strong assumptions, e.g. perfect L1 data cache, and still gives relatively pessimistic estimation [81] [82] [63]. Moreover, Hardy *et al.* argue that this method might underestimate the WCET when different cache blocks of the interfering task are not in loops but map to the same location, or when the interfering task executes several times while the real-time task is in execution [63].

In [81], Zhang *et al.* extend the Implicit Path Enumeration Technique (IPET) (proposed by Li *et al.* [83] [84]) to take into account the shared L2 cache. The system model is exactly the same as the model in [80]. The method uses integer linear programming to establish an optimization model of the execution time of a task. The objective function has two summand parts to represent the execution time of a task: the first part is the sum of all execution time costs under L1 cache hit situations; the second part is the sum of all costs under L1 cache miss situations. Since a L1 cache miss needs to access L2 cache which can lead to either a L2 cache hit or a L2 cache miss and a miss is caused either by the task itself or by interferences of other tasks running on other cores, the second part takes into account these scenarios. The objective function is subject to three kinds of constraints: (1) structural constraints, which describe the number of times flowing into a basic block equals to the number of times flowing out of the block; (2) functionality constraints, which capture conditions on feasible execution paths that depend on functionality of the task like

loop bounds; (3) micro-architectural constraints, which give the ranges to some variables based on micro-architectural features like direct-mapped cache conflict misses. Thus, the WCET of the task is to maximize the objective function.

In [82], Li *et al.* propose an iterative approach to estimate the worst-case response time (WCRT) of a concurrent program running on a multi-core processor. An observation is that there will not be interferences between tasks if their lifetimes are not overlapped, even though the same memory locations are accessed by them. Therefore, the method described in [80] is very conservative, since it does not consider the lifetime overlapping information. Although this method also only considers instruction references (i.e. perfect L1 data cache in their system model), the cache model can have different associativities, namely direct-mapped, n-way set associative, or fully associative. It is assumed that the tasks are non-preemptive but can have dependencies (two dependent tasks will of course not overlap). Thus, the task's partial order and dependency can serve as the initial overlapping information. Also, it is assumed that the architecture is free from timing anomalies, so a cache miss will always contribute more execution time than a cache hit. By using techniques described in [85] and [61], the method first performs multi-level non-inclusive cache analysis for each task mapped to each core independently. Since the initial overlapping information is very limited, the first iteration assumes all the tasks that may have overlapping lifetime are overlapped. Then, a shared L2 cache conflict analysis is carried out in order to check whether some L2 cache hits should be turned into "not classified" due to interferences. Consequently, the method can calculate conservative earliest ready time and latest finishing time for each task. According to the calculated lifetime results, a task's overlapping information may be changed because some of its interfering tasks may have a disjoint lifetime with the task. When any of the overlapping information is altered, another iteration will be required to continue refining the information. The iteration will terminate when it finds the least fix-point of the overlapping information. When the overlapping information is fixed, the WCRT of the concurrent program can be calculated.

In [63], Hardy *et al.* propose an interesting idea to tighten the WCET estimation by using bypass features of some instruction set architectures. Traditionally a cache miss would retrieve the missing block from a lower memory hierarchy level and store it into all upper levels. An observation is that some blocks may not be accessed again before evicted (single-usage blocks). Thus, the method tries to only cache blocks that are statically known as reused, but bypass other blocks without letting them pollute the shared caches. First, the method extends the multi-level non-inclusive cache analysis technique proposed in [61] to take into account shared caches. Based on their previous cache access classifications (CAC), a new category ("U-N") is added to produce a more precise identification of static single-usage blocks. On each shared cache level, for every cache set s, the method computes the worst-case number of potential cache conflicts denoted by $CCN(s)$: if any reference of a task from other cores is not "N" and the block that the reference accesses to is mapped to the cache set s, it will contribute 1 to the $CCN(s)$. Since the $CCN(s)$ serves as the worst-case interferences, a safe cache hit/miss classification (CHMC) can be derived considering these interferences. Second, on each shared cache level, static single-usage blocks can be identified by combining each memory reference's CHMC and CAC. By setting some bits in the instructions, the identified blocks can be bypassed so as to tighten the WCET.

In [86], Lv *et al.* propose to use both abstract interpretation and model checking techniques to solve the WCET estimation. The paper models the multi-core processors more realistically to take into account the shared buses. Different bus arbitration protocols are considered like TDMA and FCFS. First, the method uses abstract interpretation (see [85]) to analyze the private L1 cache behavior of a task running on a core. Then, it combines the CFG of the task with the cache hit/miss classifications to build timed automata (TA) of the execution behavior. For each "alway hit" instruction, the TA is quite simple – only considering L1 cache access time and instruction execution time. For each "alway miss" instruction, its execution will definitely access the shared bus, which can be modeled by

using channels to synchronize with the bus TA model. For each "first miss" instruction, a flag is used to record whether the instruction is the first time being referenced in order to select one of the two paths; if it is the first time, it needs to access the shared bus and the path is modeled as "always miss" TA; otherwise, all other subsequent references are cache hits and the path is modeled as "always hit" TA. For each "not classified" instruction, two non-deterministic paths are used to take into account every possible execution scenario: one path is for considering L1 cache miss and the other one is for considering L1 cache hit. The TA of the shared bus models its arbitration protocol, e.g. in the case of TDMA, fixed bus slots are allocated to cores, and in the case of FCFS, a queue is used to keep the access order. Channels are used for sending bus access requests and notifying bus access completion. The network of TA models can be checked using UPPAAL to find the WCET. However, it is assumed that there is no shared cache and so every data can be found on the second level of memory hierarchy (which is the main memory). Thus, if shared caches need to be added, the scalability of this approach is unknown.

Compared to data references, instruction references are easier to analyze, since all the instruction addresses are known from the binary. Although assuming L1 data cache is perfect is not realistic at all, only a few papers consider both data and instruction caches.

In [87], Gustavsson *et al.* propose to use timed automata to model the effects of both instruction and data caches. This approach also utilizes CFG information to build TA for the program which interact with the TA of the cores via channels. Each core has three TA: a timing model, a private instruction cache model, and a private data cache model. Their L1 data cache model is similar to the L1 instruction cache model except that the data cache model has the ability to invalidate a data cache block in other cores (i.e. modeling the coherence). All the cores' TA interact with a shared cache model which has to be mutual exclusively accessed through a spinlock synchronization model. One problem of this approach is it does not scale well, since it does not perform any value analysis and control flow analysis but depends heavily on the exhaustive search ability of the model

checker.

In [88], Lesage *et al.* combine and extend their previous work in [61], [62], and [63] to deal with shared data caches. The system model assumes *write-through* policy with *by-passing* shared cache blocks from the private caches of cores. Therefore, it is not necessary to deal with additional coherence traffic induced by cache coherence. First, the method uses address analysis to derive the set of possible addresses a memory reference may access. Then, the method uses the approach proposed in [62] to perform multi-level data cache analysis for each task as if it were running on a single core. The results are CHMC and CAC for each memory reference on each cache level. Similar to the method in [63], for a task, for each cache set on each shared cache level, the number of conflicting cache blocks (i.e. *CCN*) is computed with respect to the interfering tasks running on other cores. *CCN* is used to derive the available cache space for a cache set s on that cache level l for a task t running on a core, which basically is $CahcheAssociativity - CCN_t^l(s)$. With the available cache space after considering the conflicts of other task, the task will be analyzed again to refine its memory reference's CHMC and CAC. In order to reduce the conflicts between tasks on the shared caches, the method also uses bypassing technique to arrange some instructions to use bypassing to reduce the number of evictions on the bypassed cache levels, which is similar to the bypassing approach introduced in [63].

Due to the difficulties of timing analysis caused by many types of interferences in real COTS multi-core processors, measurement-based methods have been proposed to tackle the problem. Although these approaches are not safe in general, they can reveal the aspects of various types of interference.

In [89], Nowotsch *et al.* perform the experiments on Freescale P4080 8-core processor which may be used in avionics in the future. A benchmark with different configurations to stress different shared resources is designed (including the interconnect, shared L2 cache, L3 cache/SRAM, and main DDR memory). The method focuses on comparisons of execution time variations because of concurrent accesses to L3 SRAM and to DDR memory

with different access patterns, and variations due to cache coherence. One interesting point in this paper is that – even the current applications do not share any data, the coherence protocol may cause a small variation of execution time due to coherence traffic on the bus.

In [90], Fernandez *et al.* choose the NGMP processor as the experimental target since it is designed to be used in the future space missions of the European Space Agency. Like in [89], micro-benchmarks are designed to stress different specific shared hardware resources in order to observe the influences they cause on the execution time. Different experiments are devised to take into account the shared AHB bus, the shared unified L2 cache, and the memory controller incrementally. The paper also carries out a test concerning the write-through policy of L1 data cache. In the end, the conclusion is that shared hardware resource can cause a big execution time variation even on the COTS processor designed for space domain.

2.3.3 Path Analysis

It should be noted that the final objective is to derive the WCET for the whole program via **path analysis** after the WCET for each BB is derived through the analyses considering micro-architecture features. In general, the path analysis problem is undecidable and equivalent to the halting problem [83]. However, absence of unbounded recursions and unbounded loops, which is common in CPS software, makes this problem decidable.

The state-of-the-art path analysis approach is called implicit path enumeration technique (IPET) which is proposed in Li's seminal work [83]. The IPET has a very significant influence on the timing analysis research area (it has been used and/or extended by many researchers). The basic idea of the IPET is to convert the original problem to a set of integer linear programming (ILP) problems, which does not explicitly enumerate program paths but implicitly considers them in its solution. The converted problem is to maximize the objective function $\sum_{i=1}^{N} c_i x_i$ where N is the number of BBs, c_i is the derived WCET for a BB bb_i, and x_i is the number of times bb_i is executed. In a loop- and recursion-

bounded program, x_i cannot be any number, and these variables should be subject to a set of program structural and functionality constraints. The structural constraints are used to express control flow information of the program: each edge of the procedure's CFG has a variable d to represent how many times this control flow is take; for a BB bb_i of the CFG, the number of times it is executed equals the sum of the control flow going into it, and the sum of the control flow going out from it; thus, for a bb_i we have a constraint $x_i = \sum_{e_j \in in(bb_i)} d_j = \sum_{e_k \in out(bb_i)} d_k$ where $in(bb_i)$ denotes the set of all incoming edges of bb_i, $out(bb_i)$ denotes the set of all outgoing edges of bb_i, and d variable has the same index as the corresponding edge. In terms of procedure invocation, the flow going into the procedure via CG edges is the connection between the caller(s) and the callee. The functionality constraints are used to denote loop bounds and other path information that depend on the functionality of the program: given a loop and its bound $[1, LB]$, if x_i is the number of execution times of the BB just before entering the loop, and x_j is the number of execution times of the first BB inside the loop, we can have the loop bound constraint $x_i \leq x_j$ and $x_j \leq LBx_i$; also for some exclusively executed BBs, e.g. bb_i and bb_j can exclusively execute once, we can have a non-linear constraint $(x_i = 1 \wedge x_j = 0) \vee (x_i = 0 \wedge x_j = 1)$, which can be transformed into two sets of linear constraints, $x_i = 1, x_j = 0$ and $x_i = 0, x_j = 1$, and each of these two sets is combined with the other constraints to be solved by an ILP solver and the one gives the bigger solution will give the WCET. The authors also argue that IPET performs path analysis efficiently in practice, although solving a general ILP problem is a NP-complete problem.

The most common WCET estimation method is to combine abstract interpretation and integer linear programming (AI + ILP) [85]: AI is used to predict safely the processor behavior (e.g. caches and pipeline) and ILP is used to determine one path on which the upper bound for the execution time is computed. There is also an interesting debate about whether model checking is useful for WCET estimation. In [91], Wilhelm argues that model checking may suffer from state-space explosion although the exhaustiveness yields

more precise results. Wilhelm also points out that the result computed by using AI + ILP does not lead to a significant loss of precision based on the practical experience. However, in [92] Metzner argues that model checking is perfect for estimating the WCET. Indeed, employing model checking for processor-behavior analysis can improve the results due to the avoidance of over-approximations induced by abstraction. Nevertheless, Metzner in [92] does not tackle the fundamental reason why Wilhelm in [91] thinks model checking is not appropriate – scalability.

When taking into account all necessary micro-architecture features, using model checking to derive the WCET is indeed not appropriate with respect to the required analysis time and memory space. However, if model checking is used only for searching the worst-case path, after using abstract interpretation for deriving the WCET for each BB, model checking may be feasible. A question is: if both IPET and model checking can discover the desired path, but which one is better? In [93], Lv *et al.* perform several experiments to compare the performances of IPET and model checking. The paper does not report which ILP solver is used, but only mentions that CPLEX or lp_solve can be used. Two model checkers are used for the comparison, SPIN and NuSMV. According to the experimental results, ILP yields very good performance, while model checking only works for simple programs. When there are large loop bounds and complex program structures, SPIN runs out of memory, and if the WCET is relatively large, NuSMV can not finish checking within feasible time. Since the used benchmarks contain no more than 300 lines of code, it is believed that model checking will meet scalability problems when analyzing large systems.

2.4 Preemption Delay Analysis

The related work for timing analysis summarized above is mainly for deriving the WCET of a single task without considering the effects of other tasks (i.e. only intra-task interference is considered). While this is valid for non-preemptive real-time systems, it may not hold when preemptive scheduling is used. When a lower-priority task (**preempted**

task) is preempted due to the activation of a higher-priority task (**preempting task**), the system states for the preempted task will be different after the preemption. One prominent example is the cache – there may be much more cache misses due to the preemption compared to the one without preemption. This is because when a preempting task is running, some memory blocks may be evicted; when the preempted task resumes, it needs to reload these evicted memory blocks. The preemption delay increases the WCET of the task, and may cause the preempted task miss its deadline if there are many preemption points and the preemption delay is expensive. Therefore, integrating schedulability analysis with preemption delay analysis is needed to ensure the tasks will meet their timing constraints.

A very early seminal research on combining schedulability analysis with preemption delay can be found in [94]. In [94], Lee *et al.* use the response time R_i of a task τ_i to do schedulability analysis, namely if $R_i < d_i$ where d_i is the deadline of τ_i is met for each task, then this task set is schedulable. In order to take into account the additional time caused by preemptions, the recurrence equation of response time is modified, which becomes $R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil \times C_j + \Delta_i$, where C_i is the computation time (without considering preemption delay), $hp(i)$ gives the set of higher-priority tasks than τ_i, T_j is the period of task τ_j, and Δ_i is the delay caused by preemption. The objective is to derive the R_i through an iterative solving process (which starts from the highest priority task since it does not suffer from any preemption). The Δ_i is calculated as number of evicted cache blocks $\times$ cache reload time. Since one can conservatively assume every cache block is evicted by the preempting tasks, this can cause a huge overestimation which makes the task set unschedulable even it is actually schedulable (since the derived overestimated response time may be over the deadline). In order to tighten the estimated preemption delay, the work proposes to only consider the useful cache blocks of the preempted task which are possibly used by the preempted task after resumption. The useful cache blocks are derived by carrying out two data-flow analysis on the CFG: one is for reaching cache blocks (like to derive reaching definitions) and the other one is for live

cache blocks (like to derive live variables). Therefore, the useful cache blocks at a program point is the intersection of the reaching cache blocks and the live cache blocks. Later, the authors realize preempting tasks will not evict all the useful cache blocks, so in [95], they also perform data-flow analyses on the preempting tasks to derive the used cache blocks. Thus, the preemption can at most affect the cache blocks that are in the intersection of useful cache blocks and used cache blocks.

In [96], Tomiyama *et al.* propose to use the ILP method to derive the maximum number of possible used cache blocks in a preempting task. However, the work does not consider the preempted task, namely they assume every cache block is useful in the preempted task which will lead to an overestimation. In [97], Negi *et al.* find it is possible to refine the useful cache blocks and used cache blocks by considering paths separately. Thus, the approach records the possible cache state path-wise at a program point instead of keeping track of possible memory blocks for each cache block independently. Although the method can reduce the over-approximation, the space overhead to track every possible cache state may be overwhelming. Therefore, binary decision diagram can be used to represent the possible cache states at a program point to save space.

Although the method proposed in [97] can give more precise analysis results for one preemption, it does not consider multiple preempting tasks which preempts a task multiple times. In [98], Staschulat *et al.* notice when a task with a lower-priority gets preempted multiple times, its preemption delay is much less than $\Delta \times N$ where Δ is the worst-case preemption delay a preempting task can cause to the preempted task and N is the maximum number of possible preemptions that a preempting task can cause. The reasons for this are: (1) there may be nested preemptions in which case the preempted task never resumed; (2) each instance of the preempting task will not evict the same number of memory blocks, or even multiple preemptions will not evict any memory blocks after the first time. It also considers the remaining memory blocks among different activations in order to tighten the WCET. Later in [99], schedulability analysis in the presence of preemption delay for each

task is also considered.

The previous mentioned work only considers instruction caches without taking into account data caches. As stated in section 2.3, data caches are much harder to analyze due to statically unknown memory reference patterns. As mentioned before in [70], Ramaprasad *et al.* propose using CME to derive the exact patterns of data cache references. In [100], Ramaprasad *et al.* extend their work to analyze the data cache-related preemption delay by using their method proposed in [70]. For each task, the data reference chains are established to represent the reuse relations, so the worst-case preemption delay happens when the most number of chains are cut by the preempting task. For multiple preemptions, the delays can be ordered from the worst-case cut to the best-case cut. It is also noticed that in the presence of preemption delay the critical instant in the theory of schedulability analysis is not necessarily the point in time when every task is activated. In [101], the method is further improved by taking into account the possible range of an preemption. The improved method increasingly uses a timing analyzer to derive the possible program points when an preemption happens considering all the delays to rule out infeasible program points where the corresponding preemption can happen.

In [102], Tan *et al.* summarize their work on the study of cache-related preemption delay. Most of their methods are similar to those taking into account both useful cache blocks for the preempted task and used cache blocks for the preempting tasks. The notable difference is the work considers n-way set associative caches other than merely direct-mapped caches like all other work mentioned above. The method is straightforward: partitioning the memory blocks into different sets and counting the number of conflicts. It is also noticed that indirect preemption needs to be considered when calculating the response time of a low-priority task.

A very interesting view is presented in [103]. Altmeyer *et al.* notice that the traditional methods (e.g. the methods presented in [94], [96], and [97]) calculate the WCET and CRPD (cache-related preemption delay) separately, and may count one cache miss twice (once for

WCET calculation and the other for CRPD calculation). The reason for this is the useful cache blocks are those that *may* be at a program point and *may* be used in the future. Thus, the work proposes to find the subset of the traditional useful cache blocks that are *definitely* in the cache at the program point and *may* be used in the future. Therefore, these memory blocks are actually in the abstract cache state of the *must* analysis at this program point, and are called definitely-cached useful cache blocks. One problem of using the definitely-cached useful cache blocks to compute CRPD is the resultant CRPD may not be safe (since the set of definitely-cached useful cache blocks is an underestimation of the set of useful cache blocks), but the work proves that the combination of the over-approximated WCET and the possibly under-approximated CRPD gives a safe estimation in terms of execution time. The reason for this is the notion of definitely-cached useful cache blocks rules out the possibility of counting a cache miss twice in the separated calculation processes of WCET and CRPD.

In [104], Chattopadhyay *et al.* make the first attempt to study the CRPD in the presence of multi-level non-inclusive caches. The work finds there may be indirect effects of preemption that can cause additional L2 cache misses aside from the cache misses derived by the traditional methods for L1 cache. A framework that is fit for analysis of two-level non-inclusive caches is proposed. The main point is that if a reference was L1 *AH* in the absence of preemption, it may not be L1 *AH* after the preemption. If a reference cannot be guaranteed as L1 *AH* after preemption, it may make an additional conflict in the corresponding L2 cache set. The additional L2 conflicts may evict a block *m* from the corresponding L2 cache set at a program point, where *m* were definitely in the cache if there were no preemption. Therefore, if a reference to *m* were L2 *AH* in the absence of preemption, it may suffer an L2 cache miss in the presence of preemption. Basically, the analysis of indirect effects of preemption is to bound the number of additional L2 cache misses due to additional L2 conflicts.

2.5 Tools and Benchmarks

Formally analyzing and/or verifying properties of binary executable programs has attracted significant attention. Thus, many open source and commercial tools have been developed and utilized to solve many problems in CPS. Due to the large amount of work, here we only state several well-known tools and benchmarks that are influential and useful for our research.

In [37], Song *et al.* introduce the BitBlaze project which aims at a unified binary analysis platform for different security problems. BitBlaze consists of three parts – Vine for static analysis, TEMU for dynamic analysis, and Rudder for mixed concrete and symbolic execution. BitBlaze has been successfully applied to solving many security problems, e.g. crash analysis [105], malware detection and analysis [106], deviation detection [107], and vulnerability analysis [108]. Although BitBlaze focuses on the security perspectives, many of its concepts and sub-parts are related to this thesis – for instance, Vine uses an intermediate language for assembly (ILA) which is proposed to realize generic static analysis on binaries.

Most relevant tools are related to timing analysis. In the following, we will briefly describe three representative tools (which are commercial, open sourced, and free but close sourced respectively) in the timing analysis field.

A well-known WCET analysis tool is from AbsInt company called aiT WCET analyzer [40]. It is a commercial tool and has been used to analyze many real-life systems – Airbus and Volvo have used the tool to estimate and verify the timing properties for their products (e.g. the flight control software in Airbus 380 and the control software in Volvo CE vehicles). The success of this tool shows how the timing analysis research can be used in real systems. The processors it supports include many real models of ARM, PowerPC, TriCore, TI DSP, and so forth. An excellent advantage of this tool is the integration with many control software development tools, including SCADE, STATEMATE, Ascet/SD, and MATLAB/Simulink.

In [109], Li *et al.* introduce their open source WCET analysis tool called Chronos. At first, Chronos supports the analysis of (1) out-of-order pipelines [48], (2) various dynamic branch prediction schemes [51], and (3) instruction caches, and the interaction among these different features [51] [110]. Later, the support for data caches is added in [73] and the support for multi-core processors with TDMA interconnects is added in [111]. However, Chronos only supports the processor model of SimpleScalar simulator, which is a popular cycle-accurate micro-architectural simulator [112]. Although the tool can target a simulated processor model so that the processor can be configured with different pipeline, branch prediction, and instruction-cache options, it does not support other architectures like ARM or PowerPC.

In [113], Ballabriga *et al.* present their WCET toolbox called OTAWA. The aim of OTAWA is to provide an open framework that could be used by researchers to implement their analyses and to combine them to already implemented ones. It is not a open source tool but a toolbox, which means that it comes as a C++ library that can be used to develop WCET analysis tools. However, OTAWA includes a number of algorithms which make the extension really easy (example tools are distributed with the library). OTAWA has the ability to support various target hardware configurations (via XML descriptions) as well as various instruction sets, e.g. PowerPC, ARM, TriCore, HCS12, and Sparc. In addition to the WCET analysis, it also includes a code processor that builds and runs a cycle-accurate simulator, which is generated on top of the SystemC library and matches the XML description of the target architecture. Cycle-accurate simulation can be used to gain empirical results against the derived analysis results.

The three tools listed above apply a similar process:

1. CFG reconstruction (CFG is the representation of the program in most analyses)
2. High-level analysis (e.g. using value analysis to analyze loop bounds)
3. Low-level analysis (e.g. analyzing the effects of cache on timing)

4. Path analysis (e.g. using IPET to find the longest execution path in an implicit way)

In terms of benchmarks, the Mälardalen WCET research group collects and maintains a large number of WCET benchmark programs, used to evaluate and compare different types of WCET analysis tools and methods [114]. The benchmarks are collected from several different research groups and tool vendors around the world. There are more than 30 benchmarks which range from basic binary search program to generated control software code.

Another benchmark that is useful is called PapaBench which is maintained by the TRACES group in France [115]. The PapaBench is based on the Paparazzi project that represents a real-time application, developed to be embedded on different Unmanned Aerial Vehicles (UAV). It is designed to be valuable for experimental work in WCET computation and may be also useful for schedulability analysis. The benchmark also provides a high level AADL model, which reflects the behaviors of each component of the system and their interactions. Unlike other usual WCET benchmarks, PapaBench is based on a real and complete real-time embedded application.

Chapter 3

Generic Value-Set Analysis on Low-Level Code

Cyber-physical systems (CPS) are complex systems that are characterized by tight interactions between the physical dynamics, computational platforms, communication networks, and control software. Many CPS are safety-critical or mission-critical systems, such as aerial vehicles and defense missiles, which means any failure may cause a great damage. Since software bugs are notoriously responsible for many system failures, it is an essential task to analyze/verify various properties of CPS software to guarantee it conforms to the specification.

When analyzing/verifying many CPS software properties, it may be insufficient to only consider this software in its high-level form, e.g. its source code. Often, we also need to analyze/verify these properties in its low-level form, e.g. its machine code, namely we need to take into account the compilation that transforms the source code into the low-level code and the interactions between the compiled software and its underlying computational platform; otherwise, even the high-level code is verified, compiler-induced bugs or unexpected architectural limitations may still cause the system to fail. For example, it is reported in [116] that every tested compiler is found to be able to generate wrong code silently; and a Patriot missile failed to intercept a Scud missile in the Gulf War due to the precision error of time calculation using a 24-bit fixed point register.

Value-set analysis (VSA) has been proposed to simultaneously perform numeric and pointer analyses on low-level code [29], which can be used to analyze/verify various control software properties (e.g. variable range and saturation) and security vulnerabilities (e.g. buffer overflow and side channels) for a specific platform. However, the original work only targets at the Intel x86 instruction set architecture (ISA). Thus, if we want to perform VSA on binaries in other ISAs, we need to make changes repeatedly to take into account the

semantics of their instructions, which can be a tedious and error-prone process.

While there is some uniqueness in different ISAs, many instructions of an ISA have their counterparts in another ISA, and they share a lot of similarities in their semantics. Thus, if we can use an intermediate language to encode the instructions of different ISAs capturing their computational semantics, we can use one generic VSA program which analyzes the translated binaries in this intermediate language, instead of modifying the VSA programs to target at different ISAs.

Moreover, VSA uses an abstract numeric domain called strided-interval. While strided-interval domain captures more precise information than the traditional interval domain, it cannot precisely track the set of numbers when some of numbers in the set are big enough such that they are interpreted as overflow with respect to two's complement representation.

The main contributions of this chapter are: (1) we extend the original strided-interval domain in order to prevent huge over-approximation from happening in the case of wrap-around computations; (2) we define several operations on the extended strided-interval domain which can more precisely track the set of structured numbers; (3) we use an Intermediate Static Analysis Language (iSAL) with a straightforward concrete semantics to encode instructions of different ISAs; (4) we define an abstract semantics for the iSAL in the value-set abstract domain to facilitate any VSA program writing; (5) we give an example to show the feasibility of the generic VSA approach, which is to precisely resolve indirect branch target addresses. This work has been published in [10].

The rest of the chapter is organized as: Section 3.1 briefly describes VSA and presents the extension to the original strided-interval domain with a set of operations; Section 3.2 introduces the syntax and concrete semantics of iSAL and defines abstract semantics with respect to VSA; Section 3.3 discusses some issues when using iSAL to perform VSA; Section 3.4 presents an example on resolving indirect branches and Section 3.5 concludes this work and states some future work.

3.1 Value-Set Analysis

In this section, we first briefly review VSA, and then we argue why it is necessary to extend the original strided-interval domain, and redefine several operations on the extended domain.

VSA combines numeric analysis and pointer analysis together, and its goal is to determine an over-approximation of the set of numeric values and addresses at each program point [29]. It is a flow-sensitive, context-sensitive, and interprocedural static analysis approach based on abstract interpretation [9].

In order to avoid dependence on absolute memory addresses (since some of them may not be determined statically), the memory model in VSA consists of a set of separated memory regions, and each memory region is an abstract memory space which corresponds to the set of all concrete memory spaces with respect to specific run-time properties, e.g. all the possible stack frames of a procedure invocation are aggregated as a memory region. There are three types of memory regions in VSA: the global-region for statically allocated variables in the program, local-regions for locally allocated variables in the run-time stack, and heap-regions for dynamically allocated variables in the heap. While there is only one global-region, there are as many local-regions as procedures.

An abstract address in a memory region can be represented by $\langle \textit{memory-region}, \textit{value} \rangle$, which corresponds to the set of all memory addresses a variable can have. A global variable has a fixed address which can be represented by $\langle \textit{global-region}, \textit{address} \rangle$, while a local variable has a varying address but a fixed offset to the varying base address of the stack frame and its abstract address can be represented by $\langle \textit{local-region}, \textit{offset} \rangle$.

The explicitly referenced variables in the source code are accessed by using their addresses in the machine code, VSA needs to recover the variables from low-level code, and represent them by using abstract locations (*a-locs*). An *a-loc* is a variable-like entity which may have an explicit boundary (e.g. registers) or an implicit boundary (e.g. variables allocated in the memory).

Given an *a-loc*, the abstract state of VSA maps it to a value-set. A value-set is a function that maps a memory region to a strided-interval (*SI*). A strided-interval is an abstract object used to represent a set of structured integers with a fixed stride. A k-bit strided-interval $s[l,u]$ where $-2^{k-1} \leq l \leq u \leq 2^{k-1}-1$, $0 \leq s \leq 2^k-1$ represents the set $\{i | i = l + n \times s \ \wedge \ n \geq 0 \ \wedge \ i \leq u\}$. Depending on the type of the given memory region, the mapped strided-interval has different semantics: if the memory region is the global-region, the mapped strided-interval represent a set of numeric values which are the values held by the *a-loc* in some executions; otherwise, the mapped strided-interval represent a set of offsets in the memory region, and each $\langle$memory-region, *offset*$\rangle$ pair is an abstract address with respect to the memory region. For a given memory region, the mapped strided-interval may be a $\perp$ which means the empty set of values. If the mapped strided-interval is a $\top$, it is the set of all the representable values $1[-2^{k-1}, 2^{k-1}-1]$.

3.1.1 Extension to Strided-Interval

An important problem of the original strided-interval representation is that if l is required to be less than or equal to u (i.e. $\forall s[l,u], l \leq u$), it loses the ability to precisely track the set of numbers when some of the numbers in the set lead to an overflow interpretation with respect to two's complement representation. For example, on a 16-bit architecture, if a strided-interval **4[0x7FF0, 0x7FFC]** is added to another interval **0[4, 4]**, according to the addition operation defined in [30], the resultant strided-interval will be **1[0x8000, 0x7FFF]**, i.e. the $\top$ element in the *SI* domain. Although the result is sound, basically it treats the upper bound of the calculation as an overflow. Thus, when using $\top$ to safely capture all possible values, the representation is not precise.

In the above example the interval **4[0x7FF4, 0x8000]** would be a more precise and also sound result, in which l represents a signed positive number $2^{15}-12$ and u represents a signed negative number -2^{15}. Thus, a better decision is to eliminate the constraint $l \leq u$ in order to allow both l and u to be any number of the range $[-2^{k-1}, 2^{k-1}-1]$, which induces

the proposed extended strided-interval (*ESI*) domain that is similar to the *CLP* domain in [32]. Let γ denote the concretization function which maps an extended strided-interval $s[l,u] \in ESI$ to a set of integers, and we have

$$\gamma(s[l,u]) = \begin{cases} \{i | i = l + n \times s \wedge n \geq 0 \wedge i \leq u\} & if\ l \leq u \\ \{i | i = l + n \times s \wedge n \geq 0 \wedge i \leq u + 2^k\} & otherwise \end{cases}$$

3.1.2 Operations on Extended Strided-Interval

There are six groups of operations on *ESI*, which are arithmetic, shift, bit-wise, set, comparison, and truncation operations. Let us assume the underlying platform is a n-bit architecture, and let max_u be the number $2^n - 1$, max_s be the number $2^{n-1} - 1$, and min_s be the number -2^{n-1}. For an arbitrary number x, $n(x)$ denotes the lowest n bits representation of x, and $tz(x)$ denotes the number of trailing zeros in x's representation.

In addition, let us define several functions that are used in the operations: let $sg(p,q,s)$ return the *smallest* number that is *greater* than p and can be reached by q in multiple s strides, and let $gs(p,q,s)$ return the *greatest* number that is *smaller* than p and can be reached by q in multiple s strides (the computation wraps around in terms of n-bit).

Let $dsi_u(s[l,u])$ denote an ordered set of disjoint strided-intervals, each of which represents a maximal sub-interval with respect to unsigned integers. Depending on the $s[l,u]$, this set can be a singleton (e.g. $dsi_u(1[10,20]) = \{1[10,20]\}$), or has two members (e.g. $dsi_u(1[-2,2]) = \{1[0,2], 1[max_u - 1, max_u]\}$). Let $fst(dsi_u(s[l,u]))$ denote the first member in this ordered set, and let $snd(dsi_u(s[l,u]))$ denote the second one if it exists, or $\perp$ if the set is a singleton. Similarly, let $dsi_s(s[l,u])$ denote an ordered set of disjoint strided-intervals, each of which represents a maximal sub-interval with respect to signed integers.

3.1.2.1 Arithmetic Operations

For the **addition** operation $s_1[l_1,u_1] +^{si} s_2[l_2,u_2]$, let us assume $s = \gcd(s_1,s_2)$, $\hat{l} = l_1 + l_2$, and $\hat{u} = u_1 + u_2$ without overflow, i.e. $\hat{l}$ and $\hat{u}$ have enough bits to contain the sums. From now on, let us assume all the arithmetic operations on numeric values will not induce overflow. The operation is defined as

$$s_1[l_1,u_1] +^{si} s_2[l_2,u_2] = \begin{cases} s[n(\hat{l}),n(\hat{u})] & if\ \hat{u} - \hat{l} < 2^n \\ s[sg(min_s,\hat{l},s),gs(max_s,\hat{u},s)] & else\ if\ s = 2^m \\ 1[min_s,max_s] & otherwise \end{cases}$$

For the **negation** operation $-^{si} s_1[l_1,u_1]$, since *ESI* allows $l_1 > u_1$, we can have

$$-^{si} s_1[l_1,u_1] = s_1[-u_1,-l_1]$$

which is simpler but more precise than the negation operation for the original strided-interval defined in [30]. Thus, for the **subtraction** operation $s_1[l_1,u_1] -^{si} s_2[l_2,u_2]$, we have

$$s_1[l_1,u_1] -^{si} s_2[l_2,u_2] = s_1[l_1,u_1] +^{si} (-^{si} s_2[l_2,u_2])$$

For the **unsigned multiplication** operation $s_1[l_1,u_1] \times_u^{si} s_2[l_2,u_2]$, we define $prod_u$ as

$$prod_u = \{p | \exists s_x[l_x,u_x] \in dsi_u(s_1[l_1,u_1]), s_y[l_y,u_y] \in dsi_u(s_2[l_2,u_2]) : p = l_x \times l_y \vee p = u_x \times u_y\}$$

Let $\hat{l}_1$ be the lower bound of the interval $fst(dsi_u(s_1[l_1,u_1]))$ and let $\hat{l}_2$ be the lower bound

of the interval $fst(dsi_u(s_2[l_2,u_2]))$. We have

$$\hat{s}_1 = \gcd(s_1 \times s_2, \hat{l}_1 \times s_2, \hat{l}_2 \times s_1)$$

$$\hat{s}_2 = \gcd(\hat{s}_1, \hat{l}_1 \times 2^n, s_1 \times 2^n)$$

$$\hat{s}_3 = \gcd(\hat{s}_1, \hat{l}_2 \times 2^n, s_2 \times 2^n)$$

$$\hat{s}_4 = 2^{tz(\gcd(\hat{s}_2,\hat{s}_3))}$$

$$\hat{s} = \begin{cases} \hat{s}_1 & if\ |dsi_u(s_1[l_1,u_1])| = 1 \wedge |dsi_u(s_2[l_2,u_2])| = 1 \\ \hat{s}_2 & if\ |dsi_u(s_1[l_1,u_1])| = 1 \wedge |dsi_u(s_2[l_2,u_2])| = 2 \\ \hat{s}_3 & if\ |dsi_u(s_1[l_1,u_1])| = 2 \wedge |dsi_u(s_2[l_2,u_2])| = 1 \\ \hat{s}_4 & if\ |dsi_u(s_1[l_1,u_1])| = 2 \wedge |dsi_u(s_2[l_2,u_2])| = 2 \end{cases}$$

Since the product of two n-bit numbers should be contained in $2n$-bit, we have the multiplication operation to generate two *ESI*s: the first *ESI* corresponds to the high n-bit of the product, and the second one corresponds to the low n-bit of the product. Let $p_{min} = \min(prod_u)$, and $p_{max} = \max(prod_u)$. We have

$$s_1[l_1,u_1] \times_u^{si} s_2[l_2,u_2] = \langle 1[\lfloor \frac{p_{min}}{2^n} \rfloor, \lfloor \frac{p_{max}}{2^n} \rfloor], \hat{s}[\hat{l},\hat{u}] \rangle$$

$$\text{where } [\hat{l},\hat{u}] = \begin{cases} [n(p_{min}), n(p_{max})] & if\ p_{max} - p_{min} \leq max_u \\ [sg(0, 2^{tz(\hat{s})}, \hat{s}), max_u] & otherwise \end{cases}$$

The **signed multiplication** operation $s_1[l_1,u_1] \times_s^{si} s_2[l_2,u_2]$ is similar but makes use of dsi_s instead of dsi_u.

For the **unsigned division** operation $s_1[l_1,u_1] \div_u^{si} s_2[l_2,u_2]$, we have $quot_u$ defined as

$$quot_u = \{q | \exists s_x[l_x,u_x] \in dsi_u(s_1[l_1,u_1]), s_y[l_y,u_y] \in dsi_u(s_2[l_2,u_2]) : q = \lfloor \frac{l_x}{u_y} \rfloor \vee q = \lfloor \frac{u_x}{l_y} \rfloor \}$$

Let $\hat{l}_2$ be the lower bound of $fst(dsi_u(s_2[l_2,u_2]))$, and let $\hat{q} = \lfloor \frac{s_1}{\hat{l}_2} \rfloor$ and $\hat{r} = \frac{s_1}{\hat{l}_2} - \hat{q}$. We have

$$\hat{s} = \begin{cases} \hat{q} & if\ |\gamma(s_1[l_1,u_1])| = 1 \wedge \hat{r} = 0\ \wedge\ |dsi_u(s_1[l_1,u_1])| = 1 \wedge \hat{q} = 2^m \\ 1 & otherwise \end{cases}$$

$$s_1[l_1,u_1] \div_u^{si} s_2[l_2,u_2] = \hat{s}[\min(quot_u), \max(quot_u)]$$

For the **signed division** operation $s_1[l_1,u_1] \div_s^{si} s_2[l_2,u_2]$, it is similar but makes use of dsi_s instead of dsi_u, and the corresponding $quot_s$ also contains q that is either $q = \lfloor \frac{l_x}{l_y} \rfloor$ or $q = \lfloor \frac{u_x}{u_y} \rfloor$.

3.1.2.2 Shift Operations

Since a shift operation on a value (left or right, logical or arithmetic, but not circular) makes no difference when the numbers of bits to shift are greater than n, we define $shn(s[l,u])$ as

$$shn(s[l,u]) = \{x | 0 \le x \le n \wedge x \in \gamma(s[l,u])\}$$

to give the set of numbers by which the shift operations are performed usefully. For the logical/arithmetic **left-shift** operation $s_1[l_1,u_1] \ll^{si} s_2[l_2,u_2]$, we extract the useful sub-interval from $s_2[l_2,u_2]$ for the operation:

$$x_{min} = \min(shn(s_2[l_2,u_2]))$$

$$x_{max} = \max(shn(s_2[l_2,u_2]))$$

$$\hat{s}_2[\hat{l}_2,\hat{u}_2] = (2^{x_{min}} \times (2^{s_2} - 1))[2^{x_{min}}, 2^{x_{max}}]$$

$$s_1[l_1,u_1] \ll^{si} s_2[l_2,u_2] = snd(s_1[l_1,u_1] \times_s^{si} \hat{s}_2[\hat{l}_2,\hat{u}_2])$$

For the **logical right-shift** operation $s_1[l_1,u_1] \gg_l^{si} s_2[l_2,u_2]$, it is similar but the result is the quotient of $s_1[l_1,u_1] \div_u^{si} \hat{s}_2[\hat{l}_2,\hat{u}_2]$.

For the **arithmetic right-shift** operation $s_1[l_1,u_1] \gg_a^{si} s_2[l_2,u_2]$, it is also similar but the result is the quotient of $s_1[l_1,u_1] \div_s^{si} \hat{s}_2[\hat{l}_2,\hat{u}_2]$. Different from logically shifting, an arithmetic right-shift operation needs to fill in the sign bit of the shifted number. In the case of $|shn(s_2[l_2,u_2])| = 0 \wedge |\gamma(s_2[l_2,u_2])| \neq 0$, it means every number in $\gamma(s_2[l_2,u_2])$ is greater than n. Thus, we also have

$$if\ |shn(s_2[l_2,u_2])| = 0 \wedge |\gamma(s_2[l_2,u_2])| \neq 0$$

$$s_1[l_1,u_1] \gg_a^{si} s_2[l_2,u_2] = \begin{cases} 0[-1,-1] & if\ \forall y \in \gamma(s_1[l_1,u_1]) : y < 0 \\ 0[0,0] & if\ \forall y \in \gamma(s_1[l_1,u_1]) : y \geq 0 \\ 1[-1,0] & otherwise \end{cases}$$

3.1.2.3 Bit-Wise Operations

Since all the bit-wise operations, including the **bit-wise not** operation $\sim^{si} s_1[l_1,u_1]$, the **bit-wise or** operation $s_1[l_1,u_1] \,|^{si} s_2[l_2,u_2]$, the **bit-wise and** $s_1[l_1,u_1] \,\&^{si} s_2[l_2,u_2]$, and the **bit-wise xor** operation $s_1[l_1,u_1] \oplus^{si} s_2[l_2,u_2]$, are similar to the corresponding one defined in [30], we will not state them here.

3.1.2.4 Set Operations

For the **set union** operation $s_1[l_1,u_1] \cup s_2[l_2,u_2]$, let $\hat{s} = \gcd(s_1,s_2)$, and let us define four boolean variables: $b_1 = l_2 \in \gamma(\hat{s}[l_1,u_1])$, $b_2 = u_2 \in \gamma(\hat{s}[l_1,u_1])$, $b_3 = l_1 \in \gamma(\hat{s}[l_2,u_2])$,

and $b_4 = u_1 \in \gamma(\hat{s}[l_2, u_2])$. We have

$$s_1[l_1,u_1] \cup s_2[l_2,u_2] = \begin{cases} \hat{s}[l_a,u_a] & if\ b_1 \wedge b_2 \wedge b_3 \wedge b_4 \\ \hat{s}[l_1,u_1] & else\ if\ b_1 \wedge b_2 \\ \hat{s}[l_2,u_2] & else\ if\ b_3 \wedge b_4 \\ \hat{s}[l_1,u_2] & else\ if\ b_1 \wedge b_4 \\ \hat{s}[l_2,u_1] & else\ if\ b_2 \wedge b_3 \\ \hat{s}_b[l_b,u_b] & otherwise \end{cases}$$

$$\text{where } [l_a,u_a] = \begin{cases} [l_1,u_1] & if\ l_1 = l_2 \wedge u_1 = u_2 \\ [sg(min_s,l_1,\hat{s}), gs(max_s,u_1,\hat{s})] & otherwise \end{cases}$$

and the computation of $\hat{s}_b[l_b,u_b]$ is similar to the set union operation given in [32], so we will not state it here. The **set intersection** operation $s_1[l_1,u_1] \cap^{si} s_2[l_2,u_2]$ and the **set complement** operation $s_1[l_1,u_1] \backslash^{si} s_2[l_2,u_2]$ are similar to the ones defined in [32] and are not described here.

3.1.2.5 Comparison and Truncation Operations

For the **comparison** operation $s_1[l_1,u_1] \mathscr{R} s_2[l_2,u_2]$, where $\mathscr{R}$ is a relational operator, we compare the ranges of the members of $dsi_{u|s}(s_1[l_1,u_1])$ and $dsi_{u|s}(s_2[l_2,u_2])$. If the operation is to compare unsigned numbers, we use dsi_u; otherwise we use dsi_s.

For the **truncation** operation $s_1[l_1,u_1] \downarrow^{si} s_2[l_2,u_2]$, we assume $s_2[l_2,u_2]$ give the set of numbers of bits kept in the truncated value. In order to be meaningful, the number of kept bits given by $s_2[l_2,u_2]$ should be smaller than n; otherwise the original number will not be truncated. We borrow $shn(s_2[l_2,u_2])$ from the shift operations defined above. Given a

number $x < n$, let us define $trun(s[l,u],x)$ as

$$trun(s[l,u],x) = \begin{cases} 0[l\&(2^x-1), u\&(2^x-1)] & if\ l = u \\ s[l,u] & if\ \forall y \in \gamma(s[l,u]) : y\&2^x = 0 \\ 1[0,2^x-1] & otherwise \end{cases}$$

and we have

$$s_1[l_1,u_1] \downarrow^{si} s_2[l_2,u_2] = \bigcup_{x \in shn(s_2[l_2,u_2])} trun(s_1[l_1,u_1],x)$$

3.2 Intermediate Static Analysis Language (iSAL) with VSA Semantics

The iSAL consists of 25 intermediate instructions which are used to encode the semantics of instructions of different ISAs. These instructions are selected between trade-offs in expressivity and compactness (namely, some instructions may be redundant since they can be represented by a combination of others, but their presence makes the translation much easier).

3.2.1 Syntax and Concrete Semantics

The syntax and concrete semantics of the intermediate instructions are shown in Tab. 3.1. From Tab. 3.1, we can observe that most of the intermediate instructions have three operands (only two of them have two operands, i.e. **not** and **brc**). In the table, we use r to restrict the operand to be a register, and use f to represent the operand is a status flag. There are no restrictions on s and t, namely, each of them can be either a register or an immediate number.

The first 10 instructions are arithmetic instructions. For an arithmetic operation $*$, let $*^m$ denote the result of this operation is in m-bit. Therefore, if the result needs more than m bits to represent, there is a potential overflow. Two functions, hi and lo, are defined as

using the high $\frac{m}{2}$ bits and the low $\frac{m}{2}$ bits of a m-bit number respectively. Furthermore, we use $*_u$ to denote that the operands are treated as unsigned numbers in the operation; and we use $*_s$ to denote that the operands are treated as signed numbers in the operation.

The next 3 instructions are about shift operations. For the left-shift operation, $\ll_0^n$ means the result is confined in n bits (discarding the bits higher than n) and 0 is shifted in from the right to the left, namely, the operation is the logical left-shift. For the right-shift operations, $\gg_0$ is the logical right-shift operation which shifts 0 in from the left to the right and $\gg_{msb(s)}$ is the arithmetic right-shift operation which shifts the sign bit of s in (i.e. the most significant bit of s given by the function *msb*).

We have 4 instructions for bit-wise operations, although we can just include **not** and **or** instructions and deduce **and** and **xor** instructions by De Morgan laws. Thus, there is a trade-off between compactness and expressivity.

The next 5 comparison instructions are used to check the relations between two operands. As the arithmetic instructions, they distinguish between signed and unsigned comparisons. If the designated relation is met, the first status flag operand will be set; otherwise, the flag will be cleared.

ld and **st** are the only two instructions to operate memory. The second operand s gives the load/store size in bytes and the third operand t gives the base address of the memory operation. Given an address range, the *mem* function returns the corresponding collection of memory cells.

The sequential control flow can only be changed by the conditional branch instruction, i.e. **brc** instruction. The unconditional branches instructions can be modeled by setting the first status flag operand as always-set.

Translation from a binary executable $\mathscr{B}$ in some ISA into the corresponding program $\mathscr{B}^T$ in iSAL can be achieved automatically provided the mapping of instructions is available. The encoded mapping captures the semantics of the instructions of the ISA using the iSAL.

Table 3.1: Syntax and Concrete Semantics of 25 Intermediate Instructions

Instruction	Concrete Semantics
add r, s, t	$r := s +^n t$
sub r, s, t	$r := s -^n t$
muluhi r, s, t	$r := hi(s \times_u^{2n} t)$
mululo r, s, t	$r := lo(s \times_u^{2n} t)$
mulshi r, s, t	$r := hi(s \times_s^{2n} t)$
mulslo r, s, t	$r := lo(s \times_s^{2n} t)$
divu r, s, t	$r := s \div_u t$
divs r, s, t	$r := s \div_s t$
modu r, s, t	$r := s \bmod_u t$
mods r, s, t	$r := s \bmod_s t$
shl r, s, t	$r := s \ll_0^n t$
shrl r, s, t	$r := s \gg_0 t$
shra r, s, t	$r := s \gg_{msb(s)} t$
and r, s, t	$r := s \,\&\, t$
or r, s, t	$r := s \mid t$
not r, s	$r := \sim s$
xor r, s, t	$r := s \oplus t$
cmpeq f, s, t	if $s = t$ then $set(f)$ else $clr(f)$
cmpleu f, s, t	if $s \leq_u t$ then $set(f)$ else $clr(f)$
cmples f, s, t	if $s \leq_s t$ then $set(f)$ else $clr(f)$
cmpltu f, s, t	if $s <_u t$ then $set(f)$ else $clr(f)$
cmplts f, s, t	if $s <_s t$ then $set(f)$ else $clr(f)$
ld r, s, t	$r := mem(t, t+s)$
st r, s, t	$mem(t, t+s) := r$
brc f, t	if $isset(f)$ then $goto(t)$

The concrete semantics of a binary executable program (and its translated intermediate program) considers every possible execution path in all possible environments. This concrete semantics may be an infinite mathematical object which is not computable. In order to make the analysis tractable, some form of over-approximation is needed. Abstract interpretation [9] is proposed to formalize the notion of over-approximation in a unified framework. Based on abstract interpretation, an analysis like VSA can be used to verify the properties of CPS software.

3.2.2 Abstract Domains for VSA

Following the original work of VSA (which is summarized in [29] and has been briefly described in Section 3.1), we define several abstract domains that are used in VSA.

In our virtual iSAL architecture, in addition to the registers in the encoded ISA, temporary registers can be declared and used in order to keep intermediate values in the process of an instruction execution. There can be as many temporary registers as needed. Let *NormLoc* denote the set of ordinary *a-locs* corresponding to target ISA registers, global variables, and local variables, let $FlagLoc$ denote the set of *a-locs* corresponding to status flags used in the encoding, and let *TempLoc* denote the set of temporary *a-locs* corresponding to the declared temporary registers and other entities which hold temporary values. We have *AbsLoc* defined as

$$AbsLoc = NormLoc \cup FlagLoc \cup TempLoc$$

Let *MemRgn* denote the set of all the memory regions, which include the single global-region and all the local-regions (since CPS software seldom use dynamic memory allocation, usually we can ignore heap-regions), and let VS denote the set of all the value-sets. Thus, we have

$$VS = MemRgn \rightarrow ESI_{\perp}$$

where $ESI_{\perp}$ is the lifted extended strided-interval domain, i.e. $ESI_{\perp} = ESI \cup \{\perp\}$. Thus, there is a special value-set $vs_{\perp} \in VS$ such that $\forall mr \in MemRgn : [mr \mapsto \perp]$. We also have another special value-set $vs_{\top} \in VS$ such that $\forall mr \in MemRgn : [mr \mapsto \top]$.

Let B^3 denote the Kleene three-valued logic domain, i.e. $B^3 = \{TRUE, FALSE, UNKOWN\}$. Let *gr* denote the global-region. Given a $b \in B^3$, we define an auxiliary function $bvs : B^3 \rightarrow$

VS as

$$bvs(b) = \begin{cases} vs_{\perp}[gr \mapsto 0[1,1]] & if\ b = TRUE \\ vs_{\perp}[gr \mapsto 0[0,0]] & if\ b = FALSE \\ vs_{\perp}[gr \mapsto 1[0,1]] & otherwise \end{cases}^{1}$$

Also, let us define $vsb : VS \rightarrow B^3$ as the inverse operation of *bvs*. In order to facilitate specifying the abstract semantics for the comparison instructions, we define a product domain FS as

$$FS = VS \times AbsLoc \times VS \times VS$$

Given a $fs \in FS$, the first component of fs (denoted as $fs\langle 1\rangle$) is the answer of function *bvs* applied to the result of a comparison, the second component ($fs\langle 2\rangle$) is the *a-loc* of the second operand in a comparison instruction, the third component ($fs\langle 3\rangle$) is the partition of the value-set mapped from $fs\langle 2\rangle$ that makes the comparison *TRUE*, and the last component ($fs\langle 4\rangle$) is the partition that makes the comparison *FALSE*.

An abstract state of VSA maps an *a-loc* $a \in AbsLoc$ to a $vs \in VS$ if $a \in NormLoc \cup TempLoc$, or to a $fs \in FS$ if $a \in FlagLoc$. Let $State$ denote the set of all the abstract states of VSA. We have

$$State = AbsLoc \rightarrow VS \cup FS$$

Since the translation for a given binary executable program is has a finite length and the target architecture has a fixed word size, all the domains described above are finite.

3.2.3 Abstract Semantics for VSA

Each intermediate instruction in the iSAL has an abstract semantic function for VSA which transforms an abstract state to another state(s). Let *IInst* denote the set of all the 25

[1] Given a function $f : A \rightarrow B$, let $f[x \mapsto y]$ mean $f(x) = y$ and $\forall a \in A \wedge a \neq x : f(a) = f(a)$

intermediate instructions. Formally, we have this semantic mapping function

$$sm : IInst \rightarrow (State \rightarrow State^{+})$$

which assigns each intermediate instruction an abstract semantic function on $State$. Let us also define an auxiliary function

$$al : IInst \times \{1,2,3\} \rightarrow AbsLoc$$

that maps the i^{th} operand of an intermediate instruction to its corresponding *a-loc*. For a two-operand instruction χ, i.e. **not** or **brc** instruction, let $al(\chi,3)$ give $\varepsilon \in TempLoc$ such that $\forall \sigma \in State : \sigma(\varepsilon) = vs_{\perp}$.

Let $ASB \subset IInst$ denote the set of intermediate instructions in the first three groups (i.e. the arithmetic, shift, and bit-wise instructions). Given an instruction $\alpha \in ASB$ and an abstract state $\sigma \in State$, we have

$$sm(\alpha)(\sigma) = \begin{cases} \sigma[al(\alpha,1) \mapsto op(\alpha)^{vs}\sigma(al(\alpha,2))] & if\ al(\alpha,3) = \varepsilon \\ \sigma[al(\alpha,1) \mapsto \sigma(al(\alpha,2))op(\alpha)^{vs}\sigma(al(\alpha,3))] & otherwise \end{cases}$$

where $op(\alpha)$ gives the corresponding operation the instruction semantically performing, and $op(\alpha)^{vs}$ denotes the operation is performed on VS domain. The operations on VS domain are based on the operations on strided-interval domain (ESI domain in our case), which are defined in [30].

Let $CMP \subset IInst$ denote the set of comparison instructions. Given an instruction $\beta \in$

CMP and an abstract state $\sigma \in State$, we have

$$sm(\beta)(\sigma) = \sigma[al(\beta,1) \mapsto fs] \text{ where}$$

$$fs\langle 1\rangle = bvs(al(\beta,2)op(\beta)^{vs}al(\beta,3)) \wedge$$

$$fs\langle 2\rangle = al(\beta,2) \wedge$$

$$fs\langle 3\rangle(op(\beta)^{vs})^{-1}\sigma(al(\beta,3)) = FALSE \wedge$$

$$fs\langle 4\rangle op(\beta)^{vs}\sigma(al(\beta,3)) = FALSE$$

where $(op(\beta)^{vs})^{-1}$ gives the inverse relational operation of $op(\beta)^{vs}$. The reason of using inverse operation is: in the case of the comparison giving *TRUE*/*FALSE* instead of *UNKNOWN*, $fs\langle 4\rangle / fs\langle 3\rangle$ is $vs_{\perp}$ and we assume a relational operation on $vs_{\perp}$ always gives *FALSE*. In terms of comparing two value-sets, we only compare them if they have the same VS type – either VS_{global} (i.e. having all the memory regions mapped to $\perp$ except for the global-region) or VS_{single} with the same valid memory region *mr* (i.e. having all the memory regions mapped to $\perp$ except for the *mr*); otherwise, the comparison gives *UNKNOWN*, and both $fs\langle 3\rangle$ and $fs\langle 4\rangle$ are set as $\sigma(al(\beta,2))$.

Let η be either a **ld** or a **st** instruction, which uses the second operand to specify the load/store size. Since there is barely an architecture that has a varying size in a specific load/store instruction, we can assume $vs_2 = \sigma(al(\eta,2)) \in VS_{global}$ and $vs_2(gr) = 0[w,w]$ where $\sigma \in State$ and w is a valid size value that can be loaded/stored in the target architecture. η also uses the third operand to specify the base memory address, which can be an address of a static object, or an address of an object that is allocated in stack. For a $vs \in VS$, let us define a function $rg : VS \rightarrow MemRgn$ such that $rg(vs)$ gives the global-region if $vs \in VS_{global}$; otherwise $rg(vs)$ gives the local-region of the procedure that is under analysis. Let $vs_3 = \sigma(al(\eta,3))$, and let $Addr$ be the set of *a-locs* that are constructed from w,

$rg(vs_3)$ and $\gamma(vs_3(rg(vs_3)))$. Let $\overleftarrow{\eta}$ be a **ld** instruction. We have

$$sm(\overleftarrow{\eta})(\sigma) = \sigma[al(\overleftarrow{\eta},1) \mapsto \sqcup^{vs}_{d \in Addr}\sigma(d)]$$

where $\sqcup^{vs}$ is the join operation on VS which is to memory region-wisely join extended strided-intervals.

In terms of storing, there may be some *a-locs* overlapping with the *a-loc*(s) being modified by a store instruction. Let $Ovlp$ be the set of *a-locs* that are overlapping with the *a-loc*(s) being modified by a **st** instruction $\overrightarrow{\eta}$. We have

$$sm(\overrightarrow{\eta})(\sigma) = \sigma \left[\begin{array}{l} \forall d \in Addr : d \mapsto \sigma(al(\overrightarrow{\eta},1)), \\ \forall o \in Ovlp : o \mapsto vs_\top \end{array} \right]$$

A **brc** instruction δ denotes the end of the current basic block. It is the only way to change the control flow and fork the current state σ depending on $fs = \sigma(al(\delta,1))$. We have

$$sm(\delta)(\sigma) = \begin{cases} \langle \sigma[fs\langle 2\rangle \mapsto fs\langle 3\rangle], \sigma_\bot \rangle & if\ vsb(fs\langle 1\rangle) = TRUE \\ \langle \sigma_\bot, \sigma[fs\langle 2\rangle \mapsto fs\langle 4\rangle] \rangle & if\ vsb(fs\langle 1\rangle) = FALSE \\ \langle \sigma[fs\langle 2\rangle \mapsto fs\langle 3\rangle], \sigma[fs\langle 2\rangle \mapsto fs\langle 4\rangle] \rangle & otherwise \end{cases}$$

where $\sigma_\bot$ means $\forall a \in NormLoc \cup TempLoc : \sigma(a) = vs_\bot \wedge \forall f \in FlagLoc : \sigma(f) = fs_\bot$. Therefore, a **brc** instruction partitions the σ into two ordered parts: the first true part is for the branch taken execution and the second false part is for the fall-through execution.

3.3 Value-Set Analysis of iSAL Programs

VSA of the translated program in iSAL intends to derive the fixed-points of the abstract states of each program point using iterations. In each iteration, we update the abstract states

according to the abstract semantics described above.

3.3.1 Handling Delay Slots

Several architectures (e.g. MIPS) use delay slots to compensate the performance loss when dealing with conditional branches.

Since the **brc** instruction intends to mean the end of a basic block and partitions the $\sigma \in State$ into a true part and a fall-through part, any intermediate instruction following a **brc** instruction will not change the value-sets of the true partition. However, in the presence of delay slots, this does not conform to the original code's semantics.

Fortunately, the instructions in the delay slots are arranged by the compiler which does not allow any of the instructions in the delay slots to have a dependency with the associated conditional branch. Thus, when we perform VSA on a basic block, if the last few instructions are used as delay slots, we can rearrange the order of the analysis by processing them before the corresponding conditional branch.

3.3.2 Join Function

A *join* semantic function is needed to combine the incoming abstract states when a basic block has more than one predecessors in the control flow graph (CFG). Given two abstract states $\sigma_1 \in State$ and $\sigma_2 \in State$, we have the join function $join : State \times State \rightarrow State$ defined as

$$join(\sigma_1, \sigma_2) = \begin{bmatrix} \forall a \in NormLoc : a \mapsto \sigma_1(a) \sqcup^{vs} \sigma_2(a), \\ \forall b \in TempLoc : b \mapsto vs_{\perp}, \\ \forall f \in FlagLoc : f \mapsto \sigma_1(f) \sqcup^{fs} \sigma_2(f) \end{bmatrix}$$

where $\sqcup^{fs}$ is the join operation on FS domain. Given a $fs_1 \in FS$ and a $fs_2 \in FS$, we define $\sqcup^{fs}$ as

$$fs_1 \sqcup^{fs} fs_2 = \begin{cases} \langle bvs(UNKNOWN), \varepsilon, vs_\perp, vs_\perp \rangle & if\ fs_1\langle 2\rangle \neq fs_2\langle 2\rangle \\ \langle fs_1\langle 1\rangle \sqcup^{vs} fs_2\langle 1\rangle, fs_1\langle 2\rangle, & otherwise \\ fs_1\langle 3\rangle \sqcup^{vs} fs_2\langle 3\rangle, fs_1\langle 4\rangle \sqcup^{vs} fs_2\langle 4\rangle \rangle & \end{cases}$$

When joining two abstract states, we discard the value-set information of temporary *a-locs*, since the information is not used in the new basic block. If a basic block has more than two predecessors, a successive joining is performed, i.e. $join(\sigma_n, join(\ldots join(\sigma_1, \sigma_2)))$. Moreover, if a predecessor ends with a **brc** instruction, depending on whether the new basic block is the target of that **brc** instruction, the true/false part of the resultant states is used.

3.3.3 Handling Input Dependent Value-Sets

When analyzing a program that has input dependent variables, for the sake of safety, these variables are supposed to be any possible numbers. In the context of VSA, an abstract state maps an *a-loc* corresponding to such an input dependent variable to $vs_\top$. Since the result of an operation on a $vs_\top$ is also a $vs_\top$, the propagation of $vs_\top$ will make the analysis imprecise or even useless.

In order to improve the precision of analysis, we keep track of the operations on $vs_\top$ until a **brc** instruction is met. Since the **brc** instruction partitions the state into two parts depending on some previous comparison, in each part, we use the information discovered by the comparison to refine the $vs_\top$ propagation chain. In the current work, we only keep track of the chains of linear operations so as to reduce the complexity.

3.4 Example – Handling Indirect Branches

As model-based control design tools become mature like Simulink, a large part of control software is designed by using formal specifications like finite-state machine (FSM) or state chart, and its C code is generated by using a code generator like Simulink Coder. Usually, the code generator opts for using *switch* statements to implement the transitions between states.

However, the compilers often implement *switch* statements by using indirect branches, whose presence makes reconstructing a whole CFG from a binary very challenging. Since a typically static analysis is performed on the extracted CFG, how to precisely resolve the target addresses of these indirect branches becomes essential.

As an example, we encode most of the instructions of MIPS ISA (without considering floating-point, coprocessor, and exception instructions) using the iSAL, and show VSA can precisely resolve the indirect branch instructions when reconstructing the CFG.

As shown in Fig. 3.1, the code has an input dependent variable x, and the *switch* statement relies on the value of the input. At the address **0x4002bc**, the **brc** intermediate instruction in the **beqz** MIPS instruction can determine: when v1 ≥ 6, v0 is 0 and the branch will be taken; otherwise, the control falls through. Thus, the partitioned true part of the state has v1 ≥ 6 and the false part has $0 \leq$ v1 ≤ 5. However, since the value of x is stored in a local variable whose address is given as (s8+12) in the binary code where s8 has already been set equal to sp (stack pointer register), we also need to handle the value-set mapped from the *a-loc* corresponding to (s8+12) in the true and false parts partitioned by **brc**, whereas the value of v0 at the address **0x4002c4** will still be any possible number as x.

From v0 at the address **0x4002d4**, we can observe that it can have 6 values ranging from **0x45bcd0** to **0x45bce4**, each of which is separated by 4. These 6 values are exact data addresses where the indirect branch target addresses are stored. However, since the target addresses stored at these 6 data addresses are not regularly structured, after the **lw**

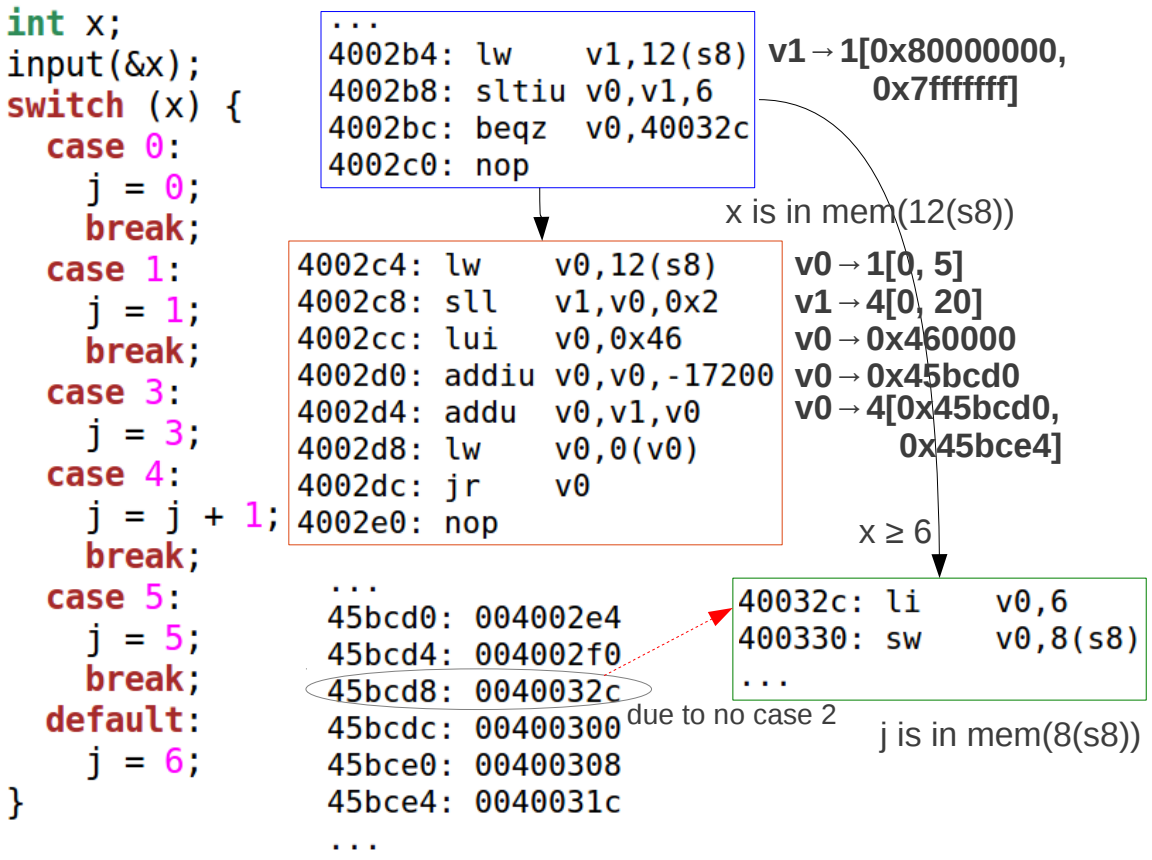


Figure 3.1: A C code snippet with *switch* statements and compiled MIPS code

instruction at the address **0x4002d8**, v0 will have an extended strided-interval **4[0x4002e4, 0x40032c]** which corresponds to 19 values. In order to remove this imprecision, we can derive the use-definition chain for the branch target register, and the target addresses are given by the right hand side of the definitions.

We also encode some instructions of ARM ISA (only for this example) using the iSAL, and VSA can also resolve the indirect branch instructions, as shown in Fig. 3.2. From Fig. 3.2, we can see the data for branch target addresses is put into the text section under ARM architecture, which is a challenge as stated in Chapter 2. However, storing the jump table in the text section saves some computation related to jump table address calculation (like in the case of MIPS code, we need to calculate where the jump table is located before loading the target address into v0). Therefore, at the address **0x815c**, PC relative addressing is used

to get the jump table. Since in ARM we can directly manipulate the PC register, the target address is loaded into the PC register directly instead of "branch to a register" like MIPS does.

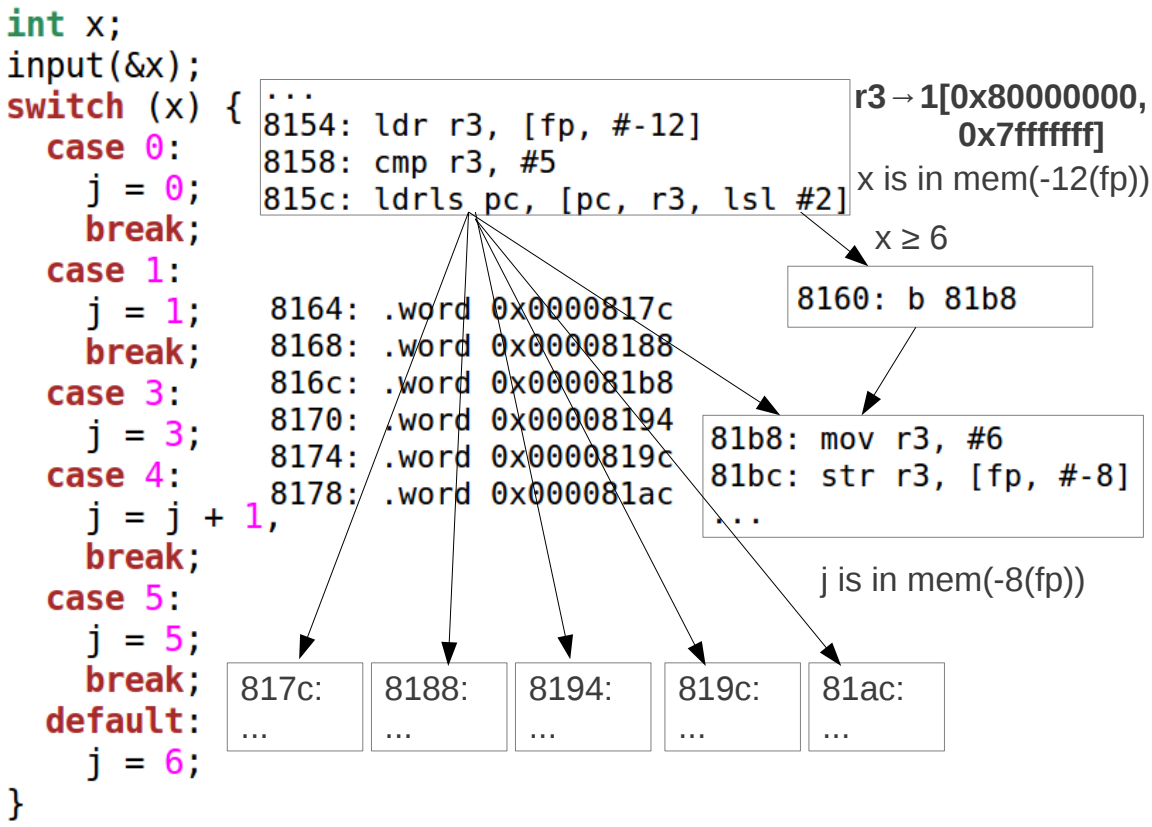


Figure 3.2: A C code snippet with *switch* statements and compiled ARM code

3.5 Conclusion and Future Work

In this chapter, we extend the original strided-interval domain to more precisely track the set of structured numbers, and also define the operations on this extended strided-interval domain. We present the syntax and concrete semantics of the iSAL, which can be used to encode the instructions of different ISAs. In order to achieve generic VSA, we define the abstract semantics for the intermediate language, and discuss how to use it in VSA. We also show an example on using the approach to reconstruct the CFG in the

presence of indirect branches.

In the future, we want to try the approach on more architectures, and also want to extend iSAL with more static analysis methods. Moreover, we want to try to analyze some control software generated from Simulink or other model-based design tools.

Chapter 4

Improving the Precision of Abstract Interpretation Based Cache Persistence Analysis

When designing hard real-time embedded systems, we need to perform schedulability analysis to guarantee the stringent timing constraints will be met. Schedulability analysis needs the worst-case execution time (WCET) of each real-time task as input. Therefore, WCET analysis is one essential step in designing such systems, and has been studied extensively (see [41] for a survey). In general, the exact WCET of a task is impossible to derive. Thus, when estimating WCET, over-approximation is necessary to guarantee safety. However, in order to maximize the resource utilization, an WCET estimation should be as tight as possible.

Due to the big timing gap between a cache hit and a miss, cache behavior can affect the execution time significantly. In order to derive a tight WCET estimation, we want the cache behavior analysis to be as precise as possible. Although model checking-based cache analysis can yield precise results since all the possible executions are examined, potential state space explosion makes it hard to use in practice [91]. Compared to model checking, cache analysis methods based on abstract interpretation may lose some precision but can achieve much better scalability. In this chapter, we focus on how to improve the precision of cache analysis that is based on abstract interpretation [9].

As described in Chapter 2, when predicting the cache behavior, a widely used method is to classify the memory references as – *AH* (it *always hits* the cache), *AM* (it *always misses* the cache), *PS* (it is *persistent* if the memory reference may result in a cache hit/miss for the first time but it hits the cache subsequently), and *NC* (it is *not classified* if the memory reference is not classified as *AH*, *AM*, or *PS*). These classifications are derived by performing three different analyses, **must**, **may**, and **persistence** analyses, on the control flow graph (CFG) [85]. While the must and may analyses are safe, it has been known that the original

persistence analysis method proposed in [117] is unsafe. Several approaches have been proposed to ensure safe cache persistence analysis [73, 74, 76]. However, different approaches may suffer from certain pessimism under different scenarios (i.e. some references should have been classified as *PS* if the analysis were precise).

In this chapter, we first analyze the sources of pessimism of safe cache persistence analysis of single-level loops. Then, we propose two methods to eliminate these sources of pessimism. We focus on persistence analysis of A-way set associative instruction caches which use LRU (least recently used) replacement policy. However, the methods can be easily extended to data/unified cache persistence analysis.

The main technical contributions of this chapter include: (1) We identify the sources of pessimism that two recent state-of-the-art persistence analysis methods may encounter: The method proposed in [73] has a pessimistic join function, and the method proposed in [74] has a pessimistic update function; (2) We optimize the update function proposed in [74] by finding a safe limit that bounds the range of the blocks whose potential maximal ages should be increased in an updating process; (3) We integrate the improved method of [74] and the method proposed in [73] to further safely reduce pessimism but the integration may have a large storage overhead. By studying the relations of these two approaches, we define two auxiliary functions to reduce this overhead in the integration; (4) We prove the proposed approaches are safe, namely if a memory reference at a program point is classified as *PS*, the memory block it accesses is not possibly evicted from the cache at this point after being loaded; (5) We demonstrate the number of memory references classified as *PS* can be increased by using the proposed methods from the experimental evaluations performed on a set of benchmarks. We also empirically compare the storage space and analysis time used by different methods. This work has been published in [11].

The rest of the chapter is organized as follows: Section 4.1 briefly summarizes two recent state-of-the-art safe approaches for cache persistence analysis; Section 4.2 compares these two approaches in terms of their sources of pessimism; Section 4.3 improves the

update function for the approach based on may analysis, and proves such an improvement is safe; Section 4.4 proposes an integration of the two existing approaches; Section 4.5 presents the evaluation and Section 4.6 concludes this chapter.

4.1 Background

We first present the objective of cache persistence analysis, and then briefly describe two recent state-of-the-art safe approaches, namely the approach based on younger set which is proposed in [73] and the approach based on may analysis which is proposed in [74].

We model an A-way set associative cache as a sequence of v cache sets $F = \langle f_1, f_2, \ldots, f_v \rangle$. Each cache set is an independent fully associative cache and is modeled by a sequence of A cache blocks $L = \langle l_1, l_2, \ldots, l_A \rangle$. Since the behaviors of the cache sets are independent of each other, we can focus on one cache set for the sake of readability. The memory consists of a set of w memory blocks $M = \{m_1, m_2, \ldots, m_w\}$, and the program has t program points $P = \{p_1, p_2, \ldots, p_t\}$.

4.1.1 Cache Persistence Analysis

Cache persistence analysis aims to categorize a memory reference that cannot be classified as *AH* at a program point in a loop as *PS*, if its accessed memory block stays in the cache after the first time this block is loaded. If a memory reference is categorized as *PS*, it can result in at most one cache miss. In the case of a loop bounded by n iterations, a reference classified as *PS* instead of *NC* can reduce the number of possible misses by $n-1$. Thus, we want to safely classify as many references as possible as *PS* for a loop.

In order to guarantee safety, we need to over-approximate a memory block's maximal age at every program point. If a memory block is not among the set of possibly evicted memory blocks, any reference to it can be treated as *PS*. In order to keep track of possibly evicted memory blocks, an additional cache block l_{A+1} is appended to L. If a memory

block's potential maximal age is greater than the cache's associativity A, it will be added into this additional cache block. Let $\top \equiv A+1$, so we have $L' = \langle l_1, \ldots, l_A, l_\top \rangle$ model a cache set to capture persistent behavior. Therefore, in cache persistence analysis, an abstract set state $\hat{s}_{pers}$ is often modeled as $\hat{s}_{pers} \in D_{\mathscr{P}} = L' \to 2^M$, and $\hat{s}_{pers}(l_\top)$ gives the over-approximated set of memory blocks that are possibly evicted after being loaded into this cache set.

In order to improve the precision, we want to tighten the over-approximation of a memory block's maximal age. Therefore, we want to eliminate possible sources of pessimism in the analysis to keep $l_\top$ from containing too many persistent memory blocks.

4.1.2 Cache Persistence Analysis Based on Younger Set

The basic idea of the approach based on younger set (*YS-Pers*) is to keep track of all the memory blocks that may be younger than a memory block for that block. Thus, a memory reference can be categorized as *PS* if the cardinality of the accessed memory block's younger set is less than A.

Let $ys^p(m)$ denote the younger set of a memory block m at a program point p, and let $YS = M \to (2^M)_\perp$ denote the set of all the younger set mappings, i.e. we have $ys^p \in YS$. Since the ys^p may be a partial function, namely there may be no younger set for some memory block at some program point, we use the lifted co-domain $(2^M)_\perp = 2^M \cup \{\perp\}$, where $\perp$ means "no younger set at all". As defined in [73], $ys^p(m)$ is *a superset of all the memory blocks that may have smaller relative ages (younger) than m at p in some possible program execution that reaches p*. The potential maximal age of m can be calculated as $|ys^p(m)|+1$ which is in the range $[1 \ldots \top]$, assuming we stop tracking when $|ys(m)|$ reaches A.

Therefore, given the younger set mapping ys^p at a program point p, the i^{th} cache set's abstract set state $\hat{s}^{p,i}_{pers}$ is actually derived from ys^p by applying the function $G_{\mathscr{P}} : YS \times$

$\{1,\cdots,v\} \to D_{\mathscr{P}}$ (i.e. $\hat{s}^{p,i}_{pers} = G_{\mathscr{P}}(ys^p, i)$), and the $G_{\mathscr{P}}$ function is defined as:

$$G_{\mathscr{P}}(ys,i) := [l_x \mapsto \{m | set(m) = i \wedge ys(m) \neq \bot \wedge x = |ys(m)| + 1\} \text{ with } x = 1,\cdots,A]^1$$

where $[\delta \mapsto \theta]$ denotes a function that maps δ to θ and $set(m)$ gives the cache set number which m is mapped to.

If a memory block m' is going to be accessed at a program point p', which is immediately following a program point p, the younger set mapping $ys^{p'}$ can be calculated by performing the younger set mapping update function $\hat{U}_{\mathscr{Y}\mathscr{S}} : YS \times M \to YS$ on ys^p to take into account the effect of the reference to m' (i.e. $ys^{p'} = \hat{U}_{\mathscr{Y}\mathscr{S}}(ys^p, m')$), and the $\hat{U}_{\mathscr{Y}\mathscr{S}}$ is defined as:

$$\hat{U}_{\mathscr{Y}\mathscr{S}}(ys,m') := [m \mapsto \begin{cases} ys(m) & \text{if } set(m') \neq set(m) \\ ys(m) \cup \{m'\} & \text{else if } m' \neq m \\ \emptyset & \text{otherwise} \end{cases}]$$

If a program point p is a join point of two points $p1$ and $p2$ at which the younger set mappings are ys^{p1} and ys^{p2} respectively, the joined younger set mapping ys^p can be calculated by applying the younger set mapping join function $\hat{J}_{\mathscr{Y}\mathscr{S}} : YS \times YS \to YS$ (i.e. $ys^p = \hat{J}_{\mathscr{Y}\mathscr{S}}(ys^{p1}, ys^{p2})$), and the $\hat{J}_{\mathscr{Y}\mathscr{S}}$ is defined as:

$$\hat{J}_{\mathscr{Y}\mathscr{S}}(ys^{p1}, ys^{p2}) := [m \mapsto \begin{cases} ys^{p1}(m) \overline{\cup} ys^{p2}(m) & \text{if } ys^{p1}(m) \neq \bot \wedge ys^{p2}(m) \neq \bot \\ ys^{p1}(m) & \text{else if } ys^{p1}(m) \neq \bot \\ ys^{p2}(m) & \text{else if } ys^{p2}(m) \neq \bot \\ \bot & \text{otherwise} \end{cases}]$$

[1] If we do not stop tracking new potentially younger blocks when $|ys(m)|$ reaches A, we would have $x = \min(|ys(m)| + 1, \top)$.

where $\overline{\cup}$ is a special set union operation which may truncate some memory blocks in the union at random to make the cardinality of the union at most A. For a persistent memory block m, the resulted younger set $ys^p(m)$ always contains all the potentially younger blocks of m (in the case that m is not persistent, namely it is possibly evicted, some of its younger blocks may be truncated, but it does not affect m will be placed in the corresponding $l_\top$).

4.1.3 Cache Persistence Analysis Based on May Analysis

The approach based on may analysis (*May-Pers*) utilizes the over-approximation of cache contents generated by a parallel running may analysis to guide the maximal age updating. Basically, *May-Pers* is a combination of two analyses: (1) the *may-part* analysis (whose join and update functions are $\hat{J}_{\mathscr{M}}$ and $\hat{U}_{\mathscr{M}}$ respectively) is the traditional may analysis and it is used to provide the other analysis with an over-approximation of cache contents; and (2) the *persistence-part* analysis (whose join and update functions are $\hat{J}_{\mathscr{Q}}$ and $\hat{U}_{\mathscr{Q}}$ respectively) is a modification of the traditional may analysis which tracks the maximal age of a memory block instead of the minimal age [74, 76]. The abstract set state domain used in this approach is

$$D_{\mathscr{P}}^{\text{may-pers}} = D_{\mathscr{M}} \times D_{\mathscr{P}} = (L \to 2^M) \times (L' \to 2^M)$$

where $D_{\mathscr{M}} = L \to 2^M$ is the abstract set state domain for the traditional may analysis and $D_{\mathscr{P}} = L' \to 2^M$ is the abstract set state domain for the original persistence analysis. Thus, an abstract set state is a 2-tuple $\langle \hat{s}_{may}, \hat{s}_{pers} \rangle$, a *may-part* $\hat{s}_{may}$ and a *persistence-part* $\hat{s}_{pers}$ respectively. While the parallel running *may-part* analysis is independent from the *persistence-part* analysis, when the *persistence-part* analysis updates $\hat{s}_{pers}$ it has to take into account $\hat{s}_{may}$.

The update function $\hat{U}_{\mathscr{P}} : D_{\mathscr{P}}^{\text{may-pers}} \times M \to D_{\mathscr{P}}^{\text{may-pers}}$ for the *May-Pers* is defined as:

$$\hat{U}_{\mathscr{P}}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) := \langle \hat{U}_{\mathscr{M}}(\hat{s}_{may}, m), \hat{U}_{\mathscr{Q}}(\hat{s}_{may}, \hat{s}_{pers}, m) \rangle$$

where $\hat{U}_{\mathscr{M}}$ is the well-defined update function for the may analysis (whose definition can be found in [85]), and $\hat{U}_{\mathscr{Q}} : D_{\mathscr{M}} \times D_{\mathscr{P}} \times M \to D_{\mathscr{P}}$ is the update function for the *persistence-part* analysis, which is defined as:

$$\hat{U}_{\mathscr{Q}}(\hat{s}_{may}, \hat{s}_{pers}, m) := \begin{cases} [l_1 \mapsto \{m\}, & \\ \quad l_i \mapsto \hat{s}_{pers}(l_{i-1}) \backslash \{m\} | i = 2 \ldots A, & \\ \quad l_\top \mapsto (\hat{s}_{pers}(l_A) \cup \hat{s}_{pers}(l_\top)) \backslash \{m\}] & \text{if } \mathit{mayevict}(\hat{s}_{may}, m) \\ [l_1 \mapsto \{m\}, & \\ \quad l_i \mapsto \hat{s}_{pers}(l_{i-1}) \backslash \{m\} | i = 2 \ldots A-1, & \\ \quad l_A \mapsto (\hat{s}_{pers}(l_A) \cup \hat{s}_{pers}(l_{A-1})) \backslash \{m\}, & \\ \quad l_\top \mapsto \hat{s}_{pers}(l_\top) \backslash \{m\}] & \text{otherwise} \end{cases}$$

$$\mathit{mayevict}(\hat{s}_{may}, m) := |\{m' | m' \neq m \wedge m' \in \hat{s}_{may}\}| \geq A$$

Basically, $\mathit{mayevict}(\hat{s}_{may}, m)$ checks whether the overestimated contents given by $\hat{s}_{may}$ have potentially filled the cache set or not. If the *mayevict* function returns true, the abstract set state $\hat{s}_{may}$ of the may analysis contains at least A many other memory blocks than m. In this case, the cache set may be completely filled already without counting m, so an access to m potentially increase the maximal ages of all the memory blocks in $\hat{s}_{may}$ and may cause some blocks evicted (as shown in the first case of the update function). On the contrary, if the *mayevict* function returns false, the cache set is definitely not full yet, so no eviction will happen due to loading m. In this case, the maximal ages of all memory blocks will not

exceed A (as shown in the second case of the update function).

The join function $\hat{J}_{\mathscr{P}} : D_{\mathscr{P}}^{\text{may-pers}} \times D_{\mathscr{P}}^{\text{may-pers}} \to D_{\mathscr{P}}^{\text{may-pers}}$ for the *May-Pers* is defined as:

$$\hat{J}_{\mathscr{P}}(\langle \hat{s}_{may}^{p1}, \hat{s}_{pers}^{p1} \rangle, \langle \hat{s}_{may}^{p2}, \hat{s}_{pers}^{p2} \rangle) := \langle \hat{J}_{\mathscr{M}}(\hat{s}_{may}^{p1}, \hat{s}_{may}^{p2}), \hat{J}_{\mathscr{Q}}(\hat{s}_{pers}^{p1}, \hat{s}_{pers}^{p2}) \rangle$$

where the $\hat{J}_{\mathscr{M}}$ function is the well-defined join function for the may analysis (whose definition can be found in [85]), and $\hat{J}_{\mathscr{Q}} : D_{\mathscr{P}} \times D_{\mathscr{P}} \to D_{\mathscr{P}}$ is defined as:

$$\begin{aligned} \hat{J}_{\mathscr{Q}}(\hat{s}_{pers}^{p1}, \hat{s}_{pers}^{p2}) := [l_i \mapsto\ & \{m | m \in \hat{s}_{pers}^{p1}(l_i) \wedge\ \nexists b \in [1 \ldots \top] : m \in \hat{s}_{pers}^{p2}(l_b)\} \cup \\ & \{m | m \in \hat{s}_{pers}^{p2}(l_i) \wedge\ \nexists a \in [1 \ldots \top] : m \in \hat{s}_{pers}^{p1}(l_a)\} \cup \\ & \{m | \exists a, b \in [1 \ldots \top] : m \in \hat{s}_{pers}^{p1}(l_a) \wedge m \in \hat{s}_{pers}^{p2}(l_b) \wedge i = \max(a, b)\}] \end{aligned}$$

The $\hat{J}_{\mathscr{Q}}$ function is much similar to the join function of the original persistence analysis, which is similar to set union operation except that if a memory block has two different ages in the two joining set states then the function takes the oldest one.

4.2 Sources of Pessimism

There have been several approaches proposed to safely analyze cache persistence. However, there has been little work done to compare and find out whether these safe approaches are precise enough under different circumstances, and to improve their precision for a single-level loop. Although the advantages and disadvantages of the approaches based on may analysis and conflict counting are discussed in [76], that paper does not compare them with the approach based on younger set that is proposed in [73].

Since we know that the approach based on conflict counting is not as precise as the one based on may analysis (due to the loss of age information), we concentrate on the comparisons between the approaches based on younger set and may analysis – we discuss under what circumstances an approach may give pessimistic analysis results and show how the approaches can complement each other.

In order to enhance the readability of examples, we assume a 2-way set associative cache is used. Memory blocks m_a, m_b, and m_c are mapped into the same cache set that we focus on. In Fig. 4.1, a basic block with a memory block shown inside (e.g. BB_1 in the figure has m_a shown inside) denotes the basic block contains an instruction which references to the corresponding memory block; otherwise, the basic block (e.g. BB_2 in the figure has no relationship with the cache set we are examining.

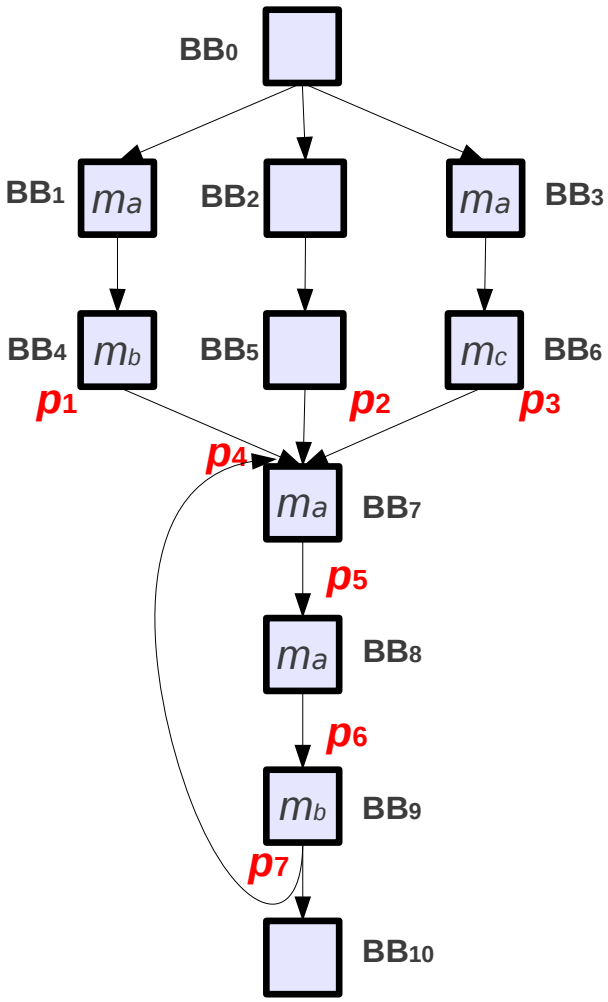


Figure 4.1: The CFG of a program: all of the references in the loop should be classified as *PS*

4.2.1 Pessimism in *YS-Pers*

The persistence analysis based on younger set is safe, but it may excessively overestimate the maximal age of a memory block m at a join point p_j, since the join function $\hat{J}_{\mathscr{Y}\mathscr{S}}$ uses the concept of set union (as mentioned in section 4.1.2, the $\overline{\cup}$ operation is used) to ensure all the possibly younger memory blocks on all the joined paths are captured for m, namely

$$ys^{p_j}(m) = \hat{J}_{\mathscr{Y}\mathscr{S}}(\ldots, \hat{J}_{\mathscr{Y}\mathscr{S}}(ys^{p_{i_1}}(m), ys^{p_{i_2}}(m)), \ldots, ys^{p_{i_n}}(m))$$

where $\{p_{i_1}, p_{i_2}, \ldots, p_{i_n}\}$ is the set of the exit points of p_j's n predecessors denoted as $pred(p_j)$. Therefore, this ***may*** introduce some pessimism if $\exists p_{i_x}, p_{i_y} \in pred(p_j) : ys^{p_{i_x}}(m) \neq ys^{p_{i_y}}(m)$, especially when disjoint sets of memory blocks are accessed in the disjoint parts of paths reaching p_{i_x} and p_{i_y}.

Consider the program point p_4 in Fig. 4.1 which is a join point with four predecessors (i.e. BB_4, BB_5, BB_6, and BB_9). Although the memory reference to m_a in BB_7 cannot be classified as *AH* due to the possible path $BB_0 \rightarrow BB_2 \rightarrow BB_5 \rightarrow BB_7$, we can easily observe the reference should be classified as *PS*, in which case, this memory reference contributes at most one cache miss to the loop independent of the number of its iterations.

However, when using *YS-Pers* to perform persistence analysis, we observe that at the exit point of BB_7's each predecessor (i.e. the program points p_1, p_2, p_3, and p_7) the m_a's younger set is as follows:

$$ys^{p_1}(m_a) = \{m_b\} \quad ys^{p_2}(m_a) = \emptyset$$
$$ys^{p_3}(m_a) = \{m_c\} \quad ys^{p_7}(m_a) = \{m_b\}$$

Since the join function of the younger set is based on the $\overline{\cup}$ operation, at p_4 the younger set of m_a is always:

$$ys^{p_4}(m_a) = \overline{\bigcup_{p \in \{p_1, p_2, p_3, p_7\}}} ys^{p}(m_a) = \{m_b, m_c\}$$

Given the cache associativity is 2, it means before the memory reference to m_a in BB_7, m_a always has the age $\top$, which prevents us from classifying the reference as *PS*.

4.2.2 Pessimism in *May-Pers*

Compared to *YS-Pers*, *May-Pers* can precisely classify the memory reference to m_a in BB_7 as *PS*. Although the approach does not suffer from pessimism when joining the states, it does not mean the approach will always yield more precise analysis. Actually, one apparent source of pessimism in this approach comes from its pessimistic update function, which we will try to optimize in the next section. In order to ensure safety, when accessing a memory block m, the *persistence-part* update function $\hat{U}_{\mathscr{Q}}$ proposed in [74] increases the potential maximal ages of all memory blocks if the abstract set state $\hat{s}_{may}$ of the parallel running *may-part* analysis contains at least cache's associativity A many other elements than m, namely when the $mayevict(\hat{s}_{may}, m)$ is true, as described in section 4.1.3.

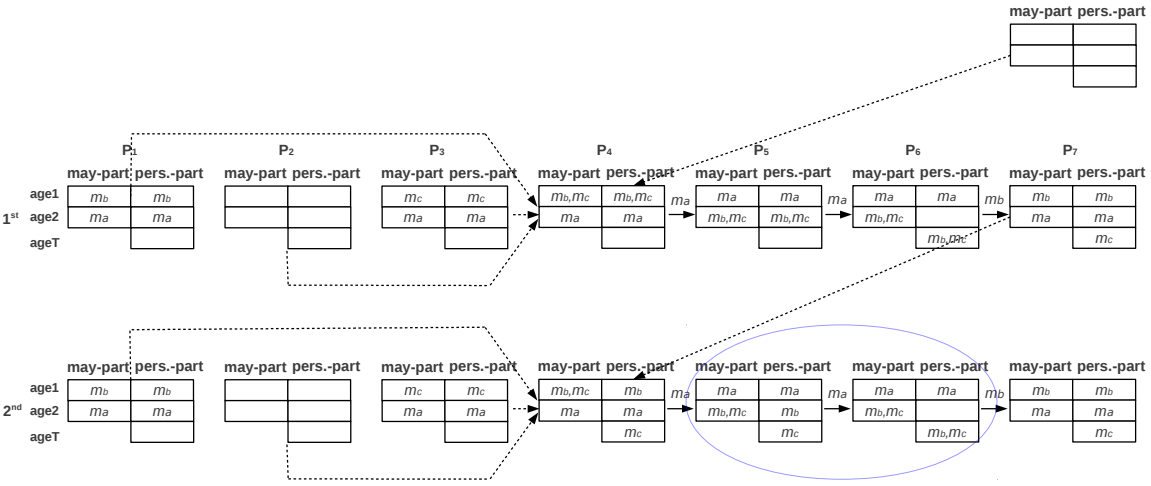


Figure 4.2: Abstract set states of Fig. 4.1 computed by the may analysis based approach

In order to help understand the source of pessimism in *May-Pers*, Fig. 4.2 shows the iterative process of deriving the fixed points of the abstract set states corresponding to the seven program points shown in Fig. 4.1. The dotted transition lines in Fig. 4.2 are related to the join function, and the solid transition lines are related to the update function. The

initial set state at each program point is an empty $\langle \hat{s}_{may}, \hat{s}_{pers} \rangle$. The figure shows in the second iteration every abstract set state reaches its fixed point (and the third iteration is omitted, which only verifies each abstract set state has reached its fixed point).

Consider the program point p_5 in Fig. 4.1. Since there is a memory reference to m_a in BB_8, the execution of BB_8 needs to update the corresponding abstract set state. When performing the update function which is piecewise, we need to consider how many memory blocks other than m_a are in the *may-part* of the input abstract set state. In our case, as shown in the circled part of Fig. 4.2, the number of other memory blocks is $|\{m_b, m_c\}| = 2$ equal to the cache's associativity, which means the cache set may be already full and the memory reference to m_a may cause an eviction. As a consequence, the update function will increase the potential maximal ages of all memory blocks in the *persistence-part* of the abstract set state and reset m_a to be the youngest, as shown in the circled abstract set state at p_6.

From the resultant abstract set state at p_6, we can find that the memory reference to m_b in BB_9 cannot be classified as *PS* (since its new age is $\top$). However, we can easily observe that it is indeed persistent in the loop independent of the number of its iterations. Again, the precision is reduced while the analysis is safe.

On the contrary, the approach based on younger set can give us the *PS* classification for the reference to m_b in BB_9, since at p_6 the younger set of m_b is $ys^{p_6} = \{m_a\}$ which contains one possibly younger memory block. Therefore, as a comparison, we can observe the *YS-Pers* approach may introduce some pessimism due to joining different younger sets while the *May-Pers* approach may introduce some pessimism due to updating the abstract set state.

4.2.3 Overview of the Proposed Approaches

Our problem is how to eliminate the sources of pessimism described above to improve the *PS* classification precision while keeping the analysis still safe. In order to solve this problem, we propose two approaches in the next two sections: The first one is based on

the *May-Pers* approach but improves its updating strategy (see Section 4.3); and the second one is an integration of the improved *May-Pers* and *YS-Pers* approaches (see Section 4.4).

In Fig. 4.3, the relationship between the various approaches is illustrated by a Venn diagram. The whole area represents all of the memory references that are persistent in the program. As shown in the figure, the approaches based on may analysis (i.e. the original *May-Pers* and the improved *May-Pers*) can classify some references as *PS* while the approach based on younger set cannot; and vice versa (as we have discussed above). However, we can see that the amount of memory references classified as *PS* by the integration approach dominates that by the others. In Section 4.5, this relationship is validated empirically by experiments.

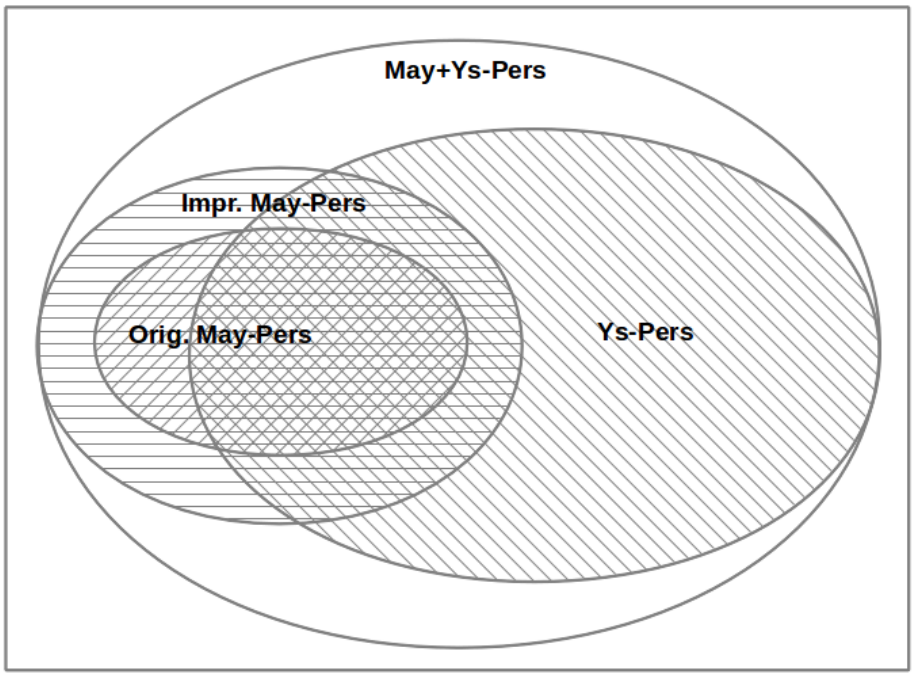


Figure 4.3: Venn diagram illustrating the relationship between different approaches

4.3 More Precise Update Function for *May-Pers*

As discussed in 4.2.2, although the *persistence-part* update function $\hat{U}_{\mathscr{Q}}$ is safe, it is not precise enough – it makes m_b be treated as possibly evicted while in reality m_b definitely

stays in the loop after the first time being loaded. We improve the $\hat{U}_{\mathscr{Q}}$ function by using a new strategy to decide where the process of maximal age updating should end, while keeping the join function of the analysis unchanged.

The new persistence analysis update function $\hat{U}^{\text{new}}_{\mathscr{P}} : D^{\text{may-pers}}_{\mathscr{P}} \times M \to D^{\text{may-pers}}_{\mathscr{P}}$ is based on the well-defined update function for the may analysis (i.e the $\hat{U}_{\mathscr{M}}$ function) and the improved $\hat{U}_{\mathscr{Q}}$ function (i.e. the $\hat{U}^{\text{new}}_{\mathscr{Q}}$ function), and the $\hat{U}^{\text{new}}_{\mathscr{P}}$ function is defined as:

$$\hat{U}^{\text{new}}_{\mathscr{P}}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) := \langle \hat{U}_{\mathscr{M}}(\hat{s}_{may}, m), \hat{U}^{\text{new}}_{\mathscr{Q}}(\hat{s}_{pers}, m, H(\hat{s}_{may}, m)) \rangle$$

The $\hat{U}^{\text{new}}_{\mathscr{P}}$ function is much similar to the $\hat{U}_{\mathscr{P}}$ function described in Section 4.1.3, except the update function for the *persistence-part* (i.e. the $\hat{U}^{\text{new}}_{\mathscr{Q}}$ function instead of the $\hat{U}_{\mathscr{Q}}$ function). In the $\hat{U}^{\text{new}}_{\mathscr{Q}}$ function, we use a $H : D_{\mathscr{M}} \times M \to \{1, \cdots, A, \top\}$ function to select a relative age h which bounds the possibly affected age range due to the memory reference to m. The H function is defined as:

$$H(\hat{s}_{may}, m) := \min(\{y | 1 \leq y \leq A \wedge \sum_{1 \leq i \leq y} |\hat{s}_{may}(l_i) \backslash \{m\}| < y\} \cup \{\top\})$$

The H function uses the over-approximation of cache contents in $\hat{s}_{may}$ to try to find the youngest y which is smaller than $\top$ and strictly bigger than the total number of the memory blocks whose relative ages are possibly younger than this y; if it cannot find a single y, the age $\top$ is used. Note that if such a y exists, it means in any concrete cache set state, without considering m, the cache blocks $\langle l_1, \ldots, l_y \rangle$ are not full of memory blocks yet, since each age position of $\hat{s}_{may}$ contains all the memory blocks that are possibly in that position. Thus, a memory reference to m leads to m having the youngest relative age and there is no memory block being possibly evicted from the $\langle l_1, \ldots, l_y \rangle$ region. Therefore, any memory block's potential maximal age which is already beyond y will not be increased. In order to gain more precision, the H function returns the smallest y, since there may exist more than one y when the cache set is not full. If such a y does not exist, the cache set

is possibly full, so every memory block's potential maximal age should be increased and the H function returns $\top$. Based on this argument, we can have the safe but more precise updated function $\hat{U}^{\text{new}}_{\mathscr{Q}}$ defined as:

$$\hat{U}^{\text{new}}_{\mathscr{Q}}(\hat{s}_{pers}, m, h) := \begin{cases} [l_1 \mapsto \{m\}, & \\ l_2 \mapsto (\hat{s}_{pers}(l_1) \cup \hat{s}_{pers}(l_2)) \backslash \{m\}, & \\ l_i \mapsto \hat{s}_{pers}(l_i) \backslash \{m\} | i = 3 \ldots \top] & \text{if } h = 1 \\ & \\ [l_1 \mapsto \{m\}, & \\ l_i \mapsto \hat{s}_{pers}(l_{i-1}) \backslash \{m\} | i = 1 \ldots h-1, & \\ l_h \mapsto (\hat{s}_{pers}(l_h) \cup \hat{s}_{pers}(l_{h-1})) \backslash \{m\}, & \\ l_i \mapsto \hat{s}_{pers}(l_i) \backslash \{m\} | i = h+1 \ldots \top] & \text{else if } 1 < h \leq A \\ & \\ [l_1 \mapsto \{m\}, & \\ l_i \mapsto \hat{s}_{pers}(l_{i-1}) \backslash \{m\} | i = 2 \ldots A, & \\ l_\top \mapsto (\hat{s}_{pers}(l_A) \cup \hat{s}_{pers}(l_\top)) \backslash \{m\}] & \text{otherwise} \end{cases}$$

From the $\hat{U}^{\text{new}}_{\mathscr{Q}}$ function we can see if the value of $H(\hat{s}_{may}, m)$ (i.e. h) is not $\top$, we do not need to pessimistically increase the maximal ages of all memory blocks like the one does in the original approach (i.e. the $\hat{U}_{\mathscr{Q}}$ function), even if $\hat{s}_{may}$ contains more than or equal to A many other elements than m.

For example, consider Fig. 4.1 again. When we need to update the corresponding abstract set state at p_5 as shown in Fig. 4.4 (which is the same as the circled one in Fig.

4.2) due to the memory reference to m_a in BB_8, $H(\hat{s}^{p_5}_{may}, m_a)$ gives $h = 1$:

$$\text{when } y = 1,\ |\hat{s}^{p_5}_{may}(l_1) \backslash \{m_a\}| = |\emptyset| = 0 < y$$

$$\text{when } y = 2,\ \sum_{i=1}^{2} |\hat{s}^{p_5}_{may}(l_i) \backslash \{m_a\}| = |\emptyset| + |\{m_b, m_c\}| = 2 \geq y$$

$$\text{therefore } h = H(\hat{s}^{p_5}_{may}, m_a) = \min(\{1\} \cup \{\top\}) = 1$$

which does not alter the state at all since $\hat{U}^{\text{new}}_{\mathscr{Q}}(\hat{s}^{p_5}_{pers}, m_a, 1)$ applies the first case of the $\hat{U}^{\text{new}}_{\mathscr{Q}}$ function. Thus, the reference to m_b in BB_9 can be safely classified as *PS*.

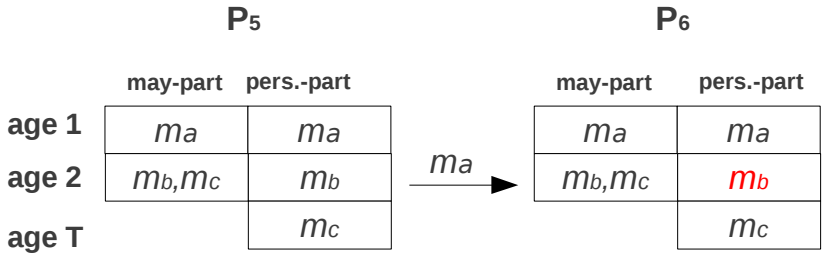


Figure 4.4: Updating abstract set state at p_5 more precisely

In the following, we use *Orig. May-Pers* to represent the *May-Pers* using the original update function $\hat{U}_{\mathscr{P}}$, and use *Impr. May-Pers* to represent the *May-Pers* using our improved update function $\hat{U}^{\text{new}}_{\mathscr{P}}$.

Theorem 4.3.1. *The Impr. May-Pers approach is safe, namely at a program point p, any memory block that is loaded into the cache is in an age position of* $\hat{s}^{p}_{pers}$ *and this age is greater than or equal to the possible maximal age of the block when the execution reaches p (which implies if this block is possibly evicted from the cache, it is in the* $\top$ *position of* $\hat{s}^{p}_{pers}$*).*

Proof. The well-developed cache may analysis is safe [118, 119]. Since in *Impr. May-Pers* the may analysis is parallel running independently, its soundness ensures that at a program point p, any memory block that is possibly in the cache is in an age position of $\hat{s}^{p}_{may}$ and

this age is smaller than or equal to the possible minimal age of this block. We use this fact to prove this theorem holds at any program point by mathematical induction.

Base case: At the beginning of any execution, we have a cold start such that no memory block is loaded; and we also have an empty $\hat{s}_{pers}$. Therefore, this theorem holds at the beginning.

Inductive hypothesis: At any program point which is immediately before a program point p, this theorem holds.

Inductive step: The program point p can be either a point inside a basic block or a join point of different control flows. We need to prove in either case this theorem holds at p.

- Case 1: the program point p is a point inside a basic block. In this case, p only has one immediately previous program point, say p'. Let us assume a memory block m is accessed at p. Thus, $\langle \hat{s}^p_{may}, \hat{s}^p_{pers} \rangle$ is $\langle \hat{U}_{\mathscr{M}}(\hat{s}^{p'}_{may}, m), \hat{U}^{\text{new}}_{\mathscr{Q}}(\hat{s}^{p'}_{pers}, m, H(\hat{s}^{p'}_{may}, m)) \rangle$. According to the discussion above, we know $\hat{s}^{p'}_{may}$ and $\hat{s}^p_{may}$ contains the over-approximated contents at the program point p' and p respectively. Therefore, with the over-approximated contents in $\hat{s}^{p'}_{may}$, according to the rationale of the H function, we know that $H(\hat{s}^{p'}_{may}, m)$ finds a position y which is the upper bound of a region $\langle l_1, \ldots, l_y \rangle$ such that no memory block will be evicted from this region due to m entering the region (note that if y is $\top$, this argument is still valid since no block will be removed from the region $\langle l_1, \ldots, l_\top \rangle$). According to the inductive hypothesis, we know the theorem holds at p', from which we can deduce any block in a position l_x has its possible maximal age at most x. Since $\hat{U}^{\text{new}}_{\mathscr{Q}}$ increase the position of a memory block except for m (whose age becomes the youngest) in the region $\langle l_1, \ldots, l_y \rangle$, we can deduce any block is in a position of $\hat{s}^p_{pers}$ which is at least its possible maximal age, namely this theorem holds at p.

- Case 2: the program point p is a join point of the exit points of $i \geq 1$ basic blocks, say these exit points are $p'_1, \cdots, p'_i$. According to the inductive hypothesis, we know

the theorem holds at $p'_1, \cdots, p'_i$, namely when the execution reaches either one of $p'_1, \cdots, p'_i$, say p', the possible maximal age of any loaded memory block m is at most x given $m \in \hat{s}^{p'}_{pers}(l_x)$. Since no memory block is accessed at a join point and the join function $\hat{J}_{\mathscr{P}}$ uses the maximum relative age of a memory block in $\hat{s}^{p'_1}_{pers}, \cdots, \hat{s}^{p'_i}_{pers}$, we can easily deduce this theorem holds at the join point p.

Combining Case 1 and Case 2, we can see this theorem holds at the program point p. Therefore, we conclude this theorem holds at any program point. □

4.4 Integration of the Two Approaches

Impr. May-Pers can precisely classify both the memory references to m_a in BB_7 and to m_b in BB_9 in Fig. 4.1 as *PS*, which cannot be achieved neither by *YS-Pers* nor by *Orig. May-Pers*. However, *Impr. May-Pers* may become imprecise when the overestimated cache contents becomes more conservative at a join point corresponding to a loop head (since may analysis does not distinguish different iterations in a loop).

Consider the program whose CFG is shown in Fig. 4.5. We can easily see the memory reference to m_b in BB_5 should be classified as *PS*, since it is not possible to be evicted once it is loaded into the cache. However, as we can observe from Fig. 4.6 which shows the process of deriving the fixed points of the abstract set states at the five program points marked in Fig. 4.5, the reference to m_b in BB_5 cannot be classified as *PS*, since from the fixed point of the abstract set state at p_4 we can observe m_b is among the possibly evicted memory blocks before the reference.

The reason for this pessimism is that at the join point p_2 may analysis merges the information of different iterations to form an over-approximation of cache contents for each age position. If using the younger set generation technique as described in section 4.4.2, m_a can even be conservatively treated as younger than m_b, which is not possible in reality. Therefore, using this $\hat{s}_{may}$, it becomes harder for $H(\hat{s}_{may}, m_c)$ to find a position better than $\top$, which leads to the state with m_b being treated as possibly evicted.

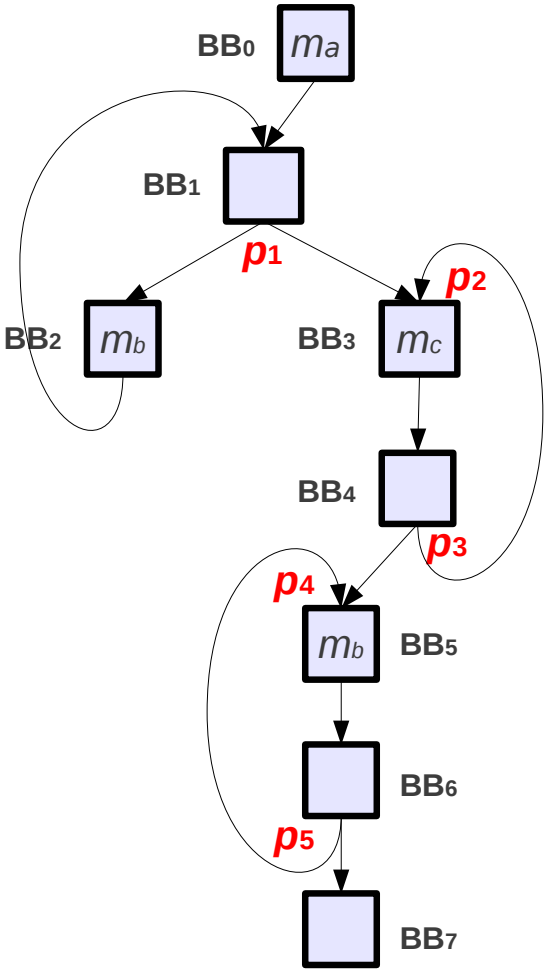


Figure 4.5: The CFG of another program: all of the references in the loops should be classified as *PS*

Loop unrolling can eliminate this pessimism but with a very large overhead [120]. Fortunately, we can observe that *YS-Pers* is immune to this pessimism ($ys^{p_2}(m_b) = \{m_c\}$) since the younger block information is combined but cannot be collapsed at join point (m_a can never be younger than m_b and m_c). Thus, we want to integrate *YS-Pers* and *May-Pers* to take advantage of both approaches to further reduce the number of possibly evicted memory blocks.

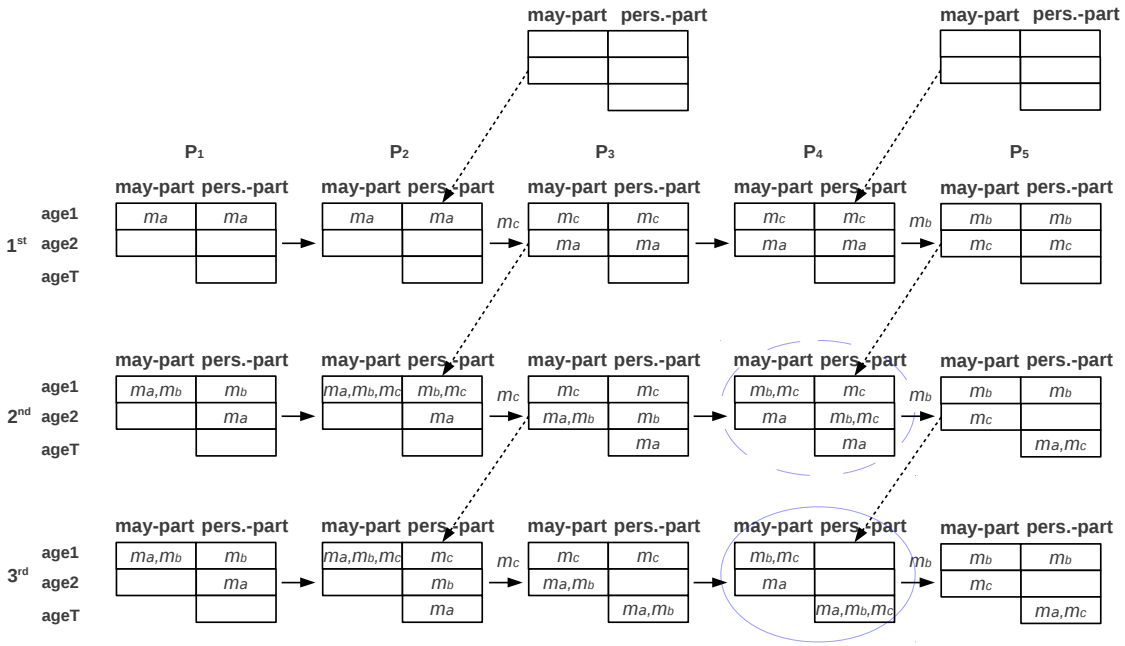


Figure 4.6: Abstract set states of Fig. 4.5 computed by the *Impr. May-Pers* approach

4.4.1 Information Exchange between Abstract Domains

Intuitively, we can take advantage of *YS-Pers* and *May-Pers* by running the two methods separately and then classifying a memory reference as *PS* if at least one method can yield such a classification. However, a more precise approach is to integrate *YS-Pers* and *May-Pers* (*May+YS-Pers*) to form an analysis that runs these two methods in parallel and increase a memory block's potential maximal age if both the methods find its current maximal age is not safe anymore. Thus, we have the abstract domain $E_{\mathscr{P}}$ for cache sets defined as $E_{\mathscr{P}} = D_{\mathscr{P}}^{\text{may-pers}} \times YS$. The join function $\overline{J}_{\mathscr{P}} : E_{\mathscr{P}} \times E_{\mathscr{P}} \to E_{\mathscr{P}}$ just simply joins corresponding components independently by using their own join functions, so it is defined as:

$$\overline{J}_{\mathscr{P}}(\langle \hat{s}_{may}^{p1}, \hat{s}_{pers}^{p1}, ys^{p1} \rangle, \langle \hat{s}_{may}^{p2}, \hat{s}_{pers}^{p2}, ys^{p2} \rangle) := \langle \hat{J}_{\mathscr{P}}(\langle \hat{s}_{may}^{p1}, \hat{s}_{pers}^{p1} \rangle, \langle \hat{s}_{may}^{p2}, \hat{s}_{pers}^{p2} \rangle), \hat{J}_{\mathscr{Y}\mathscr{S}}(ys^{p1}, ys^{p2}) \rangle$$

The update function $\overline{U}_{\mathscr{P}} : E_{\mathscr{P}} \times M \to E_{\mathscr{P}}$ uses our improved update function $\hat{U}_{\mathscr{P}}^{\text{new}}$ on

$D_{\mathscr{P}}^{\text{may-pers}}$ and the update function $\hat{U}_{\mathscr{Y}\mathscr{S}}$ on YS, and is defined as:

$$\overline{U}_{\mathscr{P}}(\langle \hat{s}_{may}, \hat{s}_{pers}, ys\rangle, m) := \langle \hat{s}^X_{may}, \overline{XY}_{\mathscr{P}}(\hat{s}^X_{pers}, \hat{s}^Y_{pers}), ys^Y\rangle$$
$$\text{where } ys^Y = \hat{U}_{\mathscr{Y}\mathscr{S}}(ys, m)$$
$$\hat{s}^Y_{pers} = G_{\mathscr{P}}(ys^Y, set(m))$$
$$\langle \hat{s}^X_{may}, \hat{s}^X_{pers}\rangle = \hat{U}^{\text{new}}_{\mathscr{P}}(\langle \hat{s}_{may}, \hat{s}_{pers}\rangle, m)$$

and the $\overline{XY}_{\mathscr{P}} : D_{\mathscr{P}} \times D_{\mathscr{P}} \to D_{\mathscr{P}}$ function is similar to the join function $\hat{J}_{\mathscr{M}}$ for the traditional may analysis, which means it is to select the smaller one between two possible ages for a memory block. The $\overline{XY}_{\mathscr{P}}$ function is defined as:

$$\overline{XY}_{\mathscr{P}}(\hat{s}^X_{pers}, \hat{s}^Y_{pers}) := [l_i \mapsto \ \{m | \exists a, b \in [1 \ldots \top] : m \in \hat{s}^X_{pers}(l_a) \wedge m \in \hat{s}^Y_{pers}(l_b) \wedge i = \min(a, b)\}]$$

Since at a program point both $\hat{s}^X_{pers}$ and $\hat{s}^Y_{pers}$ would have the same set of memory blocks (that corresponds to the set of memory blocks having been referenced so far), we do not need to check if any block is absent from either $\hat{s}^X_{pers}$ or $\hat{s}^Y_{pers}$.

The strategy that the update function $\overline{U}_{\mathscr{P}}$ uses to increase the age of a memory block in $\hat{s}_{pers}$ can be described as follows: When a memory reference causes the corresponding abstract set state $\langle \hat{s}_{may}, \hat{s}_{pers}, ys\rangle$ to be updated, ys is updated first. When we need to increase a memory block m's potential maximal age, we compare its current age x in $\hat{s}_{pers}$ with the age y computed from its younger set, i.e. $y = |ys(m)| + 1$: if $x < y$, we increase m's potential maximal age as usual; otherwise, we do not increase its maximal age.

Theorem 4.4.1. *The May+YS-Pers approach is safe.*

Proof. Since we do not change the join function on each domain, after two safe abstract states are joined, the resultant state is still safe. To see why the new update strategy is also safe, we consider why a memory block m's current safe maximal age x in $\hat{s}_{pers}$ has to be increased when using the update functions $\hat{U}^{\text{new}}_{\mathscr{P}}$: the contents in $\hat{s}_{may}$ show before x the

cache is *possibly* full, and a newly inserted younger block *might* make x no longer safe. When using the new update strategy, since we also track and update ys independently, the $y = |ys'(m)| + 1$ is the safe maximal age for m where ys' is the younger set mapping after the effect of the newly referenced memory block is taken into account. Thus, if $y \leq x$, we can guarantee x is still safe from independently updated ys and we do not need to increase x. □

For example, consider Fig. 4.5 again. As we have seen in Fig. 4.6, when the second iteration is finished, m_b is not as pessimistic as it is when the fixed points are reached (as shown in the circled states in dashed line and in solid line). When considering the memory reference to m_c in the third iteration, we can observe ys yields $ys(m_a) = \{m_b, m_c\}$, $ys(m_b) = \{m_c\}$, and $ys(m_c) = \emptyset$. Although the $\hat{s}_{pers}$ updating still tries to increase m_b's maximal age to age $\top$, the age computed from $ys(m_b)$ prevents this from happening and cause m_b to stay in age 2. In the end, the fixed point of the abstract set state at p_4 is the same as the one at p_4 of its second iteration (i.e. the circled states will become identical). Thus, the reference to m_b in BB_5 can be classified as *PS*.

4.4.2 Younger Set Generation

For a memory block m, a less precise younger set (i.e. a bigger superset) can be derived from the abstract set state $\langle \hat{s}^p_{may}, \hat{s}^p_{pers} \rangle$ at a program point p: (1) if m can be found in $\hat{s}^p_{pers}$, i.e. $\exists x \in [1 \ldots \top] : m \in \hat{s}^p_{pers}(l_x)$, the potential maximal age of m is x, and each memory block m_a, whose age y in $\hat{s}^p_{may}$ is strictly less than x, i.e. $\exists y \in [1 \ldots A] : m_a \in \hat{s}^p_{may}(l_y) \wedge y < x$, is possibly younger than m (since the *may-part* gives the possible minimal age of a memory block); therefore, one of m's possible younger sets is the set of all the memory blocks whose age in $\hat{s}^p_{may}$ is less than m's age in $\hat{s}^p_{pers}$. (2) if m cannot be found in $\hat{s}^p_{pers}$, it means it has never been brought into the cache yet; thus, its younger set does not exist (i.e. $ys^p(m) = \bot$). Formally, we have a younger set generation function $G_{\mathscr{Y}} : D_{\mathscr{P}} \times M \to (2^M)_\bot$ which is

defined as:

$$G_{\mathscr{Y}}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) := \begin{cases} \bigcup\limits_{1 \le i < x} \hat{s}_{may}(l_i) \backslash \{m\} & \text{if } \exists x \in [2 \dots \top] : m \in \hat{s}_{pers}(l_x) \\ \emptyset & \text{if } m \in \hat{s}_{pers}(l_1) \\ \bot & \text{otherwise} \end{cases}$$

If there were no join points in the program, the generated younger sets could be as precise as the tracked ones, but they would use much less memory, since the tracked ones would have some identical younger blocks in more than one of them. However, when a join point is met, a generated younger set may become less precise since some younger information may be collapsed by may analysis.

The disadvantage of using the combination of *YS-Pers* and *May-Pers* is that: An abstract set state $\langle \hat{s}_{may}, \hat{s}_{pers}, ys \rangle$ may contain a lot of redundant information wasting a lot of storage space, since the same younger sets of some memory blocks can be derived from $\langle \hat{s}_{may}, \hat{s}_{pers} \rangle$ using the $G_{\mathscr{Y}}$ function. In order to decrease this storage overhead, we use two functions to help to compress the size of an abstract set state when saving it and to restore the precise information when using it. The compress function $\overline{C}_{\mathscr{P}} : E_{\mathscr{P}} \to E_{\mathscr{P}}$ is defined as:

$$\overline{C}_{\mathscr{P}}(\langle \hat{s}_{may}, \hat{s}_{pers}, ys \rangle) := \langle \hat{s}_{may}, \hat{s}_{pers}, \ddot{ys} \rangle \quad \text{where}$$

$$\ddot{ys}(m) := \begin{cases} ys(m) & \text{if } G_{\mathscr{Y}}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) \neq ys(m) \\ \bot & \text{otherwise} \end{cases}$$

Thus, at a saving point (e.g. the entry and exit points of a basic block), if a memory block's younger set can be generated from the $\langle \hat{s}_{may}, \hat{s}_{pers} \rangle$ of that point, there is no need to keep it in the saved state. When a saved state needs to be used (e.g. joining several states at a join point), the precise abstract set state can be restored by a restore function $\overline{R}_{\mathscr{P}} : E_{\mathscr{P}} \to E_{\mathscr{P}}$,

which is defined as:

$$\overline{R}_{\mathscr{P}}(\langle \hat{s}_{may}, \hat{s}_{pers}, \ddot{ys} \rangle) := \langle \hat{s}_{may}, \hat{s}_{pers}, ys \rangle \quad \text{where}$$

$$ys(m) := \begin{cases} \ddot{ys}(m) & \text{if } \ddot{ys}(m) \neq \bot \\ G_{\mathscr{Y}}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m) & \text{otherwise} \end{cases}$$

As mentioned in [73], we do not need to continue tracking a memory block m's younger set when it reaches $|ys(m)| = A$. In our case, if $|G_{\mathscr{Y}}(\langle \hat{s}_{may}, \hat{s}_{pers} \rangle, m)| \geq A$ and $|ys(m)| \geq A$ both hold, these two sets are considered as equal.

4.5 Evaluation

We carry out the evaluation on the Mälardalen benchmarks [114], which we compile for the MIPS R3000 architecture. The evaluation is performed by using our research prototype tool, which is described in Appendix A. At first, we compare the number of instruction memory references which are in the loops and cannot be classified as *AH* but can be classified as *PS*. In order to create enough conflicts to observe differences between different methods, we utilize very small cache capacities (128B and 256B) in this experiment.

The experimental results are shown in Tab. 4.1, and the results validate the relationship between different approaches which is described in section 4.2.3 (see the Venn diagram in Fig. 4.3): (1) As shown in Tab. 4.1, the *May+YS-Pers* approach always gives the most number of *PS* references (either "more than" or "as many as"), which shows it dominates other approaches in terms of precision. (2) As we can observe from the results for *bs* under the 128B/8B/2-way configuration, the *Orig. May-Pers* approach can classify more references as *PS* than the *YS-Pers* approach (i.e. 6 references are classified as *PS* by the *Orig. May-Pers* approach but none are classified as *PS* by the *YS-Pers* approach); whereas, under other scenarios, the *YS-Pers* approach is not worse (sometimes much better) than the *Orig. May-Pers* approach. Thus, this shows they are not comparable but empirically in

Table 4.1: The Number of *PS* Instructions under Cache Configurations: 128B/8B/2-way / 256B/8B/4-way (Capacity/Block Size/Associativity)

Benchmark	*Orig. May-Pers*	*Impr. May-Pers*	*YS-Pers*	*May+YS-Pers*
adpcm	29/52	48/53	48/53	48/53
bs	6/16	6/18	0/27	6/27
compress	-/16	-/28	-/29	-/29
edn	28/49	44/98	42/118	44/121
expint	-/3	-/23	-/23	-/27
ludcmp	3/5	4/9	3/47	4/48
matmult	0/2	2/2	3/4	3/7
minver	23/41	29/41	25/67	29/73
ns	1/11	2/11	3/27	3/29
prime	-/0	-/2	-/0	-/2
statemate	-/-	-/-	-/-	-/-
ud	1/4	2/8	2/42	2/42

Note: we use "-" to denote every one is 0 to avoid cluttering.

most cases the *YS-Pers* approach is better. (3) However, the *Orig. May-Pers* and the *Impr. May-Pers* approaches are comparable: the *Impr. May-Pers* approach can always classify more number of references as *PS* than (or at least as many as) the *Orig. May-Pers* classifies. (4) In some cases, the *Impr. May-Pers* approach can have more references classified as *PS* than the *YS-Pers* (e.g. *bs*, *edn*, *ludcmp*, and *minver* benchmarks under the 128B/8B/2-way configuration), but in some cases, the *YS-Pers* can give more *PS*. Thus, the *Impr. May-Pers* and the *YS-Pers* are not comparable. Therefore, we can see the relationship shown in Fig. 4.3 is empirically validated.

The ratio of cache size to loop body size has a direct effect on the usefulness of persistence analysis. From the results for *statemate* benchmark which has a relatively large loop body compared to the cache sizes, we can see that neither of the approaches can classify any reference as *PS*. This is expected, since too many capacity misses in each cache set will evict just referenced instructions soon before they are referenced again. Many of the benchmarks, such as *compress* and *edn*, also contain nested loops which have an effect on the precision of persistence analysis. In the experiments, we do not apply the multi-level method proposed in [120] to deal with the nested loops. Although using the multi-level

method can improve the precision of any persistence analysis approach, the relationship between different approaches will still stay the same.

Next, we want to compare how much storage space and analysis time is used by each method. We save two abstract cache states for each basic block (the states of its entry and exit points). In this experiment, we use 512B, 1KB, 2KB, and 4KB capacities with 8B block size and 4-way associativity. Since some of the used benchmarks are relatively small compared to 2KB and 4KB cache capacities, we only show the results of *adpcm* and *statemate*. The relative memory usage is shown in Fig. 4.7, and the relative analysis time is shown in Fig. 4.8. The shown result is the ratio of memory (analysis time) used by a method to the corresponding value used by *Orig. May-Pers* under the same configuration.

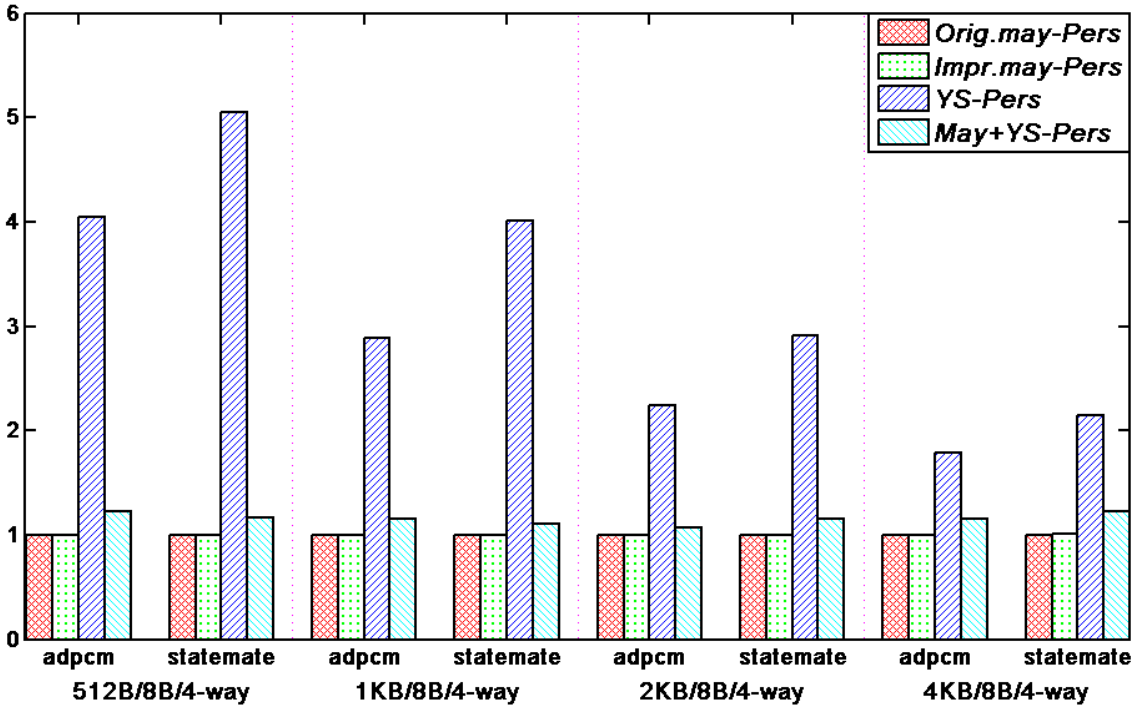


Figure 4.7: Relative storage space used by *adpcm* and *statemate*

Since there may exist redundant information in *YS-Pers*, we expect it requires more space. However, from Fig. 4.7, it is interesting to observe that: as the ratio of the total instruction size to the capacity decreases, the ratio of memory used by *YS-Pers* to that used by *Orig. May-Pers* decreases as well. When the cache capacity increases, there are fewer

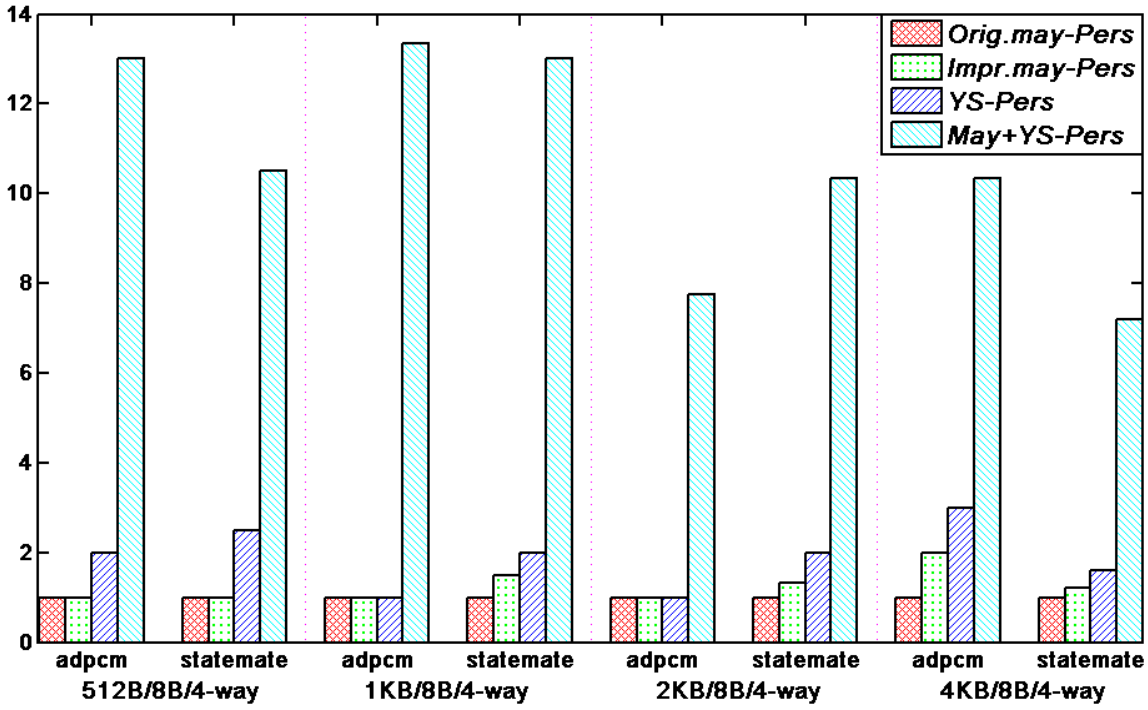


Figure 4.8: Relative analysis time of *adpcm* and *statemate*

instructions mapped into the same cache set, so each memory block has fewer younger blocks, which means less redundant information for a single memory block. From Fig. 4.8, we can find *May+YS-Pers* requires more analysis time than the other approaches. One reason is the analysis iterates more times, and the other reason is it compresses/restores younger block information when processing a basic block in order to save more memory space. Tab. 4.2 shows the ratio of memory used in *May+YS-Pers* by using $\overline{C}_{\mathscr{P}}/\overline{R}_{\mathscr{P}}$ to that without using them. As we can see, the memory space saving is more than 50%.

Table 4.2: Memory Usage Ratio (Compressed / Uncompressed)

Benchmark	512B/8B/4way	1KB/8B/4way	2KB/8B/4way	4KB/8B/4way
adpcm	0.259	0.326	0.384	0.510
statemate	0.194	0.220	0.320	0.457

4.6 Conclusion

In this chapter, we first analyze the sources of pessimism in two recent state-of-the-art safe persistence analysis methods. After identifying the update function of the may analysis-based approach is too pessimistic, we define a new safe update function for that approach but achieve more precision. We also integrate the approaches based on younger set and may analysis together to eliminate more pessimism. Through the evaluations, we observe the proposed techniques can improve the precision of cache persistence analysis. We also observe the trade-offs between precision, memory usage, and analysis time (i.e. the more precision, the more time and/or space spent).

Chapter 5

Top-Down and Bottom-Up Multi-Level Cache Analysis for WCET Estimation

Recently, multi-level cache analysis has drawn much attention in real-time systems [61, 62, 77, 78, 64], since there is a rising need of exploiting the high-performance processors, which are often equipped with multi-level caches. However, compared to single-level cache analysis, multi-level cache analysis is much more challenging. Besides the sequence of memory references, there is a need to take into account the effects of the behavior of one cache level on the behavior of other cache levels (e.g. filtering memory accesses and invalidating memory blocks), which can be different depending on the type of the cache hierarchy.

Typically, there are three cache hierarchy types, which are inclusive, exclusive, and non-inclusive. Multi-level inclusive caches require that the contents at upper cache levels must be a subset of the contents at lower cache levels. On the contrary, multi-level exclusive caches require that the contents at a cache level should not be duplicated at any other cache levels. Multi-level non-inclusive caches allow duplicated contents existing at any cache level, but they do not strictly enforce the inclusion. Moreover, there are some hybrid cache hierarchies, which have some inclusive and/or exclusive cache levels and other levels being non-inclusive. In this chapter, we call a cache hierarchy a multi-level ***inclusive*** cache as long as it maintains the inclusion property at some cache level(s).

Compared to an exclusive/non-inclusive cache hierarchy, a cache hierarchy enforcing inclusion has less effective cache capacity, but the inclusion property can significantly simplify the maintenance of cache coherence [121]. Therefore, multi-level inclusive caches are widely used in many multi-core architectures. A multi-level cache analysis framework that can precisely analyze cache hierarchies that enforce inclusion becomes necessary for WCET estimation.

Most of the current approaches target multi-level non-inclusive cache analysis, and it is not straightforward to extend these approaches to tightly analyze inclusive caches, since the invalidation behavior introduced by maintaining the inclusion property requires making conservative decisions in order to ensure safety [64]. The main idea in this chapter is that this pessimism can actually be reduced by analyzing the multi-level inclusive caches in a *bottom-up* direction, which is counter-intuitive in contrast with the natural *top-down* cache hierarchy access direction that is used in existing methods for multi-level cache analysis. In this chapter, the *top-down* direction is referring to the direction from the uppermost cache level (i.e. L1) downto the lowest cache level, and the *bottom-up* direction is referring to the opposite.

The main technical contributions of this chapter are: (1) We propose an approach which analyzes all the inclusive caches in the *bottom-up* direction first, and then analyzes the rest non-inclusive caches in the *top-down* direction. Due to the *bottom-up* analysis, the invalidation behavior becomes visible at the time of analyzing upper levels; (2) We propose a concept of aging barrier to capture the effects of the invalidations caused by inclusive caches, and by using the aging barriers, we can safely slow down the increase of memory block ages in a cache that is above an inclusive cache level, so more precise *must* and *persistence* analyses can be achieved; (3) We evaluate the proposed approach using a set of benchmarks, and we find the proposed approach can tighten the WCET estimation by 12.2% on average, compared to the approach proposed in [64]. In this chapter, we only consider multi-level inclusive instruction caches for a single processor. Although the effects of data references and inter-core interferences are not considered, this approach can serve as a basis for such extensions. Our approach has been published in [12].

The rest of the chapter is organized as: Section 5.1 shows why a multi-level inclusive cache is hard to analyze for WCET estimation; Section 5.2 gives the system model considered in this chapter; Section 5.3 presents our multi-level inclusive cache analysis; Section 5.4 formally proves the proposed approach is safe and can terminate; Section 5.5 evaluates

the proposed approach; and Section 5.6 concludes this chapter.

5.1 Problem Statement

In the case of single-level cache analysis, only the effects of the memory reference sequences need to be taken into account. In order to make the analysis scalable, most of the approaches are based on abstract interpretation. An abstract interpretation based approach aims to assign a cache hit/miss classification (CHMC) to each memory reference according to the abstract cache states (ACSs) derived by three different analyses [85, 76]. The analyses are usually performed on the control-flow graph (CFG) reconstructed from the low-level code of the program. At a given program point, a *must* analysis is used to determine the set of memory blocks that are *definitely* in the cache, so a memory reference to a block being in the set can be classified as *always hit* (*AH*); a *may* analysis is used to determine the set of memory blocks that are *possibly* in the cache, so a memory reference to a block not being in the set can be classified as *always miss* (*AM*); a *persistence* analysis is used to determine the set of memory blocks that stay in the cache once they are loaded, and a memory reference to such a block is classified as *persistent* (*PS*) or *first miss* (*FM*); and, if a memory reference cannot be classified as *AH*, *AM*, or *PS*, it is classified as *not classified* (*NC*).

When analyzing multi-level caches, it is also important to consider the effects of other cache levels, like cache access filtering and memory block invalidation. For example, if we treat every possible access at a level as always happening, the analysis may become unsafe, since doing so may underestimate the set reuse distances[1] of memory blocks [61].

For a reference at a cache level, a cache access classification (CAC) can be used to represent whether the cache access at this level will occur: *always* (*A*) denotes the access will always occur, *never* (*N*) denotes the access will never happen, and *uncertain* (*U*) denotes

[1] In [61], the set reuse distance between two memory references to the same block at a cache level is defined as the relative age of the memory block when the second reference occurs.

the access may occur [61]. In order to ensure safety, the updates of the abstract cache states due to U accesses need to take into account the two possible cases (access occurring and not occurring).

In the case of multi-level non-inclusive cache analysis, the CAC for a reference r at a cache level l can be derived from the CHMC and CAC for r at $l-1$ (as described in [61]), and the behavior of l will not be affected by any lower cache level. However, in the case of analyzing cache hierarchies containing inclusive caches, the CAC for r at l cannot be safely derived from CHMC and CAC for r at $l-1$. The reason is the behavior of l depends not only on the behavior of $l-1$, but also on the invalidation behavior induced by some lower inclusive cache level(s): When a memory block is evicted from a lower inclusive cache level, all the contents that belong to this memory block need to be invalidated from its upper cache levels (the invalidated memory blocks are called *inclusion victims*).

Example: Fig. 5.1 shows a 3-level inclusive cache, where L1 is 2-way set associative, L2 is 4-way set associative, and L3 is 8-way set associative (at each level, only one set is shown). We assume L1 has the smallest cache block size and L3 has the biggest, so a block in L1 is a sub-block of some block in L2 and that block in L2 is a sub-block of some block in L3. For a memory block m in L3, let $\dot{m}$ denote a m's sub-block in L2, and let $\ddot{m}$ denote a $\dot{m}$'s sub-block in L1. For example, we have $\ddot{m}_a \subset \dot{m}_a \subset m_a$. If the next reference needs the information that is in m_x (m_x is also mapped to the shown set of L3), the oldest m_a in that set needs to be evicted. The eviction of m_a will also invalidate $\ddot{m}_a$ in L1 and $\dot{m}_a$ in L2 to maintain the inclusion property. Due to the invalidation, $\ddot{m}_h$ in L1 can live longer, and depending on which sub-block of m_x is needed by the reference, there may be some "holes" left in L1 and L2.

In [64], multi-level non-inclusive cache analysis is adapted to multi-level inclusive cache analysis. To achieve this, several conservative decisions are made on the CAC and CHMC for a reference at a cache level due to any possible invalidation to ensure safety: (1) Except for L1 which is always accessed, the CAC at any other level should be classified

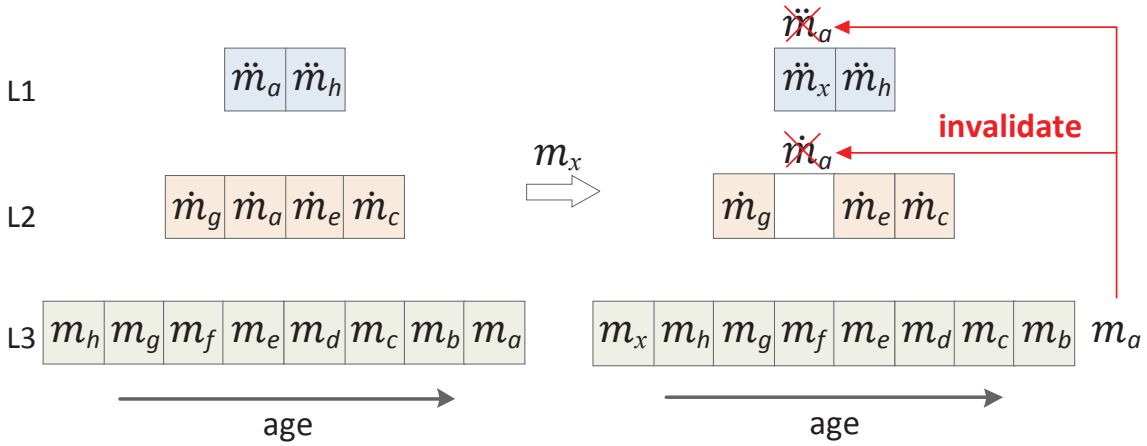


Figure 5.1: Invalidation due to the maintenance of the inclusion property of L3

as *U*; (2) If a reference is classified as *AH* or *PS* at a level, this CHMC may be changed into *NC* depending on the analysis of lower inclusive levels; (3) Even if a memory reference is classified as *AM* at a level, this CHMC has to be changed into *NC*. In this way, although safety is ensured, the tightness of the estimation may suffer a lot. Therefore, we need a method that can more precisely analyze the effects of multi-level inclusive caches on WCET estimation.

5.2 System Model

We focus on a general multi-level inclusive cache model. The model has p cache levels, where $p \geq 2$, among which q levels are inclusive, where $p > q \geq 1$, and the other $p-q$ levels are non-inclusive[2]. We also assume the time for a processing element to access a cache level is bounded and predictable, which can be achieved by using deterministic interconnects to connect the caches, like TDMA buses [122].

Let $L = \{l_x | 1 \leq x \leq p\}$ be the set of all the cache levels, in which l_x denotes the x^{th} cache level. Let I be the set of all the inclusive cache levels, and let N be the set of all the non-inclusive cache levels. Thus, we have $L = I \cup N \bigwedge I \cap N = \emptyset \bigwedge |I| = q$. Since it does not

[2]It has no meaning for L1 cache to be inclusive/non-inclusive. Later, we treat L1 as non-inclusive to facilitate the presentation. Thus, we assume $p > q$ not $p \geq q$.

matter whether l_1 is inclusive or non-inclusive, we can simply assume $l_1 \in N$, so neither I nor N is an empty set. Fig. 5.2 gives two examples of the models focusing on single cores of two multi-core architectures.

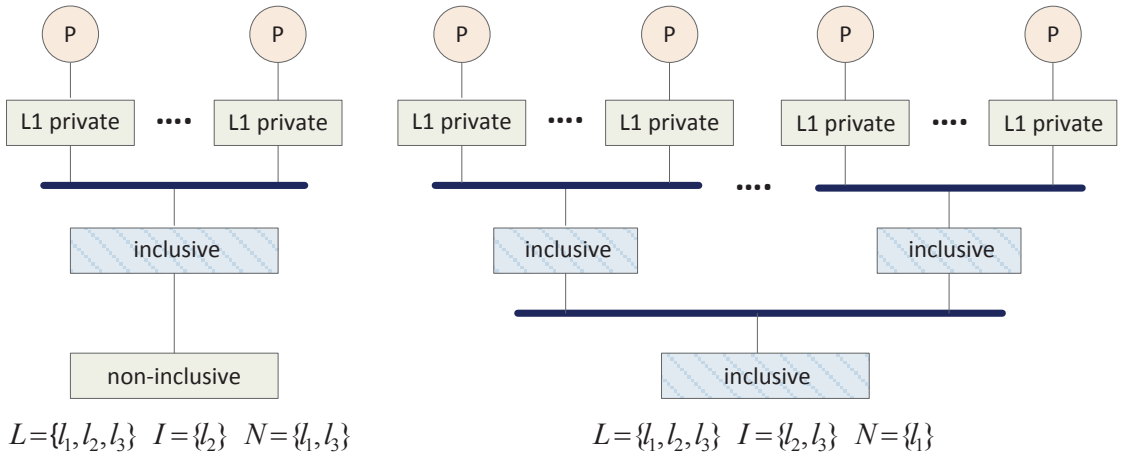


Figure 5.2: Two examples of the models with respect to a single core

We assume at each level the cache is set associative, and least recently used (LRU) replacement policy is used. The size of a cache block can be different at different cache levels, and it is common to assume the block size does not increase as the level goes up. It is also common to assume the capacity decreases as the level goes up. Let C_{l_x} denote the cache at the cache level l_x, let A_{l_x} denote the associativity of C_{l_x}, and let s_{l_x} denote the number of cache sets of C_{l_x}. Sometimes we use "cache level" to actually mean the cache located at that level if there is no ambiguity.

Although we do not consider exclusive caches in the model, we can easily add them into our analysis by using the approach proposed in [64]. Basically, the exclusive cache levels can be collapsed by concatenating them to the end of the upper level to form a single level for the analysis, as long as they all have the same number of cache sets and the same cache block size. In this chapter, we focus on how to analyze multi-level caches in the presence of invalidations caused by inclusion enforcement, so we simply consider multi-level instruction caches in terms of a single processor. This work can serve as a basis for analysis of multi-level data or unified caches, that may also suffer from invalidations, in

terms of a multi-core processor.

In order to facilitate the presentation, we introduce the following notations. As described in [85], an abstract cache state is a mapping from a cache set number to an abstract set state, where an abstract set state is a mapping from a position to a set of memory blocks. For the cache C_{l_x}, let $\alpha_{l_x}^{must}$, $\alpha_{l_x}^{may}$, and $\alpha_{l_x}^{pers}$ denote the abstract cache states of C_{l_x} with respect to the cache *must*, *may*, and *persistence* analysis respectively; and let ACS^{must}, ACS^{may}, and ACS^{pers} denote the sets of all of the abstract cache states of these three analyses. For an abstract cache state α_{l_x} (that is either $\alpha_{l_x}^{must}$, $\alpha_{l_x}^{may}$, or $\alpha_{l_x}^{pers}$), let $\alpha_{l_x}(i)$ give the i^{th} abstract set state of α_{l_x}, and let $\alpha_{l_x}(i)(h)$ give the set of memory blocks corresponding to the h^{th} position in $\alpha_{l_x}(i)$.

Let $\mathscr{U}^{must}$ and $\mathscr{J}^{must}$ represent the update and join functions for single-level cache *must* analysis. Similarly, let $\mathscr{U}^{may}$ and $\mathscr{J}^{may}$ represent the update and join functions for single-level cache *may* analysis. These two sets of functions are well-known and defined in [85]. Furthermore, let $\mathscr{U}^{pers}$ and $\mathscr{J}^{pers}$ represent the update and join functions for single-level cache *persistence* analysis. Since the original *persistence* analysis has been known unsafe, we can use the corresponding functions of the safe *persistence* analyses defined in [73] or [76].

For a memory reference r at a cache level l_x, let $m_{l_x}^r$ denote the memory block that contains the information r needs with respect to the cache block size and the number of cache sets in C_{l_x}. We use $m_{l_x}^r \in C_{l_x}$ to denote the needed memory block is in the corresponding concrete set state of C_{l_x}, and use $m_{l_x}^r \in \alpha_{l_x}^t$ to denote the block is in the corresponding abstract set state of t-analysis at this level, where t is either must, may, or persistence.

5.3 Multi-Level Inclusive Cache Analysis: Going Top-Down or Bottom-Up?

To our knowledge, existing work analyzes the cache hierarchies in a *top-down* direction, since it is the natural direction of accessing a multi-level cache. As long as there are no invalidations at any cache level, a *top-down* analysis can be safe and precise. However,

when there are inclusive caches in the cache hierarchy, a *top-down* analysis cannot capture the possible invalidation behavior precisely, since the invalidations appearing at a cache level are actually caused by the inclusive caches located below this level. Thus, as discussed in [64], conservative decisions have to be made to ensure safety which makes the analysis pessimistic.

In order to make the analysis of multi-level inclusive caches more precise, we propose a safe approach which analyzes the cache hierarchy in a rather counter-intuitive way: We first analyze all the inclusive cache levels in the *bottom-up* direction so as to make the possible invalidation behavior visible at a cache level, and then we analyze all the non-inclusive levels in the traditional *top-down* direction taking into account the revealed invalidations. The analysis process is shown in Fig. 5.3.

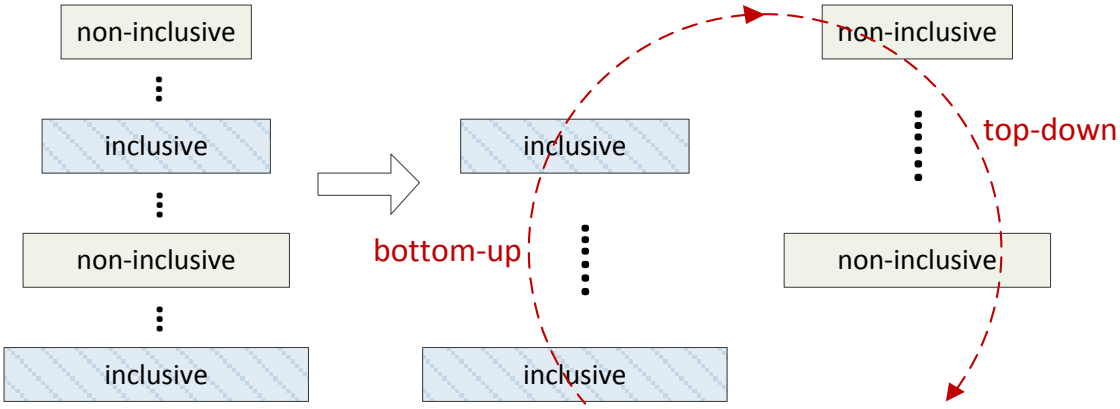


Figure 5.3: Multi-level inclusive cache analysis: going *bottom-up* and *top-down*

Our *bottom-up* analysis of inclusive caches is based on the following observation, that is related to the amount of information that can be derived for the access to an inclusive cache level l_y from the state of C_{l_y}.

Lemma 5.3.1. *When a memory reference r occurs,*

1. *l_y will be definitely accessed, if $m^r_{l_y} \notin C_{l_y}$.*

2. *l_y will be possibly accessed, if $m^r_{l_y} \in C_{l_y}$.*

Proof. If $m^r_{l_y}$ is not in C_{l_y}, it means all the contents of $m^r_{l_y}$ are not in any C_{l_x} neither, where $l_1 \leq l_x < l_y$, due to the enforced inclusion property of C_{l_y}; so l_y will be definitely accessed. However, if $m^r_{l_y}$ is already in C_{l_y}, we cannot determine whether there are some sub-blocks of $m^r_{l_y}$ that have the needed contents at above levels only from the state of C_{l_y}, so l_y will be possibly accessed. □

Based on this lemma, we show that we can first analyze each inclusive level in the *bottom-up* direction safely, and use the results of one inclusive level's analyses to guide its upper levels' analyses to derive more precise CHMC. Note that for a memory reference r which may access the cache level l_x, we can always use $\mathscr{J}^t(\mathscr{U}^t(\alpha^t_{l_x}, m^r_{l_x}), \alpha^t_{l_x})$ to handle the access uncertainty so as to carry out a safe t-analysis at this level, where t is either *must*, *may*, or *persistence* [61]. However, the more uncertainty we can resolve, the more precise the analysis can become.

5.3.1 Last Inclusive Cache Analysis

The proposed multi-level inclusive cache analysis begins with the last inclusive cache. There can be other non-inclusive caches located between the last inclusive cache and the main memory. Let us assume the last inclusive cache level corresponds to $l_{\text{LIC}} \in I$, so we have $\forall l_x \in L : x > \text{LIC} \implies l_x \in N$.

5.3.1.1 Last Inclusive Cache May Analysis

At a program point, if a memory block is not in the abstract cache state of a safe *may* analysis of the cache, it is definitely not in any concrete state of the cache. Therefore, if we can safely perform a *may* analysis of the last inclusive cache, we can use the $\alpha^{may}_{l_{\text{LIC}}}$ to safely classify some memory references as *AM* at a cache level l_x where $1 \leq x \leq \text{LIC}$ based on the inclusion property.

For the *may* analysis of the last inclusive cache, we define the join function $\mathscr{J}^{may}_{\text{LIC}}$ and

update function $\mathscr{U}_{\text{LIC}}^{may}$ as follows:

$$\mathscr{J}_{\text{LIC}}^{may} = \mathscr{J}^{may}$$

$$\mathscr{U}_{\text{LIC}}^{may}(\alpha_{l_{\text{LIC}}}^{may}, m_{l_{\text{LIC}}}^{r}) = \begin{cases} \mathscr{J}^{may}(\mathscr{U}^{may}(\alpha_{l_{\text{LIC}}}^{may}, m_{l_{\text{LIC}}}^{r}), \alpha_{l_{\text{LIC}}}^{may}) & \text{if } m_{l_{\text{LIC}}}^{r} \in \alpha_{l_{\text{LIC}}}^{may} \\ \mathscr{U}^{may}(\alpha_{l_{\text{LIC}}}^{may}, m_{l_{\text{LIC}}}^{r}) & \text{otherwise} \end{cases}$$

where $\mathscr{J}_{\text{LIC}}^{may}$ is the join function of the single-level cache *may* analysis, and the update function $\mathscr{U}_{\text{LIC}}^{may}$ is defined with respect to the two cases in Lemma 5.3.1 for a memory reference r: If $m_{l_{\text{LIC}}}^{r} \notin \alpha_{l_{\text{LIC}}}^{may}$, we can deduce $m_{l_{\text{LIC}}}^{r} \notin C_{l_{\text{LIC}}}$ (this is formally proven in Lemma 5.4.1), so it is certain that l_{LIC} will be accessed, and using $\mathscr{U}^{may}(\alpha_{l_{\text{LIC}}}^{may}, m_{l_{\text{LIC}}}^{r})$ is certainly safe; if $m_{l_{\text{LIC}}}^{r} \in \alpha_{l_{\text{LIC}}}^{may}$, $m_{l_{\text{LIC}}}^{r}$ may be in $C_{l_{\text{LIC}}}$ and l_{LIC} may be accessed, so we use $\mathscr{J}^{may}(\mathscr{U}^{may}(\alpha_{l_{\text{LIC}}}^{may}, m_{l_{\text{LIC}}}^{r}), \alpha_{l_{\text{LIC}}}^{may})$ to safely update the $\alpha_{l_{\text{LIC}}}^{may}$ by taking into account both the access occurring and not occurring.

Therefore, at a program point, $\alpha_{l_{\text{LIC}}}^{may}$ contains all the memory blocks that are possibly in $C_{l_{\text{LIC}}}$ when the execution reaches this point. If a memory reference r is classified as *AM* by the last inclusive cache *may* analysis (i.e. $m_{l_{\text{LIC}}}^{r} \notin \alpha_{l_{\text{LIC}}}^{may}$), we can safely categorize r as *AM* at any cache level l_x where $1 \leq x \leq \text{LIC}$, since, according to the inclusion property, if a memory block is absent from the underlying inclusive cache, it is also absent from all of the included upper-level caches. Therefore, compared to the *top-down* approach proposed in [64], which needs to conservatively change any reference classified as *AM* to *NC* at any cache level, the approach is more precise.

5.3.1.2 Last Inclusive Cache Must and Persistence Analysis

At a program point, the proposed *must* and *persistence* analyses of the last inclusive cache depend on the $\alpha_{l_{\text{LIC}}}^{may}$ of that point. This is because only the information deduced from

$\alpha^{may}_{l_{\text{LIC}}}$ can be used to determined whether the l_{LIC} will be definitely accessed according to Lemma 5.3.1.

For the last inclusive cache *must* (resp. *persistence*) analysis, we define the join function $\mathscr{J}^{must}_{\text{LIC}}$ (resp. $\mathscr{J}^{pers}_{\text{LIC}}$) and update function $\mathscr{U}^{must}_{\text{LIC}}$ (resp. $\mathscr{U}^{pers}_{\text{LIC}}$) as follows:

$$\mathscr{J}^{must}_{\text{LIC}} = \mathscr{J}^{must}$$

$$\mathscr{U}^{must}_{\text{LIC}}(\alpha^{must}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) = \begin{cases} \mathscr{J}^{must}(\mathscr{U}^{must}(\alpha^{must}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}), \alpha^{must}_{l_{\text{LIC}}}) & \text{if } m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}} \\ \mathscr{U}^{must}(\alpha^{must}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) & \text{otherwise} \end{cases}$$

$$\mathscr{J}^{pers}_{\text{LIC}} = \mathscr{J}^{pers}$$

$$\mathscr{U}^{pers}_{\text{LIC}}(\alpha^{pers}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) = \begin{cases} \mathscr{J}^{pers}(\mathscr{U}^{pers}(\alpha^{pers}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}), \alpha^{pers}_{l_{\text{LIC}}}) & \text{if } m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}} \\ \mathscr{U}^{pers}(\alpha^{pers}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}) & \text{otherwise} \end{cases}$$

where $\mathscr{J}^{must}_{\text{LIC}}$ (resp. $\mathscr{J}^{pers}_{\text{LIC}}$) is just the join function of the single-level cache *must* (resp. *persistence*) analysis, and similar to $\mathscr{U}^{may}_{\text{LIC}}$, for a memory reference r, the update function $\mathscr{U}^{must}_{\text{LIC}}$ (resp. $\mathscr{U}^{pers}_{\text{LIC}}$) is defined to safely update the $\alpha^{must}_{l_{\text{LIC}}}$ (resp. $\alpha^{pers}_{l_{\text{LIC}}}$) by using the join function to merge the two abstract cache states (i.e. one state corresponds to the access occurring and the other corresponds to the access not occurring), if $m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}}$ (i.e. $m^r_{l_{\text{LIC}}}$ is possibly in $C_{l_{\text{LIC}}}$); otherwise, $m^r_{l_{\text{LIC}}}$ is definitely not in $C_{l_{\text{LIC}}}$, so it can more precisely update the abstract cache state by knowing the access definitely occurs.

Thus, at any program point, the memory blocks contained in $\alpha^{must}_{l_{\text{LIC}}}$ are definitely in $C_{l_{\text{LIC}}}$, and the memory blocks not contained in the $\top$ age positions of $\alpha^{pers}_{l_{\text{LIC}}}$ are persistent when the execution reaches this point. If a memory reference r is classified as *AH* by the last inclusive cache *must* analysis (i.e. $m^r_{l_{\text{LIC}}} \in \alpha^{must}_{l_{\text{LIC}}}$), this reference will cause no cache misses

at l_{LIC}, but may result in misses at a cache level l_x where $1 \leq x < \text{LIC}$. In other words, this classification for this memory reference is only locally safe. If r is classified as *AH* by the last inclusive cache *must* analysis, no memory blocks need to be evicted from C_{l_x} because of this reference, so no invalidations are enforced by $C_{l_{\text{LIC}}}$. Similarly, if a memory reference r is classified as *PS* by the last inclusive cache *persistence* analysis (i.e. $m^r_{l_{\text{LIC}}}$ is not in $\top$ of the corresponding set of $\alpha^{pers}_{l_{\text{LIC}}}$), r will result in at most one cache miss at l_{LIC}, but may cause more than one misses at a cache level l_x where $1 \leq x < \text{LIC}$. Finally, if r is classified as *PS* by the last inclusive cache *persistence* analysis, at most one memory block will be evicted from $C_{l_{\text{LIC}}}$ so that at most one invalidation enforcement can be caused because of r.

5.3.2 Aging Barriers

In order to analyze a cache located above an inclusive cache level more precisely, the effects of the invalidations need to be captured. Since the invalidations are caused by lower inclusive caches, compared to the *top-down* approach, one advantage of the *bottom-up* approach is the invalidation behavior becomes visible when analyzing an upper level.

At a cache level, if a memory block is invalidated due to the maintenance of the inclusion property, a "hole" will be left in the cache; until this "hole" is filled by some memory block, any access to the corresponding cache set will not increase the ages of the memory blocks that are behind this "hole". Yet, it does not mean the age of a memory block behind the "hole" will not be decreased, since a reference to such a block will decrease its age to 1 and fill the "hole", in which case another "hole" will be created behind the filled "hole". A "hole" will be filled and no new one will be created when the referenced memory block is not in the cache.

We propose a concept of aging barrier to capture this "hole" behavior so as to perform more precise *must* and *persistence* analyses of a cache that may suffer from invalidations. Without loss of generality, we present the concept in terms of an A-way set associative cache C which has s cache sets.

Definition 5.3.2 (Aging Barrier)**.** *A valid aging barrier* (i, j) *satisfies* $1 \leq i \leq s \bigwedge 1 \leq j \leq A$*, and represents an unused position within the range* $[1, j]$ *in the* i^{th} *cache set, which prevents the age of any memory block in the* i^{th} *abstract set state of* α^{must} *or* α^{pers} *from increasing if the age is already greater than or equal to* j *for an access.*

We treat an aging barrier (i, j) as an abstract *must* "hole": if there is a valid aging barrier (i, j) at a program point, in any concrete state of C, there must be a corresponding real "hole" appearing in the i^{th} cache set of C within the position range $[1, j]$. Thus, j serves as the position upper bound of the real "hole". For example, the aging barrier $(1, 2)$ represents either the 1^{st} or the 2^{nd} young memory block in the 1^{st} cache set is invalidated and the position it occupied becomes available.

It is possible to have multiple valid aging barriers with respect to the i^{th} cache set, which are listed as $(i, j_1), \cdots, (i, j_k)$ where $k \geq 1$. In that case, there are certainly at least k real "holes" in the i^{th} cache set, whose positions are bounded by $j_1, \cdots, j_k$ respectively. Note that it is valid to have multiple identical j's with respect to the i^{th} cache set, as long as the multiset [3] formed by these upper bounds satisfies the condition: Given any position *pos* in the cache set, the total number of j's with $j \leq pos$ is at most *pos*. Let Ξ denote the set of all of the valid multisets formed by "hole" position upper bounds of a cache set. Formally, we have:

$$\xi \in \Xi \iff \max(\xi) \leq A \bigwedge \forall pos \in \{1, \cdots, A\} : \sum_{j=1}^{pos} \nu(\xi, j) \leq pos$$

where $\max(\xi)$ gives the maximum member and $\nu(\xi, j)$ gives the multiplicity of j in the multiset ξ.

Definition 5.3.3 (Aging Barrier State)**.** *An aging barrier state* $\beta : \{1, \cdots, s\} \to \Xi$ *is a mapping from a cache set number to a multiset of "hole" position upper bounds.*

Given an aging barrier state β, the set of all the valid aging barriers is $\{(i, j)^{\nu(\beta(i), j)} | i \in$

[3] A multiset is a set in which members are allowed to appear more than once.

$\{1,\cdots,s\} \bigwedge j \in \beta(i)\}$, which is a multiset and uses $\nu(\beta(i), j)$ as the multiplicity of (i, j). Let *ABS* denote the set of all the aging barrier states of C. We define three functions to operate on the aging barrier states.

Let $\top = A+1$ be the invalid aging barrier indicator. The function $\mathscr{A} : ABS \times \{1,\cdots,s\} \times \{1,\cdots,A,\top\} \to ABS$ is used to add an aging barrier into the state and is defined as: [4]

$$\mathscr{A}(\beta, i, j) = \beta\left[i \mapsto \beta(i) \uplus_c \{j\}\right]$$

$$\text{where } \beta(i) \uplus_c \{j\} = \begin{cases} \beta(i) \uplus \{j\} & \text{if } \beta(i) \uplus \{j\} \in \Xi \\ \beta(i) & \text{otherwise} \end{cases}$$

The function adds the aging barrier (i, j) into the state β only if the result of $\beta(i) \uplus \{j\}$ ($\uplus$ is the multiset sum operation) is a member of Ξ; otherwise, it keeps β unchanged. For example, given a 4-way set associative cache (i.e. A is 4), when we want to add an aging barrier $(1,3)$ into the state β, the function $\mathscr{A}$ needs to check if $\beta(1) \uplus \{3\}$ is a member of Ξ. Assume we have $\beta(1) = \{2,2\}$; then $\beta(1) \uplus \{3\} = \{2,2,3\}$ is a member of Ξ according to the condition given above – the maximum member in $\{2,2,3\}$ is 3 that is less than 4 and no matter what *pos* is, the total number of the members that are less than or equal to *pos* is at most *pos*. Therefore, after applying $\mathscr{A}(\beta, 1, 3)$, we will have $\beta(1) = \{2,2,3\}$.

The function $\mathscr{U} : ABS \times \{1,\cdots,s\} \to ABS \times \{1,\cdots,A,\top\}$ is used to acquire an aging barrier from the state and is defined as:

$$\mathscr{U}(\beta, i) = \left\langle \beta\left[i \mapsto \beta(i) \backslash \{\min_c(\beta(i))\}\right], \min_c(\beta(i)) \right\rangle$$

$$\text{where } \min_c(\beta(i)) = \begin{cases} \min(\beta(i)) & \text{if } \beta(i) \neq \emptyset \\ \top & \text{otherwise} \end{cases}$$

Given a cache set number i, the resultant aging barrier depends on whether the mapped mul-

[4] For a function $f : X \to Y$, $f\left[i \mapsto k\right]$ means $f(i) = k \bigwedge \forall x \in X \wedge x \neq i : f(x) = f(x)$.

tiset $\beta(i)$ is empty: If $\beta(i)$ is not empty, $\min_c(\beta(i))$ equals $\min(\beta(i))$ that is the minimum member in $\beta(i)$, and the composite $(i, \min(\beta(i)))$ will be a valid aging barrier; otherwise, $\min_c(\beta(i))$ equals $\top$ and there is no valid aging barrier for the i^{th} cache set. Since a valid aging barrier may be acquired in which case this aging barrier should no longer be in the state, the function changes the state by mapping i to $\beta(i) \backslash \{\min_c(\beta(i))\}$ ($\backslash$ is the multi-set asymmetric difference operation). For example, let us continue with the last example in which we have $\beta(1) = \{2,2,3\}$. Since the minimum member in $\{2,2,3\}$ is 2, after applying $\mathscr{U}(\beta, 1)$, we have a valid aging barrier $(1,2)$ and $\beta(1)$ becomes $\{2,3\}$.

The function $\mathscr{J} : ABS \times ABS \rightarrow ABS$ is used to join two aging barrier states and is defined as:

$$\mathscr{J}(\beta_1, \beta_2) = \left[i \mapsto \beta_1(i) \sqcap_c \beta_2(i) | i = 1, \cdots, s \right]$$

$$\text{where } \beta_1(i) \sqcap_c \beta_2(i) = \begin{cases} \emptyset & \text{if } \beta_1(i) = \emptyset \bigvee \beta_2(i) = \emptyset \\ \{j_1, \cdots, j_k\} & \text{otherwise} \end{cases}$$

$$\text{where } k = \min(|\beta_1(i)|, |\beta_2(i)|) \bigwedge$$

$$j_1 = \max(\min_c(\beta_1(i)), \min_c(\beta_2(i)) \bigwedge$$

$$j_2 = \max(\min_c^2(\beta_1(i)), \min_c^2(\beta_2(i)) \bigwedge$$

$$\cdots$$

$$j_k = \max(\min_c^k(\beta_1(i)), \min_c^k(\beta_2(i)))$$

where $\min_c^k(\beta(i))$ is similar to $\min_c(\beta(i))$ except it gives the k^{th} minimum member of $\beta(i)$ if $\beta(i)$ has at least k members (of course, if $\beta(i)$ does not have that many members, it gives $\top$). When joining two aging barrier states, for the i^{th} cache set, the cardinality of $\beta_1(i) \sqcap_c \beta_2(i)$ (i.e. k) is the smaller one of the cardinalities of $\beta_1(i)$ and $\beta_2(i)$, which implies the number of aging barriers that can be derived from $\mathscr{J}(\beta_1, \beta_2)$ will never exceed that derived from either β_1 or β_2. In the case of $k \geq 1$, j_1 is the bigger one between the

two minimum members of $\beta_1(i)$ and $\beta_2(i)$, which safely captures an aging barrier since there must be a "hole" within position range $[1, j_1]$ along either path; and j_2 is the bigger one between the 2^{nd} minimum members of $\beta_1(i)$ and $\beta_2(i)$. We repeat this process until we have j_k which is the bigger one between the k^{th} minimum members of $\beta_1(i)$ and $\beta_2(i)$. For example, assume we have $\beta_1(1) = \{2,2\}$ and $\beta_2(1) = \{1,3,4\}$. After applying $\beta = \mathscr{J}(\beta_1, \beta_2)$, we will have $\beta(1) = \{2,3\}$, since, for the 1^{st} cache set, we have $k = 2$, $j_1 = 2$, and $j_2 = 3$ when performing $\{2,2\} \sqcap_c \{1,3,4\}$.

Definition 5.3.4 (Partial Ordering)**.** *Let β_1 and β_2 be two aging barrier states. We define $\beta_1 \sqsubseteq \beta_2$ if and only if* $\forall i \in \{1, \cdots, s\} : |\beta_2| \leq |\beta_1| \bigwedge \min_c(\beta_1(i)) \leq \min_c(\beta_2(i)) \bigwedge \min_c^2(\beta_1(i)) \leq \min_c^2(\beta_2(i)) \bigwedge \cdots \bigwedge \min_c^{|\beta_2|}(\beta_1(i)) \leq \min_c^{|\beta_2|}(\beta_2(i))$.

Therefore, we have $\beta_1 \sqsubseteq \beta_2$, if and only if, for any cache set i, the mapped multisets $\beta_1(i)$ and $\beta_2(i)$ satisfy: the number of members of $\beta_2(i)$ is not greater than that of $\beta_1(i)$, and when we iterate the two multisets in the ascending order in parallel, the iterated number from $\beta_2(i)$ is not smaller than that from $\beta_1(i)$. According to Definition 5.3.4, we can deduce that $\beta_1 \sqsubseteq \mathscr{J}(\beta_1, \beta_2)$ and $\beta_2 \sqsubseteq \mathscr{J}(\beta_1, \beta_2)$. Let $\beta_\perp = \big[i \mapsto \{1, \cdots, A\} | i = 1, \cdots, s\big]$ and $\beta_\top = \big[i \mapsto \emptyset | i = 1, \cdots, s\big]$; thus, according to Definition 5.3.4, we can deduce that $\forall \beta \in ABS : \beta_\perp \sqsubseteq \beta \sqsubseteq \beta_\top$.

5.3.3 Integrating Aging Barriers into Update Functions

In order to realize more precise *must* and *persistence* analyses of the caches which suffer from invalidations, we need to integrate the aging barriers into the update functions of these analyses. Let M denote the set of all the memory blocks. Given a reference to a memory block $m \in M$ that is mapped to the i^{th} set of C and an aging barrier (i, j), where $j \in \{1, \cdots, A, \top\}$ (recall that we use $\top$ as the invalid aging barrier indicator), we redefine the update function $\overline{\mathscr{U}}^{must} : ACS^{must} \times M \times \{1, \cdots, A, \top\} \rightarrow ACS^{must}$ for the *must* analysis

as:

$$\overline{\mathscr{U}}^{must}(\alpha^{must}, m, j) = \begin{cases} \mathscr{U}^{must}(\alpha^{must}, m) & \text{if } k \leq j \\ \alpha^{must}\left[i \mapsto \varepsilon_i\right] & \text{otherwise} \end{cases}$$

$$\text{where } k = \begin{cases} h & \text{if } m \in \alpha^{must}(i)(h) \\ \top & \text{otherwise} \end{cases} \bigwedge$$

$$\varepsilon_i = \begin{array}{l} \big[\iota_1 \mapsto \{m\}, \\ \quad \iota_n \mapsto \alpha^{must}(i)(\iota_{n-1}) | n = 2, \cdots, j-1, \\ \quad \iota_j \mapsto \alpha^{must}(i)(\iota_j) \cup \alpha^{must}(i)(\iota_{j-1}), \\ \quad \iota_n \mapsto \alpha^{must}(i)(\iota_n) \backslash \{m\} | n = j+1, \cdots, A\big] \end{array}$$

The rationale of the redefined update function is: If there is no valid aging barrier available (i.e. $j = \top$), or if the current valid aging barrier (i, j) is not needed (i.e. $m \in \alpha^{must}(i)(h) \bigwedge j \leq A \bigwedge h \leq j$, in which case this update never attempts to affect the ages of the memory blocks "protected" behind this aging barrier), then we can simply use the $\mathscr{U}^{must}$ to update the α^{must}; otherwise, the current aging barrier can prevent the memory blocks that are behind it in the corresponding abstract set state from aging, since it means there is a "hole" before j (including j) that needs to be filled, and we can only increase the ages of the memory blocks until j, and keep the ages of other blocks not increased (excluding m which will be moved to the first age position if it is in the current state). Fig. 5.4 shows an example of using an aging barrier to update α^{must} more precisely – if m_c in α^{must} is invalidated, since it is definitely in the cache before the invalidation with an over-estimated maximal age 3, a "hole" will definitely appear within the range $[1,3]$, namely we have an aging barrier with $j = 3$; when m_d is referenced, even if it is not in the cache, there is a "hole" to fill, the maximal ages of m_b and m_a should not be increased. Therefore, using

the redefined function $\overline{\mathscr{U}}^{must}$ leads to more precise analysis.

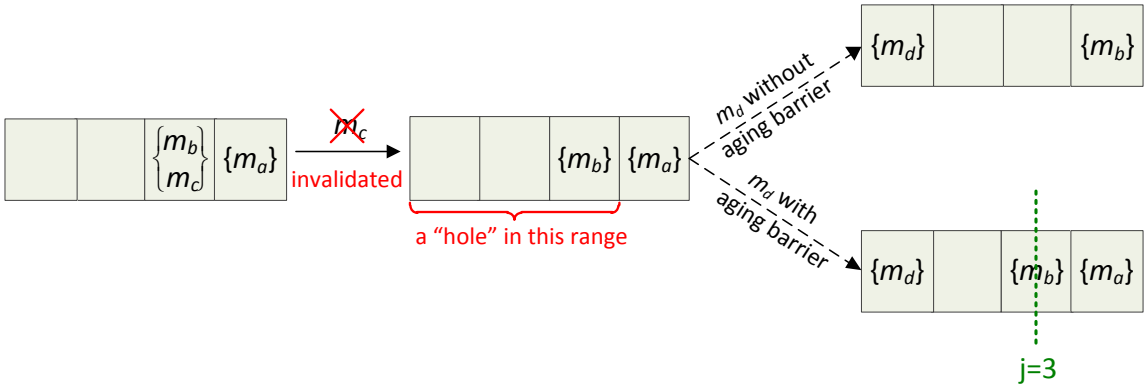


Figure 5.4: Must analysis with aging barriers

Similarly, when updating α^{pers}, given an aging barrier (i, j) and the k which is the affected position range upper bound when applying the normal $\mathscr{U}^{pers}$, if we have $k \leq j$, we simply perform the normal $\mathscr{U}^{pers}$; otherwise, we know in any concrete state of C there will be a "hole" in the i^{th} cache set within the position range $[1, j]$, so we can take advantage of this information to carry out a more precise update. We redefine the update function $\overline{\mathscr{U}}^{pers} : ACS^{pers} \times M \times \{1, \cdots, A, \top\} \rightarrow ACS^{pers}$ for the *persistence* analysis. When performing $\overline{\mathscr{U}}^{pers}(\alpha^{pers}, m, j)$, if we have $j < k$, for the memory blocks whose maximal ages are already greater than or equal to j in the i^{th} abstract set state, their ages will not be increased (but one of them may be decreased to 1 if that block is the referenced one).

We maintain an aging barrier state for each cache which is located above at least one inclusive cache level so as to achieve more precise analysis (described in the next subsection).

5.3.4 Cache Analysis above One Inclusive Cache Level

When the last inclusive cache analysis is completed, we move up to the second last inclusive level if there is any; otherwise, we start from the first cache level l_1 and move down to analyze the non-inclusive caches. No matter which level l_x (where $1 \leq x < \text{LIC}$) we are going to analyze, this level is located above at least one inclusive cache level (i.e.

the last inclusive cache level l_{LIC}), so the cache at this level may suffer from invalidations caused by the underlying inclusive cache(s).

When we analyze C_{l_x} with respect to the CFG of the program, at a join point, given the abstract cache states $\alpha^t_{l_x,1}, \alpha^t_{l_x,2}$ of the exit points of two predecessors, where t is either *may*, *must*, or *persistence*, we can simply perform $\mathscr{J}^t(\alpha^t_{l_x,1}, \alpha^t_{l_x,2})$ to safely join the abstract cache states. However, at a program point in a basic block where r is the reference that is going to occur, we need to take into account the invalidation behavior to safely update the abstract cache state of the corresponding analysis.

In order to facilitate the presentation, let us assume l_y is the uppermost inclusive level that includes l_x, and all the abstract states (i.e. $\alpha^{may}_{l_x}$ $\alpha^{must}_{l_x}$, $\alpha^{pers}_{l_x}$, and β_{l_x}) and all the arguments (e.g. the number of cache sets s_{l_x} and the associativity A_{l_x}) at the cache level l_x are the attributes of C_{l_x}.

Since we first analyze the inclusive caches in the *bottom-up* direction, the analyses of C_{l_y} are already completed at the time of analyzing C_{l_x}, and these analyses of C_{l_y} have captured the possible invalidations caused by the inclusive levels lower than l_y if there are any. Thus, from $\alpha^{may}_{l_y}$, we can deduce whether the contents of a memory block are definitely absent from C_{l_y}, and from $\alpha^{pers}_{l_y}$, we can deduce whether the contents of a memory block are possibly absent from C_{l_y}. Thus, we only need to check l_x against l_y and not any other lower inclusive cache levels.

5.3.4.1 May Analysis

As described in [64], it is unsafe to update the abstract cache state $\alpha^{may}_{l_x}$ without considering the possible invalidations caused by its underlying inclusive levels, since there possibly exist some "holes" so that some memory blocks at l_x may live longer. Fortunately, since we first analyze all the inclusive caches in the *bottom-up* direction, when we analyze C_{l_x}, the invalidation behavior induced by its underlying inclusive levels has already become visible.

First, let us redefine the update function $\overline{U}^{may} : ACS^{may} \times M \times \{1, \cdots, A, \top\} \rightarrow ACS^{may}$ for the *may* analysis of C_{l_x} that is located above the inclusive level l_y. Similar to the $\overline{\mathscr{U}}^{must}$ and $\overline{\mathscr{U}}^{pers}$ described in 5.3.3, given a memory reference r, in the $\overline{\mathscr{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j)$, j controls the upper bound on the aging process. However, different from the $\overline{\mathscr{U}}^{must}$ and $\overline{\mathscr{U}}^{pers}$, where j is given by an aging barrier, here j is decided by finding the *youngest* position in which there is a possible inclusion victim (i.e. there is *possibly* a "hole" within the range $[j, A_{l_x}]$ if such a j can be found). Thus, if we have $j = \top$, we just perform the normal $\mathscr{U}^{may}$; otherwise, for the memory blocks whose ages are already greater than or equal to j, their ages will not be increased (but may be decreased to 1 by the reference). The steps to update $\alpha_{l_x}^{may}$ are given in Algorithm 5.1.

Algorithm 5.1: Update $\alpha_{l_x}^{may}$ above an inclusive level l_y

Input: r, l_x, l_y
Result: updated $\alpha_{l_x}^{may}$
1 $i \leftarrow m_{l_x}^r$ mapped set number;
2 $j \leftarrow \top$;
3 $k \leftarrow 1$;
4 **for** $j = \top \wedge k \leq A_{l_x}$ **do**
5 **if** *the contents of a memory block* $m_{l_x} \in \alpha_{l_x}^{may}(i)(k)$ *are possibly evicted according to* $\alpha_{l_y}^{pers}$ *after* r **then** $j \leftarrow k$;
6 **else** $k \leftarrow k + 1$;
7 **if** l_x *is inclusive* **then**
8 **if** $m_{l_x}^r \notin \alpha_{l_x}^{may}(i)$ **then** $\alpha_{l_x}^{may} \leftarrow \overline{\mathscr{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j)$;
9 **else** $\alpha_{l_x}^{may} \leftarrow \mathscr{J}^{may}(\overline{\mathscr{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j), \alpha_{l_x}^{may})$;
10 **else**
11 get CAC for r at l_x from CHMC and CAC at l_{x-1};
12 **if** CAC *is always* **then** $\alpha_{l_x}^{may} \leftarrow \overline{\mathscr{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j)$;
13 **else if** CAC *is never* **then** $\alpha_{l_x}^{may} \leftarrow \alpha_{l_x}^{may}$;
14 **else** $\alpha_{l_x}^{may} \leftarrow \mathscr{J}^{may}(\overline{\mathscr{U}}^{may}(\alpha_{l_x}^{may}, m_{l_x}^r, j), \alpha_{l_x}^{may})$;

The first loop (line 4-6) checks whether there is a memory block m_{l_x} whose contents are in a block located in a $\top$ position of $\alpha_{l_y}^{pers}$ after the reference r (i.e. $\alpha_{l_y}^{pers}$ has taken into account the effect of the reference), namely it checks if m_{l_x} is a sub-block of a possibly evicted memory block due to the reference at l_y. If there is such a block found in a position $k \leq A_{l_x}$, increasing the ages of the memory blocks which are not less than k may make the *may* analysis unsafe (since there may be a "hole" within the range $[k, A_{l_x}]$), so we set j as

the *youngest k*; otherwise, j is $\top$.

If l_x is an inclusive level (line 7-9), we are still moving up in the cache hierarchy, so it is not possible to decide the access occurrence by using the traditional CAC method. Therefore, like in the last inclusive cache analyses, the algorithm checks against itself (i.e. $\alpha_{l_x}^{may}$) to find out if the memory block $m_{l_x}^r$ referenced by r is possibly in the cache. If not, this inclusive level will be definitely accessed, so we update $\alpha_{l_x}^{may}$ directly; otherwise, we have to safely update $\alpha_{l_x}^{may}$ by taking into account the two cases (i.e. access occurring and not occurring). If l_x is a non-inclusive level (line 10-14), we have already analyzed all the inclusive levels and are moving down in the cache hierarchy. Therefore, no matter which type l_{x-1} is, where $x > 1$ (when l_x is l_1, it is always accessed), the analyses of $C_{l_{x-1}}$ have been completed, so it is possible to derive the CAC for r at l_x from the CHMC and CAC for r at l_{x-1}, and then to update the $\alpha_{l_x}^{may}$ according to the derived CAC.

The last step is to update $\alpha_{l_x}^{may}$ by removing all the memory blocks whose contents are definitely not in $\alpha_{l_x}^{may}$. We perform this step by referring to the contents of $\alpha_{l_y}^{may}$ at the same point, after the *may* analysis of C_{l_x} is completed (i.e. at each program point, its $\alpha_{l_x}^{may}$ has reached a fixed-point).

5.3.4.2 Must Analysis

In the *must* analysis of C_{l_x}, we maintain both the abstract cache state $\alpha_{l_x}^{must}$ and the aging barrier state β_{l_x}. As we discussed above, at a join point, we simply perform $\mathscr{J}^{must}(\alpha_{l_x,1}^{must}, \alpha_{l_x,2}^{must})$ to safely join two abstract cache states. Similarly, given two aging barrier states $\beta_{l_x,1}, \beta_{l_x,2}$, we simply perform $\mathscr{J}(\beta_{l_x,1}, \beta_{l_x,2})$ to join these two aging barrier states. At a program point in a basic block, we update the $\alpha_{l_x}^{must}$ and β_{l_x} following the steps described in Algorithm 6.1.

The loop (line 1-7) first checks whether a memory block in $\alpha_{l_x}^{must}$ is definitely an inclusion victim (i.e. the contents of the block are not in $\alpha_{l_y}^{may}$ after the reference r). If there is such a block, there will be a "hole" created by removing this block from $\alpha_{l_x}^{must}$, since it

Algorithm 5.2: Update $\alpha_{l_x}^{must}$ and β_{l_x} above an inclusive level l_y

Input: r, l_x, l_y
Result: updated $\alpha_{l_x}^{must}$, updated β_{l_x}

1 **foreach** *memory block* $m_{l_x} \in \alpha_{l_x}^{must}$ **do**
2 **if** *the contents of* m_{l_x} *are definitely evicted according to* $\alpha_{l_y}^{may}$ *after* r **then**
3 $i \leftarrow m_{l_x}$ mapped set number;
4 $j \leftarrow$ the position where m_{l_x} is in $\alpha_{l_x}^{must}(i)$;
5 $\beta_{l_x} \leftarrow \mathscr{A}(\beta_{l_x}, i, j)$;
6 remove m_{l_x} from $\alpha_{l_x}^{must}$;
7 **else if** *the contents of* m_{l_x} *are possibly evicted according to* $\alpha_{l_y}^{pers}$ *after* r **then** remove m_{l_x} from $\alpha_{l_x}^{must}$;
8 $i \leftarrow m_{l_x}^r$ mapped set number;
9 $\langle \beta'_{l_x}, j \rangle \leftarrow \mathscr{U}(\beta_{l_x}, i)$;
10 $k \leftarrow$ the position where $m_{l_x}^r$ is in $\alpha_{l_x}^{must}(i)$ ($\top$, if not found);
11 **if** l_x *is inclusive* **then**
12 **if** $m_{l_x}^r \notin \alpha_{l_x}^{may}$ **then** $\alpha_{l_x}^{must} \leftarrow \overline{\mathscr{U}}^{must}(\alpha_{l_x}^{must}, m_{l_x}^r, j)$;
13 **else** $\alpha_{l_x}^{must} \leftarrow \mathscr{J}^{must}(\overline{\mathscr{U}}^{must}(\alpha_{l_x}^{must}, m_{l_x}^r, j), \alpha_{l_x}^{must})$;
14 **else**
15 get CAC for r at l_x from CHMC and CAC at l_{x-1};
16 **if** CAC *is always* **then** $\alpha_{l_x}^{must} \leftarrow \overline{\mathscr{U}}^{must}(\alpha_{l_x}^{must}, m_{l_x}^r, j)$;
17 **else if** CAC *is never* **then** $\alpha_{l_x}^{must} \leftarrow \alpha_{l_x}^{must}$;
18 **else** $\alpha_{l_x}^{must} \leftarrow \mathscr{J}^{must}(\overline{\mathscr{U}}^{must}(\alpha_{l_x}^{must}, m_{l_x}^r, j), \alpha_{l_x}^{must})$;
19 $\beta_{l_x} \leftarrow \mathscr{A}(\beta'_{l_x}, i, \max(j,k))$;

was definitely in the cache C_{l_x} before the reference r. Thus, we add an aging barrier corresponding to this certainly invalidated block into β_{l_x} (line 3-6). In order to guarantee safety of the *must* analysis, the algorithm also (line 7) takes into account all the possibly evicted memory blocks by removing them from the $\alpha_{l_x}^{must}$.

In the next steps, we first acquire an aging barrier (i, j) by applying $\langle \beta'_{l_x}, j \rangle = \mathscr{U}(\beta_{l_x}, i)$ (line 9). Since l_x can be either inclusive or non-inclusive, line 11-18 take into account the two possibilities, which is similar to the corresponding steps in the *may* analysis. A valid aging barrier (i, j) (i.e. we have $j \neq \top$) means there must be a "hole" in the i^{th} cache set within the position range $[1, j]$, different from that in Algorithm 5.1 where j is chosen to be the position lower bound of a possible "hole". After updating $\alpha_{l_x}^{must}$, we update the aging barrier state by performing $\mathscr{A}(\beta'_{l_x}, i, \max(j,k))$ to add an aging barrier back to the state (line 19): (1) If we have $k \leq j$, we perform the normal update function $\mathscr{U}^{must}$, and line

19 will add the acquired aging barrier back to the aging barrier state (since we have $k \leq j$, $\max(j,k)$ is always j, and no matter whether j is $\top$ or not, after line 19 the β_{l_x} will be the same as the input β_{l_x}) – in the case of $j \neq \top$, the acquired aging barrier is valid, since we have $k \leq j$, the "hole" represented by the aging barrier has not been filled yet, so after line 19, β_{l_x} becomes the same as the input β_{l_x}; in the case of $j = \top$, no valid aging barrier has been acquired from the input β_{l_x} at line 9, so β'_{l_x} was still the same as the input β_{l_x}, and after line 19, β_{l_x} is the same as β'_{l_x} as well as the input β_{l_x}. (2) If we have $j < k = \top$, it means the referenced memory block $m^r_{l_x}$ is not in the i^{th} set state of α^{must} (since $k = \top$), and the acquired aging barrier is valid (i.e. $j \neq \top$); so $m^r_{l_x}$ intends to fill the "hole" represented by this valid aging barrier; since we have $\max(j,k) = k = \top$, $\mathscr{A}(\beta'_{l_x}, i, \top)$ will not change the state β'_{l_x} which represents the valid aging barrier has already been used. (3) If we have $j < k < \top$, it means $m^r_{l_x}$ is definitely present in any concrete state, so no other memory blocks will be loaded due to this reference, and we can safely guarantee there will be a "hole" in the range $[1,k]$, even if the "hole" that was in the range $[1,j]$ has been filled; we have $\max(j,k) = k < \top$, and (i,k) is an valid aging barrier; so $\mathscr{A}(\beta'_{l_x}, i, k)$ will add the new valid aging barrier into the state β'_{l_x}.

5.3.4.3 Persistence Analysis

For the *persistence* analysis, the steps to update $\alpha^{pers}_{l_x}$ are similar to the steps in Algorithm 6.1. The differences are: (i) We set j according to the aging barrier state β_{l_x} maintained by the *must* analysis of C_{l_x}, but we do not change β_{l_x} in the steps, namely we only use the fact that if there is a valid aging barrier available before executing the reference, there is a "hole" within the position range $[1,j]$; (ii) We do not remove memory blocks from $\alpha^{pers}_{l_x}$, but for any memory block in $\alpha^{pers}_{l_x}$ which is not in the $\top$ position yet, if its contents are not in $\alpha^{may}_{l_y}$ after the reference r or its contents are in a $\top$ position of $\alpha^{pers}_{l_y}$ after the reference r, move it to the corresponding set's $\top$ position.

There can also be some non-inclusive caches located below the last inclusive cache

level, but they do not suffer from any invalidation. When moving down in the cache hierarchy, the analysis of any of them is the same as the traditional multi-level non-inclusive cache analysis. Theoretical analysis of the approach's **safety** and **termination** is provided in the next section.

5.4 Theoretical Analysis of Safety and Termination

In order to prove that the proposed multi-level (inclusive) cache analysis is safe, we need to prove the *may*, *must*, and *persistence* analyses of the last inclusive cache are safe, and we also need to prove the analyses of the cache located above at least one inclusive cache are safe.

When analyzing a cache level, we can safely use the well-defined join function of the single-level cache *may*, *must*, or *persistence* analysis at a join point for the corresponding analysis [61], so we can focus more on proving the defined update functions are safe.

5.4.1 Safe Analyses of the Last Inclusive Cache

Given the last inclusive cache level l_{LIC}, we first prove the proposed *may*, *must*, and *persistence* analyses are safe.

Lemma 5.4.1. *The last inclusive cache may analysis is safe. In other words, at a program point p, $\alpha_{l_{\text{LIC}}}^{may}$ contains all of the memory blocks that are possibly in $C_{l_{\text{LIC}}}$ when the execution reaches p.*

Proof. Since $\mathscr{J}_{\text{LIC}}^{may}$ is $\mathscr{J}^{may}$ which is safe, we only need to prove $\mathscr{U}_{\text{LIC}}^{may}$ is safe, which we do by mathematical induction.

Base case: At the beginning of any execution, $C_{l_{\text{LIC}}}$ does not have any valid blocks (cold start), and the $\alpha_{l_{\text{LIC}}}^{may}$ is also empty showing no memory block is possibly in $C_{l_{\text{LIC}}}$.

Inductive hypothesis: Before a reference r which accesses the memory block $m_{l_{\text{LIC}}}^{r}$, $\alpha_{l_{\text{LIC}}}^{may}$ contains all the memory blocks that are possibly in $C_{l_{\text{LIC}}}$.

Inductive step: When executing the reference r, we have two possibilities. (1) If $m^r_{l_{\text{LIC}}} \notin \alpha^{may}_{l_{\text{LIC}}}$, $m^r_{l_{\text{LIC}}}$ is definitely not in $C_{l_{\text{LIC}}}$ (deduced from the inductive hypothesis). Based on Lemma 5.3.1, we know that $C_{l_{\text{LIC}}}$ will be definitely accessed. Therefore, $\mathscr{U}^{may}(\alpha^{may}_{l}, m^r_{l_{\text{LIC}}})$ gives the safe result. (2) If $m^r_{l_{\text{LIC}}} \in \alpha^{may}_{l_{\text{LIC}}}$, $m^r_{l_{\text{LIC}}}$ is possibly (may or may not be) in $C_{l_{\text{LIC}}}$ (given by the inductive hypothesis), so it is uncertain whether $C_{l_{\text{LIC}}}$ will be accessed. Thus, $\mathscr{J}^{may}(\mathscr{U}^{may}(\alpha^{may}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}}), \alpha^{may}_{l_{\text{LIC}}})$ captures this uncertainty and gives the safe result. Combing (1) and (2), we conclude Lemma 5.4.1 holds. □

Lemma 5.4.2. *The last inclusive cache must analysis is safe. In other words, at a program point p, the memory blocks that are contained in $\alpha^{must}_{l_{\text{LIC}}}$ are definitely in $C_{l_{\text{LIC}}}$ when the execution reaches p.*

Proof. Since $\mathscr{J}^{must}_{\text{LIC}}$ is $\mathscr{J}^{must}$ which is safe, we only need to prove $\mathscr{U}^{must}_{\text{LIC}}$ is safe. As shown in the definition of $\mathscr{U}^{must}_{\text{LIC}}$, for a memory reference r, only when $m^r_{l_{\text{LIC}}} \notin \alpha^{may}_{l_{\text{LIC}}}$, we directly use $\mathscr{U}^{must}(\alpha^{must}_{l_{\text{LIC}}}, m^r_{l_{\text{LIC}}})$; otherwise, we conservatively join the two states coming from two possibilities (the access occurring and not occurring). Thus, as long as when $m^r_{l_{\text{LIC}}} \notin \alpha^{may}_{l_{\text{LIC}}}$, $C_{l_{\text{LIC}}}$ will be definitely accessed, the update function $\mathscr{U}^{must}_{\text{LIC}}$ is safe. From Lemma 5.4.1, it is straightforward to see that this is true. □

Lemma 5.4.3. *The last inclusive cache persistence analysis is safe. In other words, at a program point p, any memory block that has been loaded into $C_{l_{\text{LIC}}}$ is in an age position of $\alpha^{pers}_{l_{\text{LIC}}}$ which is greater than or equal to its possible maximal age when the execution reaches p (which implies if it is possibly absent from $C_{l_{\text{LIC}}}$, it is in a $\top$ position of $\alpha^{pers}_{l_{\text{LIC}}}$).*

Proof. This proof will be the same as the proof of Lemma 5.4.2, except we prove the defined $\mathscr{U}^{pers}_{\text{LIC}}$ is safe. □

5.4.2 Safe Analyses of Inclusive Caches Located above One Inclusive Cache

Since we analyze all the inclusive caches in the *bottom-up* direction at first, we prove the analyses of the inclusive caches that are located above the last inclusive cache are safe.

Let l_v be the second last inclusive cache level.

Lemma 5.4.4. *The may analysis of C_{l_v} is safe. In other words, at a program point p, $\alpha_{l_v}^{may}$ contains all the memory blocks that are possibly in C_{l_v} when the execution reaches p.*

Proof. As $\alpha_{l_v}^{may}$ is updated according to Algorithm 5.1, we need to prove the steps in the algorithm will not overestimate the age of a memory block. In the algorithm, j is calculated and used to control the upper bound on the aging process of updating $\alpha_{l_v}^{may}$. Note that if we have $j \leq j'$, where j' represents the smallest position where has a "hole", line 7-9 will be always safe (some blocks' ages will be underestimated but will not be overestimated). Based on Lemma 5.4.3, we know the last inclusive cache *persistence* analysis captures all the possibly evicted memory blocks in the $\top$ positions of $\alpha_{l_{\text{LIC}}}^{pers}$. Thus, line 4-6 will give a j such that $j \leq j'$ holds.

When $\alpha_{l_v}^{may}$ of each point reaches the fixed-point, we also remove the memory blocks from $\alpha_{l_v}^{may}$ whose contents are not in the $\alpha_{l_{\text{LIC}}}^{may}$ of that point. Based on Lemma 5.4.1, we know if a memory block is not in $\alpha_{l_{\text{LIC}}}^{may}$, it is definitely not in the last inclusive cache, so its contents are also invalidated at l_v. Thus, $\alpha_{l_v}^{may}$ is safely derived at each point, and Lemma 5.4.4 holds. □

Lemma 5.4.5. *The must analysis of C_{l_v} is safe. In other words, at a program point p, any aging barrier (i, j) derived from β_{l_v} corresponds to a "hole" in the i^{th} set within the position range of $[1, j]$, and the memory blocks contained in $\alpha_{l_v}^{must}$ are definitely in C_{l_v} when the execution reaches p.*

Proof. As discussed in 5.3.2 concerning the definition of $\mathscr{J}$ function, we know the $\mathscr{J}$ function ensures only the "holes" that definitely exist along either path are kept and the function overestimates the position upper bounds of these "holes". Since the join function $\mathscr{J}^{must}$ does not underestimate the age of a memory block, we only need to prove updating β_{l_v} and $\alpha_{l_v}^{must}$ are safe. We prove this by mathematical induction.

Base case: At the beginning, $\beta_{l_v} = \beta_{\perp}$, which means all the positions in all sets are "holes", and $\alpha_{l_v}^{must}$ corresponds to an empty state. We have a cold start, there is no memory blocks loaded. Therefore, the lemma holds in the base case.

Inductive hypothesis: Before a reference r which accesses the memory block $m_{l_v}^r$ that is mapped to the i^{th} cache set, any aging barrier (i, j) derived from β_{l_v} corresponds to a "hole" in the i^{th} set within the position range of $[1, j]$, and the memory blocks contained in $\alpha_{l_v}^{must}$ are definitely in C_{l_v}.

Inductive step: Based on the inductive hypothesis and Lemma 5.4.1, if a memory block is in the current $\alpha_{l_v}^{must}$, but its contents are not in $\alpha_{l_{\text{LIC}}}^{may}$ after the reference, this memory block needs to be invalidated, so a "hole" will be created. Since the *must* analysis captures the maximal ages of memory blocks, adding the created "hole" into β_{l_v} will not violate the lemma. Based on Lemma 5.4.3, the memory blocks in the $\top$ positions of $\alpha_{l_{\text{LIC}}}^{pers}$ after the reference are possibly evicted; thus, after line 7 the lemma still holds with respect to the updated β_{l_y} and $\alpha_{l_y}^{must}$. When updating the states according to the rest of Algorithm 6.1, after line 9, j has a position and if we have $j \neq \top$, there is a "hole" within the range $[1, j]$, and any of the rest aging barriers derived from the used β_{l_v}, namely β'_{l_v}, still corresponds to a "hole" (deduced from the inductive hypothesis). There are two possibilities when updating $\alpha_{l_v}^{must}$. (1) If $m_{l_v}^r \notin \alpha_{l_v}^{may}$, based on Lemma 5.4.4, we have $k = \top$ and we are sure that $m_{l_v}^r$ is not in C_{l_v}. Based on Lemma 5.3.1, C_{l_v} will be definitely accessed due to the reference r. Therefore, line 16 (i.e. applying $\overline{\mathscr{U}}^{must}$ which takes into account the effects of the existence of a "hole") can safely update $\alpha_{l_v}^{must}$, and that "hole" is possibly filled. In this case, $\max(j,k) = \top$ no matter what j is, so $\mathscr{A}$ will not change the β'_{l_v} at line 19. (2) If $m_{l_v}^r \in \alpha_{l_v}^{may}$, we do not know if C_{l_v} will be accessed or not, so line 17 can safely update $\alpha_{l_v}^{must}$ by taking into account the access occurring and not occurring. We have $j = \top$ or $j \neq \top$, and $k = \top$ or $k \neq \top$. If $j = \top$ or $k = \top$, $\max(j,k) = \top$, so $\mathscr{A}$ will not change the β'_{l_v} at line 19. The only case in which $\mathscr{A}$ will change β'_{l_v} is when $j \neq \top \bigwedge k \neq \top$. In this case, although the "hole" with the range $[1, j]$ may be possibly filled, there is still a "hole" within

the range $[1, \max(j,k)]$ – this is because, based on the hypothesis, $m^r_{l_v}$ is definitely in C_{l_v} if $k \neq \top$, and in either case of $k < j$ or $j < k$, the reference does not load a new memory block into C_{l_v}; so after applying $\mathscr{A}$ on β'_{l_v}, the resultant β_{l_v} does not violate the lemma. Thus, after line 19, this lemma still holds with respect to the updated β_{l_v} and $\alpha^{must}_{l_v}$. □

Lemma 5.4.6. *The persistence analysis of C_{l_v} is safe. In other words, at a program point p, any memory block that has been loaded into C_{l_v} is in an age position of $\alpha^{pers}_{l_v}$ which is greater than or equal to its possible maximal age.*

Proof. Since $\mathscr{J}^{pers}$ does not underestimate the age of a memory block, we only need to prove this lemma holds in terms of updating, which we do by mathematical induction.

Base case: At the beginning, no memory block is loaded, and all the positions of $\alpha^{pers}_{l_v}$ are empty. The lemma holds.

Inductive hypothesis: Before a reference r which accesses the memory block $m^r_{l_v}$, any memory block that has been loaded into C_{l_v} is in an age that is greater than or equal to its possible maximal age.

Inductive step: When updating $\alpha^{pers}_{l_v}$, we first move the blocks which are possibly or definitely invalidated to $\top$ positions according to $\alpha^{pers}_{l_{\text{LIC}}}$ and $\alpha^{may}_{l_{\text{LIC}}}$ after the reference. Since doing so does not decrease any block's age, the lemma still holds. Then, we get j from the aging barrier state β_{l_v} maintained by the *must* analysis (but we do not change β_{l_v}). There are two possibilities to continue updating. (1) If $m^r_{l_v} \notin \alpha^{may}_{l_v}$, based on Lemma 5.4.4, we know $m^r_{l_v}$ is not in C_{l_v}; and based on Lemma 5.3.1, C_{l_v} will be accessed due to the reference r. According to the definition of $\overline{\mathscr{U}}^{pers}$, when $j = \top$, it is $\mathscr{U}^{pers}$ and it will not underestimate the possible maximal ages of the blocks; when $j \neq \top$, no matter what k is, it will never increase the ages of the blocks that are already greater than or equal to j, so we need to prove in this case the possible maximal ages of these memory blocks are actually not greater than these unchanged ages: since $j \neq \top$, there is definitely a "hole" within the position range $[1, j]$, so even C_{l_v} is accessed and $m^r_{l_v}$ is not in C_{l_v}, the ages of the blocks that are behind this "hole" will not be increase, which means the possible maximal ages

of the memory blocks which are already greater than or equal to j will not be increase; from the inductive hypothesis, we know that before this reference, for a memory block, the position where it is in $\alpha_{l_v}^{pers}$ is the upper bound of its possible maximal age position; thus, even though the ages of the memory blocks that are already greater than or equal to j are unchanged after applying $\overline{\mathscr{U}}^{pers}$, they are still not less than the possible maximal ages of these memory blocks, based on the arguments above. (2) If $m_{l_v}^r \in \alpha_{l_v}^{may}$, we do not know if C_{l_v} is accessed or not, so we safely join the two states corresponding to the access occurring and not occurring. Thus, the lemma holds with respect to the updated $\alpha_{l_v}^{pers}$. □

Theorem 5.4.7. *The proposed may, must, and persistence analyses of the inclusive caches in the bottom-up direction are safe.*

Proof. Since we have proven the analyses of the last inclusive cache are safe (Lemma 5.4.1, Lemma 5.4.2, and Lemma 5.4.3), we only need to prove the analyses of the rest inclusive caches in the *bottom-up* direction are safe by mathematical induction.

Base case: The analyses of C_{l_v} are safe, where l_v is the second last inclusive cache level.

Inductive hypothesis: The analyses of all the inclusive caches that are located beneath C_{l_y} are safe, where l_y is an inclusive level above the last inclusive level l_{LIC}.

Inductive step: Let us assume the next inclusive level located beneath l_y in the *top-down* direction is l_y^i. Following the proofs of Lemma 5.4.4, Lemma 5.4.5, and Lemma 5.4.6, we can prove the *may*, *must*, and *persistence* analyses of C_{l_y} are safe, as long as the analyses of $C_{l_y^i}$ are safe. Since the inductive hypothesis gives the analyses of $C_{l_y^i}$ are safe, the analyses of C_{l_y} are safe. □

5.4.3 Safe Analyses of Non-Inclusive Caches

After the analyses of the inclusive caches are completed, we start from l_1 and analyze all the non-inclusive caches in the *top-down* direction. Let us assume, for a non-inclusive cache level l_z, l_z^p is the previous cache level in the *top-down* direction in the cache hierarchy

when $z > 1$ (l_z^p can be either inclusive or non-inclusive), and l_z^i is the first inclusive cache level that is beneath l_z if there is one (i.e. $C_{l_z^i}$ directly includes C_{l_z}).

Lemma 5.4.8. *The may analysis of C_{l_1} is safe. In other words, at a program point p, $\alpha_{l_1}^{may}$ contains all the memory blocks that are possibly in C_{l_1} when the execution reaches p.*

Proof. Since C_{l_1} is always accessed for a reference, we just need to prove line 12 can safely update $\alpha_{l_1}^{may}$ each time, which implies we prove j should always satisfy $j \leq j'$, where j' represents the smallest position where has a "hole" (in which case, some blocks' ages will be underestimated but will never be overestimated). Following the proof of Lemma 5.4.4, we can see j set by the loop (line 4-6) satisfies $j \leq j'$, since the *persistence* analysis of $C_{l_1^i}$ has already been safely performed (given by Theorem 5.4.7). Thus, Lemma 5.4.8 holds. □

Lemma 5.4.9. *The must analysis of C_{l_1} is safe. In other words, at a program point p, any aging barrier (i, j) derived from β_{l_1} corresponds to a "hole" in the i^{th} set within the position range of $[1, j]$, and the memory blocks contained in $\alpha_{l_1}^{must}$ are definitely in C_{l_1} when the execution reaches p.*

Proof. Following the proof of Lemma 5.4.5, we can prove the lemma holds by using mathematical induction. The difference is: since C_{l_1} is always accessed for a reference, we only use line 16 to update $\alpha_{l_1}^{must}$ each time. Following that proof, we can prove it safely updates β_{l_1} and $\alpha_{l_1}^{must}$, as long as the *may* and *persistence* analyses of $C_{l_1^i}$ are safe, which is true based on Theorem 5.4.7. Thus, Lemma 5.4.9 holds. □

Lemma 5.4.10. *The persistence analysis of C_{l_1} is safe. In other words, at a program point p, any memory block that has been loaded into C_{l_1} is in an age position of $\alpha_{l_1}^{pers}$ which is greater than or equal to its possible maximal age.*

Proof. Similarly, following the proof of Lemma 5.4.6, we can prove this lemma holds. □

Theorem 5.4.11. *The proposed may, must, and persistence analyses of the non-inclusive caches in the top-down direction are safe.*

Proof. We prove this theorem by mathematical induction.

Base case: The analyses of C_{l_1} are safe.

Inductive hypothesis: The analyses of all of the non-inclusive caches located above C_{l_z} are safe, where l_z is a non-inclusive cache level and $z > 1$.

Inductive step: When updating the abstract cache states of C_{l_z}, we need to derive the CAC at l_z for a reference. As described in [61], for a reference, the CAC at l_z can be safely derived if the CHMC and CAC at l_z^p are known. Based on Theorem 5.4.7 and the inductive hypothesis, we know, from l_1 to l_z^p, no matter whether a level is inclusive or non-inclusive, it is safely analyzed. By taking into account the effects of filtering accesses, the CHMC and CAC at l_z^p can be safely derived, so that the CAC at l_z can be safely derived. If l_z is located beneath the last inclusive cache level l_{LIC}, C_{l_z} does not suffer from any invalidation, so the unmodified analyses of C_{l_z} will be safe. On the contrary, if l_z is located above l_{LIC}, we use the methods described in 5.3.4 to analyze C_{l_z}; following the previous proofs, we can also prove the analyses are safe. Thus, combing these two cases, we can conclude this theorem holds. □

Theorem 5.4.12. *The proposed approach to analysis of multi-level inclusive caches is safe.*

Proof. We can directly conclude this Theorem holds based on Theorem 5.4.7 and Theorem 5.4.11. □

5.4.4 Termination of the Analysis

In order to prove the proposed multi-level inclusive cache analysis will terminate, we need to prove the aging barrier state domain *ABS* is a partially ordered set with a finite height; and we need to prove the joining and updating of the aging barrier states are monotonic at a program point during the iterations at one level. Since the number of sets and the associativity of a cache are both finite, based on the Definition 5.3.3 and Definition 5.3.4, it is trivial to see *ABS* is finite and partially ordered. Also, we have seen the join function

$\mathscr{J}$ is monotone with respect to the partial ordering defined in Definition 5.3.4. Thus, we need to prove the aging barrier state updating is also monotone.

Lemma 5.4.13. *Given an aging barrier state β' which is updated by a reference from β, $\beta \sqsubseteq \beta'$ always holds.*

Proof. When updating β, first we have $\langle \beta'', j \rangle = \mathscr{U}(\beta, i)$, where i is fixed for a reference in a cache. Therefore, we have $\beta \sqsubseteq \beta''$ according to the definition of $\mathscr{U}$. Then, we have $\beta' = \mathscr{A}(\beta'', i, \max(j,h))$. According to the definition of $\mathscr{A}$: if $\max(j,h) = \top$, we have $\beta' = \beta''$, so $\beta \sqsubseteq \beta'' = \beta'$ holds; if $\max(j,h) = j \neq \top$, we have $\beta' = \beta$, since the partial ordering $\sqsubseteq$ is reflexive, so $\beta \sqsubseteq \beta'$ holds; if $\max(j,h) = h \neq \top$, we have $|\beta'(i)| = |\beta(i)|$ and since the only difference between $\beta(i)$ and $\beta'(i)$ is in $\beta'(i)$ we have $j = \min_c(\beta(i))$ replaced by h which is $\max(j,h)$, so $\beta \sqsubseteq \beta'$ holds. □

Theorem 5.4.14. *The proposed multi-level inclusive analysis approach terminates in finite iterations at each level.*

Proof. Since the analyses of the last inclusive cache are not affected by other factors, they will terminate. The analyses of a cache located above at least one inclusive level are affected by its aging barrier state and the abstract cache states of some safely analyzed caches. Although aging barriers can slow down the age increasing, the abstract cache states at this level are still updated along an ascending chain. Based on Lemma 5.4.13, we know the aging barrier states are also updated along an ascending chain. Since all the domains are finite partially ordered sets, the proposed analysis will terminate. □

5.5 Evaluation

The objective of this chapter is to tighten the WCET estimation in the presence of inclusive caches. We evaluate the proposed approach and compare with the approach proposed in [64]. The experiments are performed using our research prototype tool, which is described in Appendix A.

Due to the limitations of our current tool, we only take into account the timing effects of multi-level caches on the WCET estimation and do not consider the effects of other micro-architectural components like pipelines and branch predictors, so we assume there are no timing anomalies. Therefore, a reference that is classified as *NC* can be safely treated as a *AM* when used to estimate the WCET. However, if the timing anomalies are considered, we will gain more precision using the proposed approach, since it can safely classify some references as *AM* compared to the approach in [64]. We leave this as future work.

Our experiments are carried out on the set of benchmarks maintained by the Mälardalen WCET research group [114], and they are compiled for MIPS R3000 processor using gcc-3.4.4. Since the approach proposed in [64] only considers strict multi-level inclusive caches (i.e. it does not consider mixed inclusive and non-inclusive cache levels), we carry out the experiments on a three-level cache hierarchy and configure L2 and L3 to be inclusive. The parameters of the cache at each level are shown in Tab. 6.1. Moreover, we assume every needed information can be found in the main memory with a 200-cycle latency.

Table 5.1: 3-Level Inclusive Cache Parameters

Level	Cache Capacity	Block Size	Associativity	Latency
L1	2KB	8B	4-way	1-cycle
L2	8KB	32B	8-way	10-cycle
L3	16KB	64B	8-way	80-cycle

The experimental results are shown in Tab. 5.2. For a benchmark, $\text{WCET}_{\text{top-dw}}$ is derived by using the method proposed in [64], and $\text{WCET}_{\text{bot-up}}$ is derived by using the method proposed in this chapter. The WCET estimation is reported in clock cycles. The precision improvement is calculated by $\frac{\text{WCET}_{\text{top-dw}}}{\text{WCET}_{\text{bot-up}}} - 1$. We also report the computation time overhead in seconds, along with the reported WCET. The experiments are performed on a Linux machine with a $1.2GHz$ quad-core processor and $12GB$ memory.

We sort Tab. 5.2 in descending order of the precision improvement. From the results, we can see that the bound can be tightened about 12.2% on average. In some cases, the improvement is more than 20%, e.g. up to 57.3% is gained in the case of *fibcall* and up to

44.4% is gained in the case of *insertsort*. For some benchmarks, the improvement rate is not that substantial (less than 3%), e.g. only 2.7% is gained in the case of *ludcmp* and only 2.4% is gained in the case of *adpcm*. We find most of these benchmarks contain nested loops and/or are context-sensitive. The advantage of the proposed method may become larger if the *persistence* analysis is multi-leveled to handle the nested loops [120] and contexts are taken into account in the inter-procedural analysis. Furthermore, as mentioned above, our prototype tool does not analyze other micro-architectural features than multi-level caches for the present. Since the proposed approach can classify some references as *AM* while the method in [64] cannot, we would expect more precision gains if timing anomalies are considered. Although these techniques are not integrated in our tool yet, the improvement is still significant. Even in some cases the improvement rate is less than 3%, thousands of overestimated cycles are reduced (e.g. up to 12400 clock cycles are reduced in the case of *adpcm*). However, it should be noted that the proposed approach is standalone and can be integrated with other techniques without any changes.

From the results, we can see the computation time overhead differences between the two methods are within a few seconds in most cases. The biggest difference is about 93 seconds in the case of *nsichneu*. Since this difference is just a small portion of the overheads, which are 6.4 and 7.9 minutes respectively, we believe the computation time overhead is acceptable.

5.6 Conclusion and Future Work

In this chapter, we propose an approach that can safely and more precisely analyze multi-level inclusive caches for WCET estimation. The approach first analyzes all the inclusive levels in the *bottom-up* direction and then analyzes the rest non-inclusive levels in the *top-down* direction. Although *bottom-up* sounds counter-intuitive considering the cache levels are accessed in the *top-down* direction, we show that it is actually very suitable for analyzing inclusive caches. In order to capture the effects of the invalidations caused by an

Table 5.2: Experiment Results of Estimated WCET and Computation Time Overhead

Benchmark	Method in [64]		Method in This chapter		Precis. Improv.
	Ovhd.	$WCET_{top\text{-}dw}$	Ovhd.	$WCET_{bot\text{-}up}$	
fibcall	0	7250	0	4610	57.3%
insertsort	0	18349	0	12709	44.4%
recursion	0	6942	0	4982	39.3%
bs	0	10979	0	8579	28.0%
fir	0	28046	1	22406	25.2%
sqrt	0	19662	0	15742	24.9%
janne_cmplx.	0	11367	0	9247	22.9%
cnt	0	43138	1	35778	20.6%
ns	0	45731	0	38131	19.9%
duff	0	23169	1	19609	18.2%
prime	0	31690	0	27890	13.6%
edn	3	303483	4	272123	11.5%
expint	0	35855	0	32775	9.4%
qurt	0	41122	1	37922	8.4%
statemate	17	404050	31	377550	7.0%
lcdnum	0	18939	0	17819	6.3%
fdct	1	92089	1	88329	4.3%
minver	5	111053	5	106533	4.2%
select	3	63744	3	61344	3.9%
compress	13	299514	14	288514	3.8%
cover	9	187579	10	182259	2.9%
ludcmp	3	87526	3	85206	2.7%
qsort_exam	3	69903	5	68063	2.7%
adpcm	41	522619	42	510219	2.4%
ndes	10	737997	11	728637	1.3%
bsort100	0	287104	1	281904	1.8%
st	6	380532	5	374572	1.6%
jfdctint	1	99865	1	98465	1.4%
matmult	0	513672	0	508032	1.1%
crc	1	95794	0	95274	0.5%
lms	5	1226776	6	1221496	0.4%
nsichneu	383	2985648	476	2985088	0.02%
average					12.2%

inclusive level, we propose a concept of aging barrier. Aging barriers can safely slow down the increase of memory blocks' ages, and we show how to integrate them into the *must* and *persistence* analyses to gain more precision. From the experiment results, we can observe the proposed approach can tighten the bound by 12.2% on average. In the future, we want

to extend the approach to take into account the effects of data references and inter-core interferences, and we also want to enhance our tool to consider the interactions between multi-level caches and other micro-architectural features.

Chapter 6

Precise Multi-Level Inclusive Cache Analysis for WCET Estimation

Most of the current approaches attempt to separately analyze each cache level of a cache hierarchy, usually starting from the topmost level and moving downward. For multi-level non-inclusive caches, this analysis style does not have a large impact on the precision, since the interactions between different cache levels in such a hierarchy are only related to the memory access filtering behavior which appears in a *top-down* direction. However, for multi-level inclusive caches, such an analysis style may have a big influence on the precision, because some blocks may be invalidated by lower inclusive levels (i.e., the invalidation behavior appears in a *bottom-up* direction). Due to the unknown underlying invalidation behavior, when analyzing an upper level, conservative decisions have to be made in order to ensure safety [64]. In order to reduce the number of conservative decisions, some approaches are proposed to separately analyze all the inclusive levels in a *bottom-up* direction (i.e. Chapter 5). However, as shown in this chapter, this *bottom-up* approach may not perform well when the cache size is relatively small compared to the program size.

In this chapter, we propose an approach that can precisely analyze multi-level inclusive caches, even in the case that the ratio of cache size to program size is low. The main idea is to analyze a multi-level inclusive cache as a whole following its concrete behavioral semantics, instead of analyzing it in a level-by-level manner.

The main contributions of this chapter are: (1) We define a concrete operational semantics which formally describes how a multi-level inclusive cache changes its state when a memory reference occurs. (2) Based on the abstract interpretation of this concrete semantics, we propose an approach that analyzes a multi-level inclusive cache as a whole by integrating three analyses (i.e., *must*, *may*, and *persistence*). (3) We evaluate the proposed approach on a set of benchmarks, and the evaluation results show significant precision

improvements when the ratio of cache size to program size is low, compared to the state-of-the-art approaches. Our approach has been published in [13].

The rest of the chapter is organized as: Section 6.1 describes the background on cache analysis; Section 6.2 gives the system model; Section 6.3 states why multi-level inclusive cache analysis is challenging; Section 6.4 presents the proposed approach to multi-level inclusive cache analysis; Section 6.5 proves the proposed approach is safe and can terminate; Section 6.6 evaluates the proposed approach; and Section 6.7 concludes this chapter.

6.1 Background

First, we introduce the terminologies and notations used in this chapter. Given a reference r to a memory block m, the effect of this reference on the ACS θ^t, where t is either *must*, *may*, or *persistence*, is defined by an *update* function $\mathscr{U}^t : \Theta^t \times M \rightarrow \Theta^t$, where Θ^t is the set of all the ACSs of the cache (i.e. $\theta^t \in \Theta^t$), and M is the set of all the memory blocks *w.r.t.* the cache block size (i.e. $m \in M$). In order to safely combine information at a join point during the analysis on the CFG, a *join* function $\mathscr{J}^t : \Theta^t \times \Theta^t \rightarrow \Theta^t$ is also defined. The definitions of the *update* and *join* functions can be found in [53, 76].

Single-level caches are always accessed by each memory reference, so we only need to consider the effects of memory reference sequences in the analysis. However, in the case of multi-level caches, it is also important to consider the effects of other cache levels, in particular, cache access filtering and memory block invalidation. For example, if we treat a possible access at a level as always happening, the analysis may become unsafe, since doing so may underestimate the set reuse distances[1] of memory blocks [61].

For a reference r, a *cache access classification* (CAC) at a cache level l is used to represent the possibility that l will be accessed [61]. Let $\theta_{l,r}^{t,i}$ denote the ACS at this level immediately before r, and let $\theta_{l,r}^{t,o}$ denote the ACS at this level immediately after r. Let m_l^r

[1] In [61], the set reuse distance between two memory references to the same block at a cache level is defined as the relative age of the memory block when the second reference occurs.

denote the memory block *w.r.t.* the cache block size at l containing the information needed by r. If the CAC is *always* (A), the access will always occur, so r will always affect the ACS:

$$\theta_{l,r}^{t,o} = \mathscr{U}^t(\theta_{l,r}^{t,i}, m_l^r)$$

On the other hand, if the CAC is *never* (N), the access will never happen, so the ACS at l is not affected by r:

$$\theta_{l,r}^{t,o} = \theta_{l,r}^{t,i}$$

If the CAC cannot be either A or N, it is *uncertain* (U), which means the access may or may not happen. In order to ensure safety, the updates of the ACS due to U accesses need to take into account the two possible cases (access occurring and not occurring) by joining them:

$$\theta_{l,r}^{t,o} = \mathscr{J}^t(\underbrace{\mathscr{U}^t(\theta_{l,r}^{t,i}, m_l^r)}_{\text{access occurring}}, \overbrace{\theta_{l,r}^{t,i}}^{\text{access not occurring}})$$

As described in [61], for a reference r that is possible to access a cache level (i.e. its CAC is not N at this level), if r can be safely classified as AH at this level, r will never need to access all the lower levels, namely its CAC is N at any lower level; if r can be safely classified as AM at this level, r is also possible to access the next lower level, namely its CAC at the next lower level is the same as the CAC at this level (i.e. A or U). Note that if a reference always/never accesses a cache level in reality, but its CAC at the level is U in the analysis, the analysis is still safe but may not give a tight result.

6.2 System Model

In this chapter, we focus on a generalized cache hierarchy model, in which the inclusion property is enforced at some cache level(s). The model has n cache levels, represented by $L = \{l_1, \cdots, l_n\}$. Each cache level is either inclusive or non-inclusive[2]. Let $inc : L \rightarrow$

[2]Note that it has no meaning for L1 cache to be inclusive/non-inclusive, i.e., it can be either one.

$\{\text{true}, \text{false}\}$ be an auxiliary function that returns true if the level is inclusive and false otherwise. The inclusive cache hierarchy model C is a two-tuple $\langle L, inc \rangle$.

Although we do not consider exclusive caches in the model, we can easily add them into our analysis by using the approach proposed in [64]. The exclusive cache levels can be collapsed by concatenating them to the end of the upper level to form a single level for the analysis, as long as they all have the same number of cache sets and the same cache block size.

For a cache level l_x (where $1 \leq x \leq n$), we assume that the cache is set associative, and LRU (Least Recently Used) replacement policy is used. The size of a cache block can be different at different cache levels, and it is assumed the block size does not increase as the level goes up. It is also assumed the capacity decreases as the level goes up.

We also assume the time to access a cache level is bounded and predictable, which can be achieved by using deterministic interconnects to connect the caches, like TDMA buses [122]. Fig. 6.1 gives an example of the model focusing on a single core and all the cache levels that can be affected by this core in a multi-core architecture.

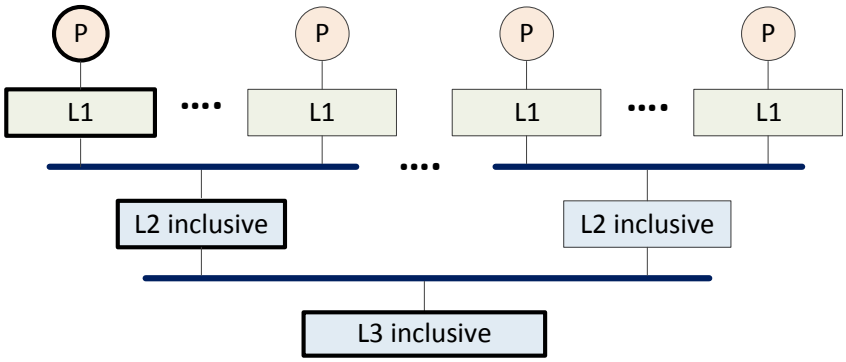


Figure 6.1: An example of the system model: only the cache levels that can be affected by the first core are considered, i.e. $L = \{l_1, l_2, l_3\}$ and $inc(l_1) = \text{false}$, $inc(l_2) = \text{true}$, and $inc(l_3) = \text{true}$.

In this chapter, we focus only on how to analyze multi-level caches in the presence of invalidations caused by the inclusion enforcement, so we simply consider instruction references in terms of a single processor (i.e. no data references and inter-core interferences).

This work can serve as a basis for analysis of multi-level data or unified caches, that enforce the inclusion property, in terms of a multi-core processor.

6.3 Problem Statement

In the case of multi-level non-inclusive cache analysis, the CAC for a reference r at a cache level l_x can be derived from the CHMC and CAC for r at l_{x-1} ($x > 1$ is assumed, since l_1 is always accessed, i.e. the CAC is A at l_1), and the cache behavior at any level will not be affected by any lower cache level. Thus, the cache hierarchy can be analyzed level-by-level in a *top-down* direction.

However, in the case of analyzing cache hierarchies containing inclusive caches, the CAC for r at l_x cannot be safely derived from CHMC and CAC for r at l_{x-1}. The reason is the behavior of l_x depends not only on the behavior of l_{x-1}, but also on the invalidation behavior induced by some lower inclusive cache level(s): When a memory block is evicted from a lower inclusive cache level, all the contents that belong to this memory block need to be invalidated from its upper cache levels (these invalidated memory blocks are called *inclusion victims*).

Example: Fig. 6.2 shows a 3-level inclusive cache, where L1 is 2-way set associative, L2 is 4-way set associative, and L3 is 4-way set associative (at each level, only one set is shown). We assume L1 has the smallest cache block size and L3 has the biggest, so a block in L1 is a sub-block of some block in L2 and that block in L2 is a sub-block of some block in L3. For a memory block m in L3, let $\dot{m}$ denote a m's sub-block in L2, and let $\ddot{m}$ denote a $\dot{m}$'s sub-block in L1. For example, we have $\ddot{m}_a \subset \dot{m}_a \subset m_a$. If the next reference needs the information that is in m_x (m_x is also mapped to the shown set of L3), the oldest m_a in that set needs to be evicted. The eviction of m_a will also invalidate $\ddot{m}_a$ in L1 and $\dot{m}_a$ in L2 to maintain the inclusion property. Due to the invalidation, $\ddot{m}_b$ in L1 can live longer, and depending on which sub-block of m_x is needed by the reference, there may be some "holes" left in L1 and L2.

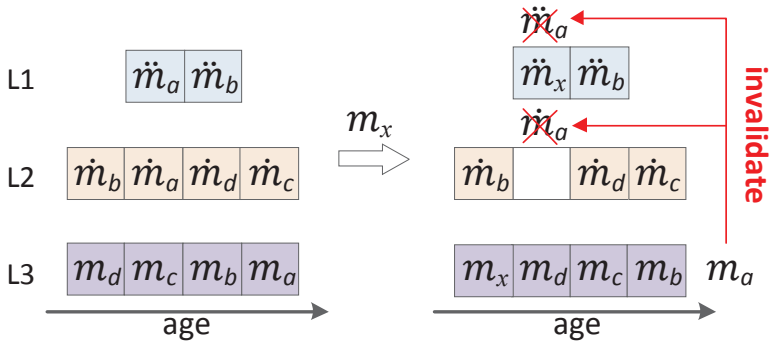


Figure 6.2: Invalidation due to the maintenance of the inclusion property of L3

Due to the possible invalidation behavior and the induced consequences (i.e., some blocks may live in the cache longer, but some blocks may live in the cache shorter – being invalidated instead of being evicted), the traditional CAC derivation method for multi-level non-inclusive cache analysis becomes unsuitable for multi-level inclusive cache analysis. At a cache level, it is challenging to safely classify a reference as *AH* or *AM* without knowing the behavior of lower inclusive cache level(s).

Since safely and tightly determining the CAC (i.e. trying to derive *A* or *N* instead of *U*) for a reference at a cache level relies on the safe CHMC at the upper levels, the approach proposed in [64] decides to classify the CAC for every reference at any level (except for the L1 which is always accessed) as *U*. In this way, although the safety is ensured, the tightness of the analysis may suffer considerably.[3]

A naive method may involve refining the ACSs of upper levels when finishing a lower inclusive level analysis, which also refines the CHMC and CAC for a reference at these levels. However, such a method may not ensure the monotonicity so that it may not guarantee termination.

The approach proposed in Chapter 5 can have the CACs for some references at some levels classified as *A* instead of *U*. However, it cannot have the CAC for any reference at the inclusive levels classified as *N*, since it analyzes all of the inclusive levels in a *bottom-*

[3]The approach proposed in [64] also changes every reference's *AM* CHMC into *NC* at any level, and may also change some references' *AH* or *PS* CHMC into *NC* at a level depending on the analysis of lower inclusive levels.

up direction (i.e. when analyzing an inclusive level, the ACSs of its upper levels remain unknown). Moreover, as the ratio of the cache size at an inclusive level to the program size decreases, its ability to safely classify the CAC as *A* instead of *U* at this level also decreases.

Thus, how to precisely analyze multi-level inclusive caches is still a very challenging problem. Specifically, we need to find ways to safely determine the *A* or *N* instead of *U* CACs for as many references as possible at a cache level.

6.4 Precise Multi-Level Inclusive Cache Analysis Based on Abstract Interpretation

The approach proposed in this chapter is based on abstract interpretation which is a framework for deriving sound analyses (i.e., the results are sound approximations) [9]. In order to make use of abstract interpretation, we first need to define the concrete semantics. The concrete semantics for single-level caches and multi-level non-inclusive caches have been given in [53] and [78], respectively. In this section, we use an operational semantics to describe how multi-level inclusive caches change their states when a reference occurs. Based on the abstraction of this concrete semantics, we propose an approach that can more precisely analyze multi-level inclusive caches due to its ability to determine *A* or *N* CACs for memory references at a cache level. Since a cache level in an inclusive cache hierarchy can be affected by two behaviors coming in two directions, i.e. the memory access filtering behavior appearing in a *top-down* direction and the invalidation behavior appearing in a *bottom-up* direction, the intuition behind the approach is to analyze a multi-level inclusive cache as a whole instead of level-by-level in isolation so as to make both behaviors available at any cache level[4].

[4]Level-by-level approaches can only make one of these two behaviors available when analyzing a cache level, so conservative decisions have to be made concerning the other unknown behavior.

6.4.1 Concrete Semantics

Since we focus only on instruction references, we just give a concrete operational semantics to the multi-level inclusive instruction caches. The semantics related to data references and dirty blocks can be added (we leave this as our future work).

The state $\bar{s}$ of a multi-level inclusive cache $\langle L, inc \rangle$ is a n-tuple (recall that $L = \{l_1, \cdots, l_n\}$). Each tuple component is a cache state s_{l_x} of the cache level l_x and they are ordered by the level numbers: $\bar{s} = \langle s_{l_1}, \cdots, s_{l_n} \rangle$.

Given a cache level l_x, let c_{l_x} denote its capacity, let b_{l_x} denote its cache block size, and let k_{l_x} denote its associativity. Therefore, the cache at this level has $d_{l_x} = \frac{c_{l_x}}{b_{l_x} \times k_{l_x}}$ cache sets. The cache state s_{l_x} of this cache level is a mapping:

$$s_{l_x} : \{1, \cdots, d_{l_x}\} \rightarrow \Big(\{1, \cdots, k_{l_x}\} \rightarrow M_{l_x} \Big)$$

where M_{l_x} is the set of all of the memory blocks *w.r.t.* b_{l_x}, and we also assume there is an element $I \in M_{l_x}$ which denotes the memory block is invalid. The cache state maps a cache set number to a cache set state and the cache set state is also a mapping that maps a logical position (ordered by LRU ages) to a memory block. For a reference r, let $m^r_{l_x} \in M_{l_x}$ denote the memory block *w.r.t.* b_{l_x} containing the information needed by r. We have an auxiliary function $set : M_{l_x} \rightarrow \{1, \cdots, d_{l_x}\}$ which gives us the cache set number to which $m^r_{l_x}$ is mapped.

When a reference r occurs, the multi-level inclusive cache carries out a sequence of actions that may change the state of a cache level intermittently. For a multi-level inclusive cache $\langle L, inc \rangle$, let $\bar{s} = \langle s_{l_1}, \cdots, s_{l_n} \rangle$ denote the cache hierarchy state before r, and let $\bar{s}' = \langle s'_{l_1}, \cdots, s'_{l_n} \rangle$ denote the state after r. The operational semantics is described as follows:

1. [Search Cache Levels] Starting from $l_x = l_1$, check whether the needed information is at l_x:

$$\exists p \in \{1, \cdots, k_{l_x}\} : s_{l_x}(set(m^r_{l_x}))(p) = m^r_{l_x}$$

If it is true (i.e. cache hit at l_x), stop searching and go to step (2). Otherwise (i.e. cache miss at l_x), if $x < n$, go to the next cache level (i.e. increment x) to continue searching; else (i.e. x is n which means the needed information is absent from the whole cache hierarchy), increment x (i.e. x becomes $n+1$ which denotes the main memory) and go to step (3).

2. [Update LRU Ages] Change the logical position (i.e. the LRU age) of $m^r_{l_x}$ from p to 1 in the cache set it is mapped to, and increment the positions of other blocks which were smaller than p. The resultant cache state s'_{l_x} is:

$$\forall i \in \{1,\cdots,d_{l_x}\} : s'_{l_x}(i) = \begin{cases} s_{l_x}(i) & \text{if } i \neq set(m^r_{l_x}) \\ [1 \mapsto m^r_{l_x}, & \\ q \mapsto s_{l_x}(i)(q-1) | q = 2 \cdots p, & \text{otherwise} \\ q \mapsto s_{l_x}(i)(q) | q = p+1 \cdots k_{l_x}] & \end{cases}$$

When the updating is completed, go to step (3).

3. [Move Upwards or Terminate] If $2 \leq x \leq n+1$, decrement x and go to step (4). Otherwise, terminate (and send the needed information to the processor).

4. [Find New Position] Find the smallest logical position p where $s_{l_x}(set(m^r_{l_x}))(p) = I$, if there is any. Otherwise, p is k_{l_x}. Go to step (5).

5. [Load Memory Block] Load the memory block $m^r_{l_x}$ into $s_{l_x}(set(m^r_{l_x}))(p)$ to replace the previous memory block $\hat{m}_{l_x}$ in that position. Go to step (6).

6. [Invalidate Memory Blocks] If $inc(l_x) =$ false or $\hat{m}_{l_x} = I$, go back to step (2). Otherwise, for all cache levels located above l_x, check if any block in their states is a

sub-block of $\hat{m}_{l_x}$, and invalidate it if so:

$$\forall y \in \{1,\cdots,x-1\}, \exists i \in \{1,\cdots,d_{l_y}\}, \exists q \in \{1,\cdots,k_{l_y}\} : s_{l_y}(i)(q) \subseteq \hat{m}_{l_x} \Rightarrow s_{l_y}(i)(q) = I$$

When the invalidation is finished, go back to step (2).

Let $\bar{S} = S_{l_1} \times \cdots \times S_{l_n}$ denote the set of all the states of a multi-level inclusive cache, where S_{l_x} ($1 \leq x \leq n$) denotes the set of all the states of the cache level l_x. Let R denote the set of all the references the program can generate. We define a function $f : R \times \bar{S} \to 2^L \times \bar{S}$ as:

$$f(r,\bar{s}) = \langle \{l_1,\cdots,l_z\}, \bar{s}' \rangle$$

where $\bar{s}'$ is semantically updated from $\bar{s}$ due to the reference r and l_z is the last cache level being updated by step (2) (i.e. $1 \leq z \leq n$) during the process. We can also lift the f function to deal with a sequence of references $\pi = (r_1,\cdots,r_h)$, i.e. we sequentially apply f to each reference in π with its prior state, and the result consists of the levels updated by r_h and the state at r_h. The collecting semantics $cs : R \to 2^L \times 2^{\bar{S}}$ is defined as:

$$cs(r) = \bigcup_{\pi \in \Pi_r} \bigcup_{\bar{s} \in \bar{S}_0} f(\pi, \bar{s})$$

where $\bar{S}_0$ is the set of all the initial states, and Π_r is the set of all the possible program reference sequences that reaches r. Note that here we use a loose notation to avoid cluttering: We treat the second component of $f(\pi,\bar{s})$ (i.e. a state) as a singleton, and we also treat $\bigcup$ can realize the set union of the first and second components respectively.

6.4.2 Abstract Semantics-Based Approach

Based on the concrete semantics, we propose an approach which attempts to analyze the cache levels together. Given a multi-level inclusive cache $\langle L, inc \rangle$, we define its abstract

cache hierarchy state domain Ω as:

$$\Omega = \Phi_{l_1} \times \cdots \times \Phi_{l_n}$$

where Φ_{l_x} $(1 \leq x \leq n)$ is the abstract state domain for the cache level l_x. Before giving the definition of Φ_{l_x} and the semantic functions on Φ_{l_x}, we want to present the approach first in order to give a general idea. To this end, we assume the abstract state domain Φ_{l_x} and the operations on Φ_{l_x} meet the conditions:

- From an abstract state $\phi_{l_x} \in \Phi_{l_x}$, we can safely derive a set of memory blocks that are definitely in the cache.
- From an abstract state $\phi_{l_x} \in \Phi_{l_x}$, we can safely derive a set of memory blocks that are possibly in the cache.
- From an abstract state $\phi_{l_x} \in \Phi_{l_x}$, we can safely derive a set of memory blocks that may be out of the cache after being loaded into the cache.
- An *update* function $\mathscr{U} : \Phi_{l_x} \times M_{l_x} \rightarrow \Phi_{l_x}$ can safely update the abstract states in the presence of possible "holes" caused by invalidations.
- A *join* function $\mathscr{J} : \Phi_{l_x} \times \Phi_{l_x} \rightarrow \Phi_{l_x}$ can safely join the abstract states in the presence of possible "holes" caused by invalidations.
- An *invalidate* function $\mathscr{I} : \Phi_{l_x} \times 2^{M_{l_x}} \rightarrow \Phi_{l_x}$ can safely perform invalidations on the abstract states.

Note that the first three conditions mean that we can safely derive the CHMC for a reference at a cache level l_x from its ϕ_{l_x}. Let us use an auxiliary function $chmc : \Phi_{l_x} \times M_{l_x} \rightarrow \{AH, AM, PS, NC\}$ to do this. In addition, we have another auxiliary functions $pout : \Phi_{l_x} \rightarrow 2^{M_{l_x}}$ to acquire an over-approximated set of memory blocks that may be out of the cache after being loaded into the cache.

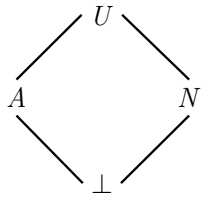
We also define a function domain Ψ that captures all the mappings from references to CACs at all the cache levels:

$$\Psi = R \rightarrow \Big(L \rightarrow \{U, A, N\}_{\perp} \Big)$$

In order to ensure the analysis *update* function monotone, we establish a lattice on the lifted set of CACs $\{U, A, N\}_{\perp} = \{U, A, N\} \cup \{\perp\}$ where $\perp$ means there has not been any CAC yet, and we also define a least upper bound operator $\sqcup$, as shown in Fig. 6.3. This ensures that if the CAC for a reference at a cache level becomes U, it will never be changed. A semantic function $\mathscr{F} : \Psi \times R \times L \times \{U, A, N\} \rightarrow \Psi$ is defined as:

$$\mathscr{F}(\psi, r, l, \text{CAC}) = \psi \Big[r \mapsto \psi(r)[l \mapsto \text{CAC} \sqcup \psi(r)(l)] \Big]$$

which derives the least upper bound of all possible CACs for a reference. Instead of having an abstract element of Ψ at each program point, we have a global abstract object $\psi \in \Psi$ for the whole program which is initialized as $\forall r \in R, l \in L : \psi(r)(l) = \perp$.



$\sqcup$	A	N	U	$\perp$
A	A	U	U	A
N	U	N	U	N
U	U	U	U	U
$\perp$	A	N	U	$\perp$

Figure 6.3: CAC lattice and least upper bound operator

At a join point of CFG, we use a *join* function $\mathscr{J} : \Omega \times \Omega \rightarrow \Omega$ to combine the abstract cache hierarchy states. Given two abstract states $\omega_1 = \langle \phi_{1,1}, \cdots, \phi_{n,1} \rangle \in \Omega$ and $\omega_2 = \langle \phi_{1,2}, \cdots, \phi_{n,2} \rangle \in \Omega$, the *join* function $\mathscr{J} : \Omega \times \Omega \rightarrow \Omega$ is defined as:

$$\mathscr{J}(\omega_1, \omega_2) = \langle \mathscr{J}(\phi_{1,1}, \phi_{1,2}), \cdots, \mathscr{J}(\phi_{n,1}, \phi_{n,2}) \rangle$$

namely it independently combines each level's abstract states.

An *update* function $\mathscr{U} : \Omega \times \Psi \times R \rightarrow \Omega \times \Psi$ is used to take into account the effect of a reference r on the abstract hierarchy state and the ψ abstract object. The *update* function is defined based on the concrete operational semantics regarding how a multi-level inclusive cache changes its states when a reference occurs, as described in Section 6.4.1. The steps to perform this *update* function are given in Algorithm 6.1.

Algorithm 6.1: Definition of the *update* function $\mathscr{U}$

Input: $\omega = \langle \phi_1, \cdots, \phi_n \rangle$, ψ, r
Output: $\omega' = \langle \phi'_1, \cdots, \phi'_n \rangle$, ψ

1 $x \leftarrow 1$;
2 CAC $\leftarrow A$;
3 **repeat**
4 $\quad \psi \leftarrow \mathscr{F}(\psi, r, l_x, \text{CAC})$;
5 $\quad$ CHMC $\leftarrow chmc(\phi_{l_x}, m^r_{l_x})$;
6 $\quad$ **if** CAC = A **then**
7 $\quad\quad$ **if** CHMC = AH **then** CAC $\leftarrow N$;
8 $\quad\quad$ **else if** CHMC = AM **then** CAC $\leftarrow A$;
9 $\quad\quad$ **else** CAC $\leftarrow U$;
10 $\quad$ **else if** CAC = U **then**
11 $\quad\quad$ **if** CHMC = AH **then** CAC $\leftarrow N$;
12 $\quad\quad$ **else** CAC $\leftarrow U$;
13 $\quad$ **else** CAC is unchanged (i.e. N) ;
14 $\quad x \leftarrow x + 1$;
15 **until** $x > n$;
16 $x \leftarrow n$;
17 $\omega' = \omega$;
18 **repeat**
19 $\quad$ CAC $\leftarrow \psi(r)(l_x)$;
20 $\quad$ **if** CAC = A **then** $\phi'_{l_x} \leftarrow \mathscr{U}(\phi'_{l_x}, m^r_{l_x})$;
21 $\quad$ **else if** CAC = N **then** $\phi'_{l_x} \leftarrow \phi'_{l_x}$;
22 $\quad$ **else** $\phi'_{l_x} \leftarrow \mathscr{J}(\mathscr{U}(\phi'_{l_x}, m^r_{l_x}), \phi'_{l_x})$
23 $\quad$ **if** $inc(l_x)$ **then**
24 $\quad\quad PO \leftarrow pout(\phi'_{l_x})$;
25 $\quad\quad$ **foreach** $1 \leq y < x$ **do**
26 $\quad\quad\quad PO' \leftarrow \emptyset$;
27 $\quad\quad\quad$ **foreach** $m_{l_x} \in PO$ **do**
28 $\quad\quad\quad\quad PO' \leftarrow PO' \cup \{m_{l_y} \in M_{l_y} | m_{l_y} \subseteq m_{l_x}\}$;
29 $\quad\quad\quad \phi'_{l_y} \leftarrow \mathscr{I}(\phi'_{l_y}, PO')$;
30 $\quad x \leftarrow x - 1$;
31 **until** $x = 0$;

The first loop (lines 3 – 15) abstracts the first step of the concrete operational semantics. It starts from the cache level l_1 and moves downwards through L in sequence to check if

a cache level l_x will be updated. Line 4 safely approximates the first component of the collecting semantics *cs* for a reference r (given $cs(r) = \langle L_r, \bar{S}_r \rangle$ where L_r is the set of all the cache levels that can be updated due to r, if we have $l_x \in L_r$, then safety means we should have $\psi(r)(l_x) \neq N$ when the analysis reaches its fixed-point). Lines 5 – 13 performs the same CAC derivation as described in [61]. Different from the traditional multi-level cache analysis method, we do not use the resultant CAC to update this cache level's state directly but we use it to derive the least upper bound for the reference's CACs at this level.

The second loop (lines 18 – 31) begins from the last cache level l_n and moves upwards through L in sequence to carry out updating. It first acquires the least upper bound of the CACs at a cache level l_x (line 19) and updates the level's abstract state ϕ_{l_x} according to the acquired CAC[5]. The abstract cache level state updating (lines 20 – 22) uses the traditional method described in [61]. After updating the state, it checks whether l_x is an inclusive level, and tries to invalidate the memory blocks in the abstract cache states of the levels located above l_x if it is inclusive (lines 23 – 29). Line 24 ensures that the set of memory blocks that cause invalidations at upper levels is over-approximated, and lines 26 – 28 extracts all the sub-blocks *w.r.t.* the cache block size at an upper level. This loop abstracts the behavior of step (3) and partial behavior of step (6) of the concrete operational semantics. The rest of the behavior of the steps is abstracted in the $\mathscr{U}$ and $\mathscr{I}$ functions.

6.4.3 Abstract Domain for A Cache Level

According to the conditions introduced in Section 6.4.2 for the abstract cache level state domain, it is actually to perform the three analyses (i.e. the *may*, *must*, and *persistence*) on each level at the same time. Therefore, we define the abstract cache state domain Φ_{l_x} for a cache level l_x as:

$$\Phi_{l_x} = \Theta_{l_x}^{may} \times \Theta_{l_x}^{must} \times \Theta_{l_x}^{pers} \times \Delta_{l_x}$$

[5]If the acquired CAC is *A* or *N*, it means the CAC for this reference at this level is always the same (*A* or *N*) during the analysis iterations.

where $\Theta_{l_x}^{may}$ is the set of all the ACSs in terms of *may* analysis of the cache level l_x in isolation; likewise, $\Theta_{l_x}^{must}$ and $\Theta_{l_x}^{pers}$ are that in terms of *must* and *persistence* analysis, respectively. Note that the $\Theta_{l_x}^{may}$ and $\Theta_{l_x}^{must}$ we use are consistent with the ones defined in [53], namely:

$$\Theta_{l_x}^{may} = \Theta_{l_x}^{must} = \{1, \cdots, d_{l_x}\} \rightarrow \left(\{1, \cdots, k_{l_x}\} \rightarrow 2^{M_{l_x}}\right)$$

However, there are different approaches to safe cache *persistence* analysis, in which different abstract domains are used [76, 73]. No matter which approach is used, for a cache set i, there is an additional logical position $\top_{l_x} \equiv k_{l_x} + 1$. Given a $\theta_{l_x}^{pers} \in \Theta_{l_x}^{pers}$, $\theta_{l_x}^{pers}(i)(\top_{l_x})$ is an over-approximated set of memory blocks that are possibly evicted after being loaded into this cache set. The last component domain Δ_{l_x} is defined as:

$$\Delta_{l_x} = \{1, \cdots, d_{l_x}\} \rightarrow \{1, \cdots, k_{l_x}, \top_{l_x}\}$$

An element in the domain Δ_{l_x} is used to capture the minimum logical position in which there may be an invalidated memory block for each cache set.

For simplicity, let us use Φ directly in the following without its subscript l_x. Three semantic functions need to be defined as described in Section 6.4.2.

Given two abstract states $\phi_1 = \langle \theta_1^{may}, \theta_1^{must}, \theta_1^{pers}, \delta_1 \rangle \in \Phi$ and $\phi_2 = \langle \theta_2^{may}, \theta_2^{must}, \theta_2^{pers}, \delta_2 \rangle \in \Phi$, the *join* function $\mathscr{J} : \Phi \times \Phi \rightarrow \Phi$ is defined as:

$$\mathscr{J}(\phi_1, \phi_2) = \langle \mathscr{J}^{may}(\theta_1^{may}, \theta_2^{may}), \mathscr{J}^{must}(\theta_1^{must}, \theta_2^{must}), \mathscr{J}^{pers}(\theta_1^{pers}, \theta_2^{pers}), \delta_\perp \rangle$$

where $\mathscr{J}^{may}$ is the traditional *join* function used for single-level cache *may* analysis (so are $\mathscr{J}^{must}$ and $\mathscr{J}^{pers}$), and $\delta_\perp \in \Delta$ is defined as $\forall i \in \{1, \cdots, d_{l_x}\} : \delta_\perp(i) = \top_{l_x}$.

Given a set *PO* of possibly invalidated memory blocks, the *invalidate* function $\mathscr{I} : \Phi \times 2^M \rightarrow \Phi$ is defined in Algorithm 6.2. It (lines 7 – 10) removes any possibly invalidated

memory block from the *must* and *persistence* components and puts the removed block into the corresponding $\top$ position. It (lines 4 – 6) also captures the minimum logical position in which there may be an invalidated memory block for each cache set. Note that we do not remove any block from the *may* component since a block in *PO* may not cause invalidation.

Algorithm 6.2: Definition of the *invalidate* function $\mathscr{I}$

Input: $\phi = \langle \theta^{may}, \theta^{must}, \theta^{pers}, \delta \rangle$, PO
Output: $\phi' = \langle \theta^{may'}, \theta^{must'}, \theta^{pers'}, \delta' \rangle$

```
1  φ' ← φ ;
2  δ' ← δ_⊥ ;
3  foreach m ∈ PO do
4      if m ∈ θ^may'(i)(j) then
5          if j < δ'(i) then
6              δ'(i) = j ;
7      if m ∈ θ^must'(i)(j) then
8          remove m from θ^must'(i)(j) ;
9      if m ∈ θ^pers'(i)(j) then
10         remove m from θ^pers'(i)(j) and put it into θ^pers'(i)(⊤) ;
```

According to the concrete operational semantics, we know that if a memory block in the logical position p of a cache set is invalidated, loading a new block into this cache set will not increment the logical positions of the blocks which are greater than p. When updating an abstract state, if we do not consider this behavior, it will not affect the safety but only the precision of the *must* and *persistence* analyses (both analyses stay safe as long as the logical positions of memory blocks are not underestimated). However, without considering this behavior we can have an unsafe *may* analysis, since it is required not to overestimate the logical positions of memory blocks in the *may* analysis. Therefore, we extend the traditional *update* function $\mathscr{U}^{may}$ to $\overline{\mathscr{U}}^{may} : \Theta^{may} \times M \times \Delta \to \Theta^{may}$ for the *may* analysis to take into account any possible invalidation behavior:

$$\overline{\mathscr{U}}^{may}(\theta^{may}, m, \delta) = \begin{cases} \mathscr{U}^{may}(\theta^{may}, m) & \text{if } p \geq j \\ \theta^{may}[set(m) \mapsto \varepsilon] & \text{otherwise} \end{cases}$$

where

$$p = \delta(set(m)) \bigwedge j = \begin{cases} h & \text{if } m \in \theta^{may}(set(m))(h) \\ \top & \text{otherwise} \end{cases} \bigwedge$$

$$\varepsilon = \begin{array}{l} \big[\iota_1 \mapsto \{m\}, \\ \iota_i \mapsto \theta^{may}(set(m))(\iota_{i-1}) | i = 2 \cdots p, \\ \iota_{p+1} \mapsto \theta^{may}(set(m))(\iota_p) \cup (\theta^{may}(set(m))(\iota_{p+1}) \backslash \{m\}), \\ \iota_i \mapsto \theta^{may}(set(m))(\iota_i) \backslash \{m\} | i = p+2 \cdots k\big] \end{array}$$

If there is no invalidation possible in this cache set (i.e. $p = \top$), or if m is in θ^{may} and its logical position is not behind the minimum position p where an invalidated memory block may reside, using the traditional $\mathscr{U}^{may}$ will not overestimate the logical positions of memory blocks. Otherwise, we need to consider there may be a "hole" behind p (including p) that needs to be filled first; so we can only increment the logical positions of the memory blocks until p, and keep the positions of other blocks unchanged (excluding m which will be moved to the first logical position if it is in the current θ^{may}). Based on this, given an abstract state $\phi = \langle \theta^{may}, \theta^{must}, \theta^{pers}, \delta \rangle \in \Phi$, we define the *update* function $\mathscr{U} : \Phi \times M \to \Phi$ as:

$$\mathscr{U}(\phi, m) = \langle \overline{\mathscr{U}}^{may}(\theta^{may}, m, \delta), \mathscr{U}^{must}(\theta^{must}, m), \mathscr{U}^{pers}(\theta^{pers}, m), \delta_{\perp} \rangle$$

where $\mathscr{U}^{must}$ and $\mathscr{U}^{pers}$ are the traditional *update* functions for *must* and *persistence* analysis, respectively.

In Chapter 5, an abstract domain called aging barrier is proposed to improve precision. Therefore, Φ can be extended by including that domain as a component domain (and also extend the *update* functions for *must* and *persistence* analyses). However, there are two reasons why we do not include this domain: (1) it is not necessary for safety of the analysis; (2) it may not be useful in the absence of data references.

6.4.4 Concretization

The set of concrete cache hierarchy states represented by an abstract cache hierarchy state $\omega = \langle \phi_{l_1}, \cdots, \phi_{l_n} \rangle$ where $\phi_{l_x} = \langle \theta_{l_x}^{may}, \theta_{l_x}^{must}, \theta_{l_x}^{pers}, \delta_{l_x} \rangle$ $(1 \leq x \leq n)$ can be derived by a concretization function $con_\Omega : \Omega \rightarrow 2^{\bar{S}}$. This concretization function is defined as:

$$con_\Omega(\omega) = \cdots \times \left(\gamma_{l_x}^{may}(\theta_{l_x}^{may}) \cap \gamma_{l_x}^{must}(\theta_{l_x}^{must}) \cap \gamma_{l_x}^{pers}(\theta_{l_x}^{pers}) \right) \times \cdots$$

where γ^{may}, γ^{must}, γ^{pers} are the well-defined concretization functions in [53] for the *may*, *must*, and *persistence* abstract states respectively. For an abstract cache level state, the set of concrete cache level states is derived independently of other levels. A concrete cache level state should satisfy the *may*, *must*, and *persistence* meanings of the abstract cache level state, so set intersection is used to guarantee this.

The meaning of the global abstract object $\psi \in \Psi$ is given by another concretization function $con_\Psi : \Psi \rightarrow \left(R \rightarrow 2^L \right)$ that is defined as:

$$con_\Psi(\psi) = \left[r \in R \mapsto \{ l \in L | \psi(r)(l) \neq N \} \right]$$

For each reference r, this concretization function determines a set of cache levels that may be accessed by r. Given a cache level l, if we have $\psi(r)(l) \neq N$ (i.e. A or U), r may access l and l is in the derived set of cache levels for r.

6.4.5 Discussion on Data References

Since most of the inclusive caches are unified caches, data references need to be taken into account eventually. As always, one difficulty related to data references is to precisely derive a set of possibly referenced memory addresses for each dynamic load/store instruction [77, 73]. In addition, we need to consider which write policy is used in the cache hierarchy. For example, if write-back policy is used, a dirty block needs to be written into

a lower level when it is evicted, which changes the cache state at that level. Thus, we need to safely classify whether a memory block is dirty (like *always*, *never*, or *uncertainly* dirty) and take into account the effect of eviction of dirty blocks on lower cache level states.

6.5 Theoretical Analysis of Safety and Termination

Theorem 6.5.1. *The proposed approach to multi-level inclusive cache analysis is safe.*

Proof. Given a reference r, the collecting semantics cs where $cs(r) = \langle L_r, \bar{S}_r \rangle$, the derived global abstract object ψ, and the derived abstract cache hierarchy state ω at r, if the analysis is safe, it should satisfy the following condition:

$$L_r \subseteq con_{\Psi}(\psi)(r) \bigwedge \bar{S}_r \subseteq con_{\Omega}(\omega)$$

We prove this by mathematical induction.

Base case: At the beginning of an execution, no memory block is loaded and the first reference needs to access all the cache levels. The abstract cache hierarchy state at the entry point derived by the analysis is empty. By the definition of the function $\mathscr{U}$ the first reference is certainly classified as *AM* at each cache level, so the ψ maps the first reference to *A* at each cache level. The condition holds.

Inductive hypothesis: Let T be the set of all the predecessor references of r. In the last iteration of the analysis, we have $\forall t \in T : L_t \subseteq con_{\Psi}(\psi)(t) \bigwedge \bar{S}_t \subseteq con_{\Omega}(\omega')$ where $cs(t) = \langle L_t, \bar{S}_t \rangle$. For simplicity, let us assume r only has one predecessor t, so we have $\omega = \mathscr{U}(\omega', \psi, t)$. If r has more than one predecessor, we can use the $\mathscr{J}$ function, which is safe by construction, to combine the abstract cache hierarchy states. Therefore, we need to prove ω' updated by t is safe and ψ updated by r is also safe.

Inductive step: When updating ω', since the CAC for t at each level is safe (given by the hypothesis $L_t \subseteq con_{\Psi}(\psi)(t)$), updating the components of the last level in ω' will be safe. Let us assume the last level is inclusive: Since its components are safely updated, the

set of possibly evicted blocks will be overestimated. According to the definition of the $\mathscr{I}$ function which remove blocks in the *must* and *persistence* state components, over-removal in these two state components made by the $\mathscr{I}$ function will not make the analysis unsafe. As described in Section 6.4.3, the *update* function $\mathscr{U}$ can safely update an abstract cache level state. By using mathematical induction on levels, we can prove the component for each level in ω' can be safely updated. Therefore, the ω' updated by t is also safe, i.e. the abstract cache hierarchy state ω is safe. Thus, we have $\bar{S}_r \subseteq con_\Omega(\omega)$. When updating $\psi(r)$, r's CHMC at each level is derived from the ω which is safe as proven above. Since we have $\bar{S}_r \subseteq con_\Omega(\omega)$, the CHMC for r at each level is safe, so is the deduced CAC. The $\mathscr{F}$ function uses the least upper bound operator to derive the reference's possible CAC. Thus, the ψ updated by r is safe, i.e. $L_r \subseteq con_\Psi(\psi)(r)$. □

In order to prove the proposed approach terminate, we need a well-defined partial ordering on each abstract domain. Given two abstract cache hierarchy states $\omega_1 = \langle \phi_{1,l_1}, \cdots, \phi_{1,l_n} \rangle$ and $\omega_2 = \langle \phi_{2,l_1}, \cdots, \phi_{2,l_n} \rangle$, the partial ordering $\preceq_\Omega$ on the the abstract cache hierarchy state domain Ω is defined as:

$$\omega_1 \preceq_\Omega \omega_2 \iff \phi_{1,l_1} \preceq_\Phi \phi_{2,l_1} \bigwedge \cdots \bigwedge \phi_{1,l_n} \preceq_\Phi \phi_{2,l_n}$$

where $\preceq_\Phi$ on the abstract cache level state domain Φ is defined naturally as the conjunction of the orders on the corresponding components in the Θ^{may}, Θ^{must}, Θ^{pers}, and Δ respectively. The partial orderings on the Θ^{may}, Θ^{must} and Θ^{pers} abstract cache state domains are already well-defined, so we define the partial ordering $\preceq_\Delta$ on the domain Δ. Given two elements δ_1 and δ_2 of the domain Δ, the partial ordering $\preceq_\Delta$ on the domain Δ is defined as:

$$\delta_1 \preceq_\Delta \delta_2 \iff \forall i \in \{1, \cdots, d\} : \delta_1(i) \geq \delta_2(i)$$

We also need to define the partial ordering on the domain Ψ. Given two elements ψ_1

and ψ_2 of the domain Ψ, the partial ordering $\preceq_\Psi$ on the domain Ψ is defined as:

$$\psi_1 \preceq_\Psi \psi_2 \iff \forall r \in R, \forall l \in L : \psi_1(r)(l) \preceq_{CAC} \psi_2(r)(l)$$

where $\preceq_{CAC}$ on the CAC domain is defined by the lattice as shown in Fig. 6.3.

Theorem 6.5.2. *The proposed approach to multi-level inclusive cache analysis terminates in finite iterations.*

Proof. The abstract cache hierarchy state domain Ω and the Ψ domain are finite and partially ordered, so if both $\mathscr{J}$ and $\mathscr{U}$ functions are monotone, the proposed analysis is guaranteed to terminate in finite iterations.

The $\mathscr{J}$ function only applies the monotone *join* functions of the *may*, *must*, and *persistence* analyses independently to the corresponding components of two abstract cache hierarchy states, so it is monotone by construction.

The $\mathscr{U}$ function is composed of four functions $\mathscr{F}$, $\mathscr{I}$, $\mathscr{U}$, and $\mathscr{J}$: (1) For any reference, the $\mathscr{F}$ function uses the $\sqcup$ operator on the reference's all possible CACs to derive a least upper bound. Thus, given $\psi_1 \preceq_\Psi \psi_2$, for any reference r at a cache level l with its CAC c, we have $\mathscr{F}(\psi_1, r, l, c) \preceq_\Psi \mathscr{F}(\psi_2, r, l, c)$. (2) The $\mathscr{I}$ function removes blocks only from θ^{must} and θ^{pers} of an abstract cache level state. Thus, given $\phi_1 \preceq_\Phi \phi_2$ and $PO_1 \subseteq PO_2$, we can easily verify the resultant θ_1^{must} from $\mathscr{I}(\phi_1, PO_1)$ and the resultant θ_2^{must} from $\mathscr{I}(\phi_2, PO_2)$ have the relation: $\theta_1^{must} \preceq^{must} \theta_2^{must}$; also the resultant θ_1^{pers} and θ_2^{pers} have the relation: $\theta_1^{pers} \preceq^{pers} \theta_2^{pers}$. Since the θ_1^{may} and θ_2^{may} are not changed by the $\mathscr{I}$ function (i.e. $\theta_1^{may} \preceq^{may} \theta_2^{may}$ still holds) and $PO_1 \subseteq PO_2$, we can also easily verify the resultant δ_1 and δ_2 have the relation: $\delta_1 \preceq_\Delta \delta_2$. (3) The $\mathscr{U}$ function is composed of the $\overline{\mathscr{U}}^{may}$ function and the well-defined monotone *update* functions $\mathscr{U}^{must}$ and $\mathscr{U}^{pers}$. Thus, the $\mathscr{U}$ function is monotone as long as the $\overline{\mathscr{U}}^{may}$ function is monotone. Given a memory block m, $\theta_1^{may} \preceq^{may} \theta_2^{may}$, and $\delta_1 \preceq_\Delta \delta_2$, we can verify $\overline{\mathscr{U}}^{may}(\theta_1^{may}, m, \delta_1) \preceq^{may} \overline{\mathscr{U}}^{may}(\theta_2^{may}, m, \delta_2)$, since $\delta_1 \preceq_\Delta \delta_2$ means we have $\delta_1(set(m)) \geq \delta_2(set(m))$ such that age increasing in θ_2^{may} is

more conservative by the definition of the $\overline{\mathscr{U}}^{may}$ function. (4) The $\mathscr{J}$ function is monotone as shown above. Since the functions $\mathscr{F}$, $\mathscr{I}$, $\mathscr{U}$, and $\mathscr{J}$ are all monotone, the $\mathscr{U}$ function is monotone. □

6.6 Evaluation

In this section, we evaluate the approach proposed in this chapter and also compare with the state-of-the-art approaches that can deal with inclusion enforcement at some levels and are proposed in [64] and Chapter 5. We use the research prototype tool described in Appendix A to perform the experiments. The evaluations are carried out on a set of benchmarks (as shown in Tab. 6.2) maintained by the Mälardalen WCET research group [114], and they are compiled for MIPS R3000 processor using gcc-3.4.4. First, we describe the evaluation setup and also some assumptions in the experiments which nevertheless do not affect the validity of the evaluation. Then, we present the evaluation results showing that the proposed approach improves the precision, and we also show the computational overhead associated with the evaluation.

Due to the limitations of our current tool, we only take into account the timing effects of multi-level inclusive instruction caches and do not consider data references for now as argued in Section 6.2. We also do not consider the effects of other micro-architectural features like pipelines and branch predictors, so we assume there are no timing anomalies. Therefore, a memory reference that is classified as *NC* can be safely treated as a *AM* when used to estimate the WCET.

We carry out the experiments on a two-level cache hierarchy and configure L2 to be inclusive. The fixed parameters of the cache hierarchy are shown in Tab. 6.1. Moreover, we assume every needed information can be found in the main memory with a 100-cycle latency.

Three experiments are performed on each benchmark under different cache capacity configurations. Let $size(L1)$ denote the capacity of L1 cache, and let $size(L2)$ denote the

Table 6.1: Fixed Parameters of Two-Level Cache Hierarchy

Level	Block Size	Associativity	Latency
L1	8B	2-way	1-cycle
L2	16B	4-way	10-cycle

capacity of L2 cache. Let us assume that L2 cache size is always four times bigger than that of L1 cache, namely we have $size(L1) = \frac{size(L2)}{4}$. For a benchmark bm, let $size(bm)$ denote the code size of this benchmark. The cache size configurations for each benchmark are shown in Tab. 6.2, whose criteria are described as follows:

Large: L2 cache size is not smaller than the code size, and it also satisfies $size(L2) \geq 2 \times size(bm) > \frac{size(L2)}{2}$.

Medium: L2 cache size is not bigger than the code size, and it also satisfies $size(L2) \leq size(bm) < 2 \times size(L2)$.

Small: L2 cache size is not bigger than half the code size, and it also satisfies $size(L2) \leq \frac{size(bm)}{2} < 2 \times size(L2)$.

Tab. 6.3 shows the experimental results. For a benchmark, WCET_1 is derived by using the approach proposed in [64], WCET_2 is derived by using the approach proposed in Chapter 5, and WCET_3 is derived by using the approach proposed in this chapter. The WCET estimation is reported in clock cycles. The precision improvement compared to the approach proposed in [64] is calculated by $\frac{\text{WCET}_1}{\text{WCET}_3} - 1$. Likewise, the precision improvement compared to the approach proposed in Chapter 5 is calculated by $\frac{\text{WCET}_2}{\text{WCET}_3} - 1$. We also report the computation time overhead in seconds, along with the reported WCET. The experiments are performed on a Linux machine with a $1.2GHz$ quad-core processor and $12GB$ memory.

As we can observe from the results, the approach proposed in this chapter dominates the other two approaches in most cases. Under the large cache size configuration, the approach proposed in this chapter performs almost the same as the approach proposed in

Table 6.2: Large, Medium, and Small Cache Size Configurations for Each Benchmark

Benchmark	Code Size	Large		Medium		Small	
		L1	L2	L1	L2	L1	L2
bs	320B	256B	1KB	64B	256B	32B	128B
insertsort	440B	256B	1KB	64B	256B	32B	128B
janne	324B	256B	1KB	64B	256B	32B	128B
cnt	944B	512B	2KB	128B	512B	64B	256B
expint	888B	512B	2KB	128B	512B	64B	256B
fir	600B	512B	2KB	128B	512B	64B	256B
ns	588B	512B	2KB	128B	512B	64B	256B
prime	556B	512B	2KB	128B	512B	64B	256B
qurt	1328B	1KB	4KB	256B	1KB	128B	512B
select	1580B	1KB	4KB	256B	1KB	128B	512B
compress	3564B	2KB	8KB	512B	2KB	256B	1KB
edn	3576B	2KB	8KB	512B	2KB	256B	1KB
jfdctint	2580B	2KB	8KB	512B	2KB	256B	1KB
lms	2588B	2KB	8KB	512B	2KB	256B	1KB
ludcmp	2276B	2KB	8KB	512B	2KB	256B	1KB
minver	3052B	2KB	8KB	512B	2KB	256B	1KB
ndes	3392B	2KB	8KB	512B	2KB	256B	1KB
adpcm	7612B	4KB	16KB	1KB	4KB	512B	2KB
statemate	10296B	8KB	32KB	2KB	8KB	1KB	4KB
nsichneu	40036B	32KB	128KB	8KB	32KB	4KB	16KB

Chapter 5: For only a few benchmarks, $WCET_3$ is lower than $WCET_2$ but the precision improvement is only within 3%. However, both the approaches perform better than the approach proposed in [64]. This is expected and reasonable, since the approach proposed in [64] does not try to classify any access as *A* or *N* instead of *U* at a lower level than L1.

A striking difference appears under the medium and small configurations. Under both configurations, the approach proposed in this chapter gives above 10% improvement in most cases. For some benchmarks, the precision improvement is very significant (over 100%). Due to this huge improvement, we also analyze some small benchmarks by hand to find out the reasons. For example, under the medium configuration, for *insertsort*, the approach proposed in this chapter can achieve more than 170% improvement compared to the other two approaches. The reason for this is explained as follows: *insertsort* has two nested loops, and the total size of the two nested loops is 228 bytes which is smaller than

Table 6.3: Experimental Results: WCET Estimates and Computation Time Overheads

Benchmark	Configuration	Approach in [64]		Approach in Chapter 5		Approach proposed		$\frac{WCET_1}{WCET_3}$-1	$\frac{WCET_2}{WCET_3}$-1
		$WCET_1$	Overhead	$WCET_2$	Overhead	$WCET_3$	Overhead		
bs	Large	4827	0	4027	0.1	4027	0.1	19.87%	0.00%
	Medium	7357	0	6857	0	3757	0.1	95.82%	82.51%
	Small	10079	0	9579	0	6679	0	50.90%	43.42%
insertsort	Large	17229	0.1	15929	0.2	15929	0.2	8.16%	0.00%
	Medium	187559	0	186359	0.1	68959	0.1	171.99%	170.24%
	Small	187559	0	186359	0.1	110859	0.1	69.19%	68.10%
janne	Large	5863	0	4963	0.1	4963	0.1	18.13%	0.00%
	Medium	20577	0	20077	0.1	15077	0.1	36.48%	33.16%
	Small	33767	0	33267	0	24167	0	39.72%	37.65%
cnt	Large	27176	0.4	25176	0.8	25176	0.7	7.94%	0.00%
	Medium	362188	0.2	358308	0.4	277508	0.6	30.51%	29.12%
	Small	537178	0.1	533298	0.3	289198	0.3	85.75%	84.41%
expint	Large	30737	0.2	29537	0.4	29337	0.3	4.77%	0.68%
	Medium	183462	0.1	171262	0.2	109162	0.3	68.06%	56.89%
	Small	474762	0.1	473162	0.2	322262	0.2	47.32%	46.83%
fir	Large	15643	0.2	14443	0.4	14443	0.5	8.31%	0.00%
	Medium	196536	0.1	170586	0.2	136786	0.3	43.68%	24.71%
	Small	380736	0.1	379636	0.1	213936	0.1	77.97%	77.45%
ns	Large	33601	0.3	31601	0.4	31601	0.4	6.33%	0.00%
	Medium	164491	0.1	162591	0.2	162291	0.2	1.36%	0.18%
	Small	1473161	0.1	1472305	0.2	972305	0.2	51.51%	51.42%
prime	Large	38044	0.3	36644	0.5	36644	0.4	3.82%	0.00%
	Medium	190664	0.1	189164	0.2	185064	0.3	3.03%	2.22%
	Small	1690694	0.1	1689194	0.2	612594	0.1	175.99%	175.74%
qurt	Large	43705	3.3	40905	6.2	39897	5.4	9.54%	2.53%
	Medium	175156	1.3	168176	2.5	150276	2.8	16.56%	11.91%
	Small	182356	1.0	179556	2.0	114056	1.5	59.88%	57.43%
select	Large	43264	3.0	42264	5.5	41064	6.6	5.36%	2.92%
	Medium	237114	1.0	236194	1.8	168594	3.2	40.64%	40.10%
	Small	237114	0.5	202594	1.1	153394	1.7	54.58%	32.07%
compress	Large	217433	13.0	213333	25.2	212933	26.7	2.11%	0.19%
	Medium	1624044	5.1	1614444	9.5	1472344	11.7	10.30%	9.65%
	Small	1624044	2.9	1614444	5.5	1299144	6.7	25.01%	24.27%
edn	Large	217583	19.1	211483	25.8	211483	35.6	2.88%	0.00%
	Medium	740873	5.7	734663	9.4	602663	9.8	22.93%	21.90%
	Small	2360263	2.8	2354263	5.3	2254963	5.3	4.67%	4.40%
jfdctint	Large	42825	11.9	42225	14.7	42125	20.9	1.66%	0.24%
	Medium	164895	3.2	164295	4.8	48195	4.1	242.14%	240.90%
	Small	265895	1.9	197895	3.2	107895	1.8	146.44%	83.41%
lms	Large	480987	8.9	479187	18.0	479187	16.2	0.38%	0.00%
	Medium	10438520	4.7	10414720	8.8	10410720	10.4	0.27%	0.04%
	Small	25358756	2.7	25047856	5.5	21404296	8.7	18.48%	17.02%
ludcmp	Large	40885	3.9	39285	7.2	39285	6.8	4.07%	0.00%
	Medium	61245	2.4	59645	4.3	59645	4.5	2.68%	0.00%
	Small	293163	1.4	292263	2.8	290063	3.8	1.07%	0.76%
minver	Large	56314	6.7	53914	11.9	53914	12.9	4.45%	0.00%
	Medium	73759	3.4	71159	6.1	70759	7.3	4.24%	0.57%
	Small	228073	2.1	224333	3.9	215233	5.7	5.97%	4.23%
ndes	Large	204676	12.1	202376	21.6	202076	25.4	1.29%	0.15%
	Medium	2967930	5.6	2926910	10.0	2926010	14.4	1.43%	0.03%
	Small	5615987	3.2	5497527	6.4	4845877	11.0	15.89%	13.45%
adpcm	Large	388500	105.1	385600	199.5	384300	233.0	1.09%	0.34%
	Medium	1262808	37.4	1251888	71.3	1127392	111.2	12.01%	11.04%
	Small	1640818	24.9	1600638	49.9	1524568	45.3	7.63%	4.99%
statemate	Large	102633	124.0	96733	230.9	96533	233.6	6.32%	0.21%
	Medium	159710	54.5	153150	93.9	133530	143.8	19.61%	14.69%
	Small	160220	33.1	153560	58.5	119560	112.0	34.01%	28.44%
nsichneu	Large	610498	3023.7	610198	6327.7	608598	8880.6	0.31%	0.26%
	Medium	1096788	897.3	1096688	1491.0	1088088	5675.8	0.80%	0.79%
	Small	1144988	528.6	1144888	887.3	1138288	3725.9	0.59%	0.58%

the L2 cache size under the medium configuration which is 256 bytes; thus, if a reference in the loops cannot be classified as L1 *AH*, it should at least be classified as L2 *PS* if not L2 *AH*. While the approach proposed in this chapter achieves this (i.e. it gives L2 *AH* or L2 *PS* to the references in the loops if they cannot be classified as L1 *AH*), the other two approaches fails to give such L2 classifications to the references in the loops (they assign L2 *NC* to them).

When applying the approach proposed in this chapter, an interesting phenomenon is that a *smaller* cache size configuration may result in a *tighter* estimate (namely *bs*, *qurt*, *select*, *compress*, and *statemate*). As observed from the results of *bs* benchmark, the calculated bound under the large cache size configuration (4027 clock cycles) is higher than that under the medium cache size configuration (3757 clock cycles). The reason for such a phenomenon is as follows: The code size in the loop of *bs* is 208 bytes which is smaller than 256 bytes, so most of the loop memory blocks are at least persistent if not always in both L1 and L2 caches under the large configuration; whereas, most of the blocks are persistent in L2 cache under the medium configuration, but their sub-blocks will be evicted from L1 cache along with the loop iterations (since L1 cache size under the medium configuration is only 64 bytes and the shortest path in the loop involves 128 bytes). Therefore, there are several references which are classified as L1 *PS* & L2 *PS* under the large configuration, and are classified as L1 *AM* & L2 *PS* under the medium configuration. Given such a reference under the large configuration, L1 *PS* & L2 *PS* means that the corresponding L2 block is not in the $\theta_{l_2}^{must}$ and its L2 *CAC* is *U*; and it cannot bring the corresponding L2 block into the $\theta_{l_2}^{must}$; thus, its subsequent references to the same L2 block can only be classified as L2 *PS* (if not L1 *AH*) but not L2 *AH*; namely, such subsequent ones can add additional main memory access latencies (100 clock cycles) to the total estimate[6]. On the contrary, such a reference under the medium configuration is classified as L1 *AM* & L2 *PS*, so its L2 *CAC* is *A* which enables it to bring the corresponding L2 block into the $\theta_{l_2}^{must}$; some of

[6]If a reference is not classified as L1 *AH* and is classified as L2 *PS*, it can contribute at most one main memory access latency even it is in a loop.

its subsequent references to the same L2 block can be classified as L2 *AH* instead of L2 *PS*. Thus, the number of L1 misses under the medium one is proportional to the number of the iterations and can be much more than that under the large one; however, the number of overestimated L2 misses under the medium one can be less than that under the large one. Given the loop bound is only 4 and the difference between the latency of L2 access and main memory access is big, this phenomenon occurs. This phenomenon also shows the approach proposed in this chapter can tightly analyze multi-level inclusive caches.

Compared to the other two approaches, the approach proposed in this chapter seems to require more computation time, especially in the case of *nsichneu* benchmark (the biggest one we use). This is because the approach proposed in this chapter needs to perform more iterations due to the CACs for many references at L2 changing from *A* or *N* to *U*.

6.7 Conclusion

In this chapter, we propose an approach that can safely and more precisely analyze multi-level inclusive caches for WCET estimation. We first define a concrete operational semantics for multi-level inclusive caches. Based on this concrete semantics, the proposed approach analyze a multi-level inclusive cache as a whole by integrating three analyses together. We evaluate the proposed approach using a set of benchmarks. From the experimental results, we can observe the proposed approach can significantly tighten the WCET bound under relatively small cache size configurations, compared to state-of-the-art approaches.

Chapter 7

Cache-Related Preemption Delay Analysis for Multi-Level Inclusive Caches

CPS software usually consists of many hard real-time tasks running on a single processor, e.g. brake control task and engine control task may run on the same ECU in a automotive vehicle. Many such systems employ preemptive scheduling strategies, under which a task may be frequently preempted by other higher priority tasks. When executing these higher priority tasks (*preempting tasks*), the states of many underlying micro-architectural components are often "polluted". Therefore, additional overhead on the preempted task's execution time is expected due to the state "pollution". Since caches have significant impact on the variation of execution time, most of the overhead is due to cache state "pollution", which is referred to as cache-related preemption delay (CRPD).

CRPD analysis has been studied with respect to single-level caches extensively. As embedded processors are increasingly equipped with cache hierarchies nowadays, CRPD analysis deserves a study in terms of multi-level caches. However, in the presence of multi-level caches, CRPD analysis becomes much more challenging since the amount of intra-task interference may be changed by preemption [104]. Therefore, it becomes unsafe to estimate CRPD without considering the interrelations between different levels.

In [104], CRPD analysis in the context of multi-level non-inclusive caches has been investigated. Since cache hierarchies of different types may have different behaviors even for the same memory reference sequence, CRPD analysis for multi-level non-inclusive caches may not be fit for multi-level inclusive caches. Therefore, in this chapter, we first study the distinctions between non-inclusive and inclusive cache hierarchies in order to see why and how the effect of preemption can be different; then we propose an approach which can safely analyze CRPD in the context of multi-level inclusive caches. To the best of our knowledge, this is the first time to analyze CRPD in terms of multi-level ***inclusive*** caches.

The main contributions of this chapter are: (1) We identify the challenges of analyzing CRPD in the context of multi-level inclusive caches, compared to the multi-level non-inclusive caches; (2) We present a tight upper bound on how many times the classification of a reference may be changed in a loop, and we also prove the bound is safe; (3) We propose a "watermark" method based on the *persistence* analysis to safely estimate the amount of inter-task interference made by a preempting task; (4) We propose a calculation method to safely estimate CRPD in the presence of multi-level inclusive caches.

The rest of this chapter is organized as: Section 7.1 states why CRPD analysis for multi-level inclusive caches is a hard problem; Section 7.2 describes the system model; Section 7.3 presents the proposed approach to CRPD analysis for multi-level inclusive caches; and Section 7.4 concludes this chapter.

7.1 Problem Statement

Cache analysis for WCET estimation usually only takes into account intra-task interference to derive an upper bound on cache misses. This upper bound on cache misses may not be safe if inter-task interference is also possible. Therefore, CRPD analysis needs to bound the number of additional cache misses caused by inter-task interference.

CRPD analysis for single-level caches (without loss of generality, let us focus on A-way set associative caches with LRU replacement policy) relies on a concept of "useful cache blocks" (UCBs)[1] [123]. At a program point, a UCB only captures the first references to a memory blocks after that program point. The number of UCBs at a program point serves as an upper bound on the additional cache misses when preemption happens at this program point. This upper bound is safe (although may not tight) since (1) a single-level

[1] They are called "useful cache blocks" due to a historical reason, although they represent the memory blocks in these cache blocks are "useful". The name is coined in [94] which studies direct-mapped caches: At a program point, a cache block is "useful" only if it *may* contain a memory block that the *first* reference to it *after* this program point along *some* path can be a cache hit. Since a memory block can only stay in one cache block in direct-mapped caches, the "usefulness" of a memory block can be inherited by the corresponding cache block.

cache is always accessed, so the amount of intra-task interference to a memory block will not changed by preemption; (2) after the first reference to a memory block beyond the preemption point, the memory block will become the youngest and its LRU age can only be affected by intra-task interference. Therefore, in terms of single-level caches, the first reference to a memory block beyond the preemption point can be treated as a "firewall" to prevent the inter-task interference from affecting the LRU age of the memory block afterwards. As a result, after the first reference to a memory block beyond the preemption point, any further reference to this memory block would have the same *cache hit/miss classification* (CHMC) as that in the absence of preemption.

However, as stated in [104], only considering the effect of inter-task interference on the first references to a memory block may not be safe in terms of multi-level caches. While L1 is always accessed, the other lower cache levels are usually not be accessed by every memory reference. Compared to the case where there is no preemption, some references may need to access the lower cache levels in the presence of preemption. Therefore, the amount of intra-task interference to a memory block at a lower level may be increased due to the preemption. Due to the possibility that the amount of intra-task interference to a memory block can change, we cannot treat the first reference to the memory block at a cache level as the "firewall" to stop the effect of preemption on the memory block's LRU age afterwards. This phenomenon is referred to as *the indirect effect of preemption* in [104].

CRPD analysis for multi-level non-inclusive caches has been studied in [104]. However, non-inclusive cache hierarchies have less strict inclusion properties from inclusive cache hierarchies[2], which induces different interactions between cache levels. Since multi-level inclusive caches enforce strict inclusion property, when a memory block is evicted from a lower inclusive level, it also needs to invalidate its contents at any upper level. This invalidation behavior can make CRPD analysis for multi-level inclusive caches challeng-

[2]Multi-level inclusive caches require that the contents at upper cache levels *must* be a subset of the contents at lower levels, while multi-level non-inclusive caches allow duplicated contents existing at cache levels but do not strictly enforce the inclusion property

ing. For instance, when analyzing CRPD in terms of multi-level non-inclusive caches, given a sequence of references to a memory block after a preemption point, except for the first reference whose L1 behavior needs further analysis, we can safely inherit the others' L1 behavior as what they were classified in the absence of preemption [104]. However, when analyzing CRPD in terms of multi-level inclusive caches, due to the possible invalidation behavior, it may become unsafe to do such an inheritance without any further analysis. An example is shown in Fig. 7.1, in which a two-level inclusive cache with L2

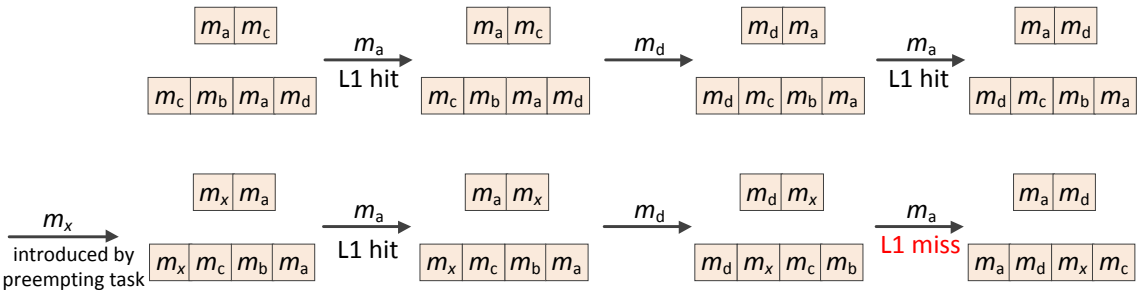


Figure 7.1: L1 behavior inheritance is affected by invalidation behavior

being inclusive is used. For simplicity, we assume the cache block sizes are the same and the memory blocks m_a, m_b, m_c, m_d, and m_x are mapped to the same sets at these two levels. The upper part of the figure shows the states of the cache hierarchy in the absence of preemption. From the figure, we can observe that the second reference to m_a is a L1 cache hit. However, when an preemption happens before the first reference to m_a and introduces another memory block m_x into the cache hierarchy, the second reference to m_a cannot be L1 cache hit anymore. This is because when m_d is referenced, it evicts m_a from L2 cache, which invalidates m_a in L1 cache.

Therefore, the problem is how to adapt the approach to CRPD analysis for multi-level inclusive caches in order to derive a safe and precise estimate.

7.2 System Model

In this chapter, we focus on a two-level inclusive cache model, which can be found in many commercial multi-core processors. In the model, L2 cache is inclusive which means the contents of L1 is always a subset of the contents of L2. At each cache level, we assume that the cache is set associative – let A_{L1} denote the associativity of L1 cache and let A_{L2} denote the associativity of L2 cache. We also assume LRU (Least Recently Used) replacement policy is used. The size of a cache block can be different at different cache levels, and it is assumed the cache block size does not increase as the level goes up. It is also assumed the capacity decreases as the level goes up.

As mentioned in previous chapters, we also assume the time to access a cache level is bounded and predictable, which can be achieved by using deterministic interconnects to connect the caches, like TDMA buses [122].

In this chapter, we focus only on how to analyze CRPD in the context of multi-level caches which can suffer invalidations caused by the inclusion enforcement, so we simply consider instruction references in terms of a single processor (i.e. no data references and inter-core interferences). This work can serve as a basis for CRPD analysis of multi-level inclusive data or unified caches in terms of a multi-core processor.

Given a task τ, we use the approach proposed in Chapter 6 to perform the multi-level inclusive cache analysis of this task. Therefore, for a reference, we can obtain its cache hit/miss classification (CHMC) at both cache levels. As usual, CHMC includes *always hit AH*, *always miss AM*, *persistent PS*, and *non-classified NC*. However, as shown later in Section 7.3, the inter-task interference due to the preemption may not only affect *AH* and *PS* references, but also *AM* references, namely *AM* classification may not hold in the presence of preemption. Therefore, we use the *bottom-up* approach proposed in Chapter 5 (only the *may* analysis of L2 inclusive cache) to categorize some references as *definitely always miss DAM*, since when classifying *AM* references, that approach does not rely on the contents of L1 cache.

Since CRPD analysis is after the WCET analysis, given a reference r, we assume r has the following attributes as shown in Tab. 7.1 and these attributes are all set during the WCET analysis. Each attribute is accessed by using "." operator.

Table 7.1: Attributes of A Reference

Attribute	Description
$r.m_{L1}$	the referenced memory block by r w.r.t. the L1 cache block size
$r.m_{L2}$	the referenced memory block by r w.r.t. the L2 cache block size
$r.c_{L1}$	cache hit/miss classification of r at L1 cache level
$r.c_{L2}$	cache hit/miss classification of r at L2 cache level
$r.a_{L1}^{must}$	if $r.c_{L1}$ is *AH*, $r.m_{L1}$'s position in the *must* abstract set state; else, $\top_{L1}$
$r.a_{L2}^{must}$	if $r.c_{L2}$ is *AH*, $r.m_{L2}$'s position in the *must* abstract set state; else, $\top_{L2}$
$r.a_{L1}^{pers}$	if $r.c_{L1}$ is *PS*, $r.m_{L1}$'s position in the *persistence* abstract set state; else, $\top_{L1}$
$r.a_{L2}^{pers}$	if $r.c_{L2}$ is *PS*, $r.m_{L2}$'s position in the *persistence* abstract set state; else, $\top_{L2}$
$r.a_{L1}^{may}$	if $r.c_{L1}$ is not *AM*, $r.m_{L1}$'s position in the *may* abstract set state; else, $\top_{L1}$
$r.a_{L2}^{may}$	if $r.c_{L2}$ is not *AM*, $r.m_{L2}$'s position in the *may* abstract set state; else, $\top_{L2}$

Note that $\top_{L1} = A_{L1} + 1$ and $\top_{L2} = A_{L2} + 1$.

Moreover, we assume the task τ can only be preempted by one task $\hat{\tau}$. In the case of multiple preemptions due to preempting tasks, we can directly extend this work by using the preempting task composition approach proposed in [124].

7.3 CRPD Analysis for Multi-Level Inclusive Caches

Since the CRPD estimate is often used together with the WCET estimate in schedulability analysis, we do not need to derive a sound standalone CRPD estimate. Instead, as long as the derived WCET + CRPD is safe, we can guarantee the schedulability analysis is sound [103]. Therefore, our approach to CRPD analysis is similar to the method proposed in [104], namely we only consider how to bound the number of additional L1/L2 cache misses due to the effect of preemption on the *always hit* (*AH*) and *persistent* (*PS*) references (i.e., they refer to the definitely and persistently cached memory blocks).

7.3.1 Preempted Task Analysis

Given the preempted task τ and a two-level inclusive cache hierarchy, we use the approach proposed in Chapter 6 to derive the WCET estimate and the references classified as L1/L2 *AH*/*PS* . At a program point, if a reference is classified as L1/L2 *AH*/*PS*, they contribute much less than other references to the WCET estimate. However, when preemption introduces interference of other tasks, the reference's L1/L2 *AH*/*PS* classification may not hold.

Note that due to the inclusion property, if a reference is classified as L1 *AH*, it implies this reference is also L2 *AH*. Since the concept of *persistence* only makes sense in loops, we also assume that if a reference is classified as *PS*, it implies this reference is in a loop.

As stated in Section 7.1, when analyzing CRPD in terms of a multi-level cache, a method only based on the traditional UCB concept may underestimate the additional cache misses. Therefore, at a program point, we need to use a new concept to capture what references beyond this program point may be influenced by the preemption-introduced interference such that they may contribute more to the execution time. As we argued above, we only focus on how to make WCET + CRPD sound, so the references of interest should be "positively" classified by the WCET analysis, and these references can be categorized into 7 types as shown in Tab. 7.2.

Table 7.2: 7 Types of Positively Classified References

	Type 1	Type 2	Type 3	Type 4	Type 5	Type 6	Type 7
L1	*AH*	*PS*	*AM*	*NC*	*PS*	*AM*	*NC*
L2	*AH*	*AH*	*AH*	*AH*	*PS*	*PS*	*PS*

Note: L1/L2 *PS* implies the reference is located in a loop.

Definition 7.3.1. *A positive reference is a reference that is classified as one of the 7 types shown in Tab. 7.2, i.e., its referenced memory block is definitely/persistently in L2 cache when the reference occurs.*

Definition 7.3.2. *Given a* positive *reference, if we cannot ensure its L2 "positiveness" in the presence of preemption, we call this positive reference* declined.

Therefore, a declined positive reference contributes *at least* a L2 cache miss penalty to the CRPD estimate. Due to the inclusion property, if a positive reference in type 1, type 2, or type 5 is declined, it can also contribute a L1 cache miss penalty to the CRPD estimate.

7.3.1.1 Bound on Times A Positive Reference Can Be Declined

Before presenting how to gather the positive references at a program point, we want to derive an upper bound on how many times such a reference may be "declined". If a positive reference is not in a loop, it is straightforward to see it can only be declined once since it can only be executed once. The problem emerges when a positive reference is in a loop, in which case the positive reference can be executed many times.

In [104], the number of times a positive reference in a loop can be declined has been studied in the context of multi-level non-inclusive caches[3]. In that case, the bound depends on the number of sets and associativities of L1 and L2 caches. However, we find that bound in the context of multi-level non-inclusive caches may not be suitable for multi-level inclusive caches. One important difference is: in the context of inclusive caches, the amount of intra-task interference to a memory block may be reduced due to the preemption at L1/L2; but in the context of non-inclusive caches, the amount of intra-task interference will not change at L1 and the amount cannot be reduced at L2.

For example, consider the situation shown in Fig. 7.2. In the example, we suppose L2 cache is fully associative and L1 cache is 2-way set associative with 4 cache sets. The cache block size at L1 is a quarter of the cache block size at L2. Since the cache block size at L1 is smaller than that at L2, given a memory block m in terms of L2 cache, we use m^i to denote the i^{th} sub-block of m in terms of L1 cache. Therefore, each sub-block of a

[3] Although not stated explicitly, we can find one **implication** in this study is that the cache block sizes are the same at both L1 and L2 caches. However, in this work, we do not impose this restriction.

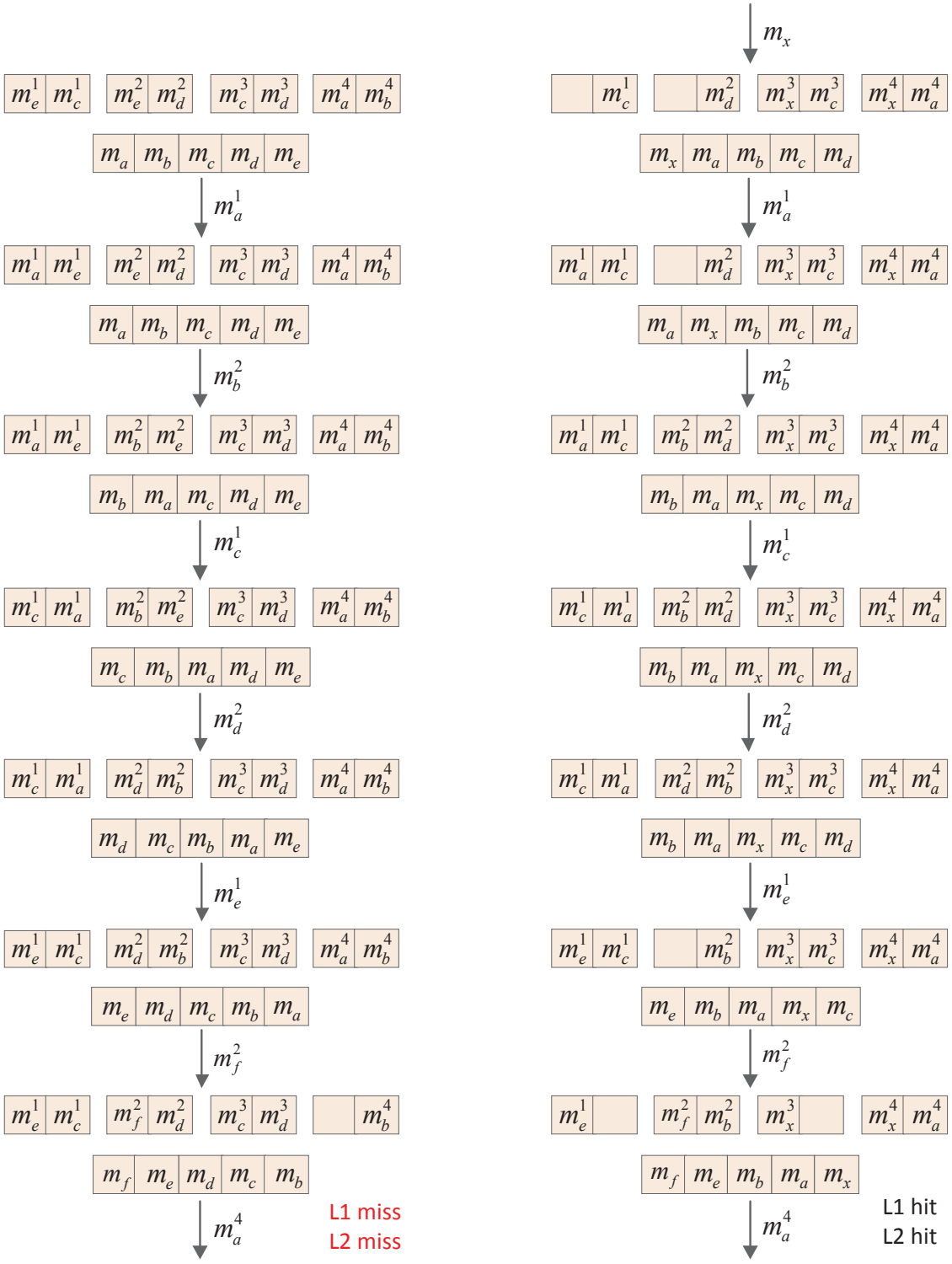


Figure 7.2: The amount of intra-task interference may be reduced due to preemption.

L2 memory block is mapped to the corresponding L1 cache set (e.g., m_a^1 is mapped to the first L1 cache set and so forth). Let us also assume the preemption introduces one memory block m_x into the cache hierarchy (m_x is called an evicting cache block in the literature). As we can observe, the introduced m_x causes less amount of intra-task interference to m_a in L2 cache, specifically the references to m_c and m_d do not access L2 cache since the needed sub-blocks can be found in L1 cache. The result of this phenomenon is: The reference to some information in m_a^4 can be classified as L1 *AM* and L2 *AM* in the absence of preemption, the reference may hit both caches in the presence of preemption.

Now we need to consider when a memory reference r at a program point p in a loop is classified positively by the cache analysis method in Chapter 6, whether r can maintain its "positiveness" in the presence of preemption. In order to facilitate the presentation, let us define what we mean by "a conflicting memory block of $r.m_{L2}$" and also "a memory block is demanded on a path".

Definition 7.3.3. *A conflicting memory block of $r.m_{L2}$ is a memory block other than $r.m_{L2}$ that is mapped to the same L2 cache set as $r.m_{L2}$.*

Definition 7.3.4. *A memory block is demanded on a path – some information in this memory block is needed by some reference occurring on the path.*

Lemma 7.3.5. *For a memory reference r at a program point p in a loop, if r is a positive reference (i.e., $r.m_{L2}$ is definitely or persistently in L2 cache at p), then at least one of the following two cases holds.*

Case A: *There are fewer than A_{L2} distinct conflicting memory blocks of $r.m_{L2}$ demanded on any cyclic path $p \rightsquigarrow p$.*

Case B: *If there is a cyclic path $p \rightsquigarrow p$ on which there are at least A_{L2} distinct conflicting memory blocks of $r.m_{L2}$ demanded, there is at least one reference to $r.m_{L2}$ classified as L1* AM *at some program point q ($q \neq p$) on the path.*

Proof. We prove the statement

$$r \text{ is a positive reference} \implies \text{Case A} \bigvee \text{Case B}$$

by contraposition, namely we are to prove

$$\neg\text{Case A} \bigwedge \neg\text{Case B} \implies r \text{ is not a positive reference}$$

We have ¬Case A as "there are at least A_{L2} distinct conflicting memory blocks of $r.m_{L2}$ demanded on some cyclic path $p \rightsquigarrow p$ in the loop". We also have ¬Case B as "given any cyclic path $p \rightsquigarrow p$ on which there are at least A_{L2} distinct conflicting memory blocks of $r.m_{L2}$ demanded, there is no reference to $r.m_{L2}$ classified as L1 *AM* at any program point $q \neq p$ on the path".

Let us assume some path $p \rightsquigarrow p$ on which there are at least A_{L2} distinct conflicting memory blocks of $r.m_{L2}$ demanded is ρ. According to ¬Case B, there is no reference other than r to $r.m_{L2}$ classified as L1 *AM* on ρ. Due to the inclusion property, at least $A_{L2}+1$ distinct memory blocks needs to appear in the corresponding L2 cache set. Therefore, at least one memory block, say m_z, among these distinct memory blocks is not definitely/persistently in L2 cache at the end of the path ρ (i.e. p since the path ρ is cyclic). Therefore, we have: (1) If m_z is $r.m_{L2}$, it means $r.m_{L2}$ is not definitely/persistently in L2 cache at p, so r is not a positive reference. (2) If m_z is not $r.m_{L2}$, since m_z is not in L2 cache at p (i.e. the beginning of the path ρ) and m_z is demanded on the path ρ, after m_z is first referenced on the path, m_z will become the youngest in the cache set, namely it is possible that m_z is younger than $r.m_{L2}$ at this point. Since the *must* and *persistence* analyses are safe, in the abstract set states of the *must* and *persistence* analyses, $r.m_{L2}$ should not be younger than m_z at this point. Since there is no reference to $r.m_{L2}$ classified as L1 *AM* at any program point $q \neq p$ on the path, $r.m_{L2}$ cannot be put into the first position of the abstract set states. Therefore, at the end of the path ρ, $r.m_{L2}$ is still treated as possibly older than m_z. Since

m_z is safely classified as "not definitely/persistently in L2 cache at p", $r.m_{L2}$ should also be classified as "not definitely/persistently in L2 cache at p" with respect to the safety of the analyses. Therefore, r is not a positive reference. □

Theorem 7.3.6. *Given a positive reference r at a program point p in a loop, if **Case A** holds, it can suffer at most one L2 cache miss in the presence of preemption.*

Proof. We prove this statement by contradiction. Let us assume r can suffer at least two L2 cache misses in the presence of preemption. After r's first L2 cache miss, $r.m_{L2}$ becomes the youngest memory block in the corresponding cache set. Since r can suffer at least another L2 cache miss, it means $r.m_{L2}$ will be evicted from L2 cache. Therefore, there are at least A_{L2} conflicting memory blocks of $r.m_{L2}$ demanded on some path $p \rightsquigarrow p$, which means **Case A** does not hold. Therefore, we reach a contradiction. □

As shown above, when there exists inter-task interference, we cannot guarantee *AM* classifications safe anymore. If only **Case B** holds for a positive reference in a loop, **Case B** may not hold in the presence of preemption since L1 *AM* may not hold, which means the reference's "positiveness" may not hold. Therefore, the number of times that a positive reference r at a program point p in a loop $\mathscr{L}$ may be declined is given by $dec(\mathscr{L}, r)$ which is defined as:

$$dec(\mathscr{L}, r) = \begin{cases} 1 & \text{if } cmb(\mathscr{L}, r.m_{L2}) < A_{L2} \\ lb(\mathscr{L}) & \text{otherwise} \end{cases}$$

where $cmb(\mathscr{L}, r.m_{L2})$ overestimates the maximum distinct conflicting memory blocks of $r.m_{L2}$ demanded on any cyclic path $p \rightsquigarrow p$ in the loop $\mathscr{L}$, and $lb(\mathscr{L})$ gives the loop bound of $\mathscr{L}$. While $lb(\mathscr{L})$ can be manually input or estimated by other techniques which is not in the scope of study, $cmb(\mathscr{L}, r.m_{L2})$ is defined as:

$$cmb(\mathscr{L}, r.m_{L2}) = |\{m | m \in \mathscr{L} \bigwedge m \neq r.m_{L2} \bigwedge m \text{ is mapped to the same L2 cache set as } r.m_{L2}\}|$$

which calculates in the loop how many memory blocks other than $r.m_{L2}$ are mapped to

the same L2 cache set as $r.m_{L2}$. Since only a subset of these memory blocks is demanded on any cyclic path in the loop, $cmb(\mathscr{L}, r.m_{L2})$ over-approximates the number of distinct conflicting memory blocks of $r.m_{L2}$ demanded on any path $p \rightsquigarrow p$.

7.3.1.2 Useful Positive References

As stated in Section 7.1, at a program point, instead of memory blocks that may be referenced beyond the point, we need to focus on positive references that may appear after this point in the context of multi-level caches. Therefore, we use the term "useful positive references" (UPRs) instead of UCBs to capture the possible CRPD contributors.

Definition 7.3.7. *At a program point, a useful positive reference (UPR) is a positive reference that is reachable from the program point.*

Similar to how to capture UCBs at a program point, we use a backward-flow analysis to capture UPRs at the point. The domain $\mathbb{D}_{\text{UPR}}$ of the analysis is defined as:

$$\mathbb{D}_{\text{UPR}} = \mathbb{R}^+ \rightarrow (2^{\mathbb{R}^+} \times \mathbb{B})_\perp$$

where $\mathbb{R}^+$ is the set of all the positive references in the preempted task, $\mathbb{B} = \{\text{true}, \text{false}\}$, and $(2^{\mathbb{R}^+} \times \mathbb{B})_\perp$ denotes the product domain $2^{\mathbb{R}^+} \times \mathbb{B}$ is lifted by adding a bottom element $\perp$. Given an element $\alpha \in \mathbb{D}_{\text{UPR}}$ and a positive reference $r \in \mathbb{R}^+$ at a program point with $\alpha(r) \neq \perp$, the first component of $\alpha(r)$ is denoted by $\alpha(r)_1$ which is an overestimated set of references that may have negative effect on the "positiveness" of r only in the presence of preemption (potential additional intra-task interference to r in L2 cache), and the second component of $\alpha(r)$ is denoted by $\alpha(r)_2$ which gives whether the first component can be expanded at the program point.

Let $\mathbb{R}$ denote the set of all the references in the preempted task, so we have $\mathbb{R}^+ \subseteq \mathbb{R}$. The *update* function $\mathscr{U}_{\text{UPR}} : \mathbb{D}_{\text{UPR}} \times \mathbb{R} \rightarrow \mathbb{D}_{\text{UPR}}$ of the backward-flow analysis is used to take into account the effects of a reference on the useful positive references. The function

is defined as:

$$\mathscr{U}_{\text{UPR}}(\alpha, r) = \begin{cases} \alpha & \text{if } r \notin \mathbb{R}^+ \bigwedge r.c_{L2} \neq DAM \\ \alpha[r' \mapsto \langle \alpha(r')_1, \text{false} \rangle | r'.m_{L2} = r.m_{L2}] & \text{if } r \notin \mathbb{R}^+ \bigwedge r.c_{L2} = DAM \\ \alpha[r \mapsto \langle \emptyset, \text{true} \rangle] & \text{if } r \in \mathbb{R}^+ \bigwedge r.c_{L1} \neq AH \\ \alpha[r \mapsto \langle \emptyset, \text{true} \rangle, & \text{otherwise} \\ \quad r' \mapsto \langle \alpha(r')_1 \cup \{r\}, \alpha(r')_2 \rangle | & \\ \quad \alpha(r')_2 = \text{true} \bigwedge r'.m_{L2} \neq r.m_{L2} \bigwedge r.a_{L2}^{may} \geq & \\ \quad \min(r'.m_{L2}\text{'s } \textit{must} \text{ and } \textit{pers} \text{ ages at } r)] & \end{cases}$$

where "$r'.m_{L2}$'s *must* age at r" is the position where the memory block $r'.m_{L2}$ is in the abstract set state of the *must* analysis corresponding to where r is occurring, and similarly "$r'.m_{L2}$'s *persistence* age at r" is the position where the memory block $r'.m_{L2}$ is in the abstract set state of the *persistence* analysis corresponding to where r is occurring. If $r'.m_{L2}$ is not definitely cached at r, its *must* age is $\top_{L2}$, so the min function returns "$r'.m_{L2}$'s *persistence* age at r".

The definition of the *update* function $\mathscr{U}_{\text{UPR}}$ has four pieces. The updating of state α depends on whether the reference r is a positive reference and its cache behavior. If r is not a positive reference, when performing WCET analysis, it cannot be classified as *never* accessing L2 cache, so its effect on aging the definitely/persistently cached memory blocks has been taken into account. Even if in reality r does not access L2 cache in the absence of preemption but accesses L2 cache in the presence of preemption, the "positiveness" of r' that is a positive reference after r will not be affected by r. When $r.c_{L2}$ is classified as *DAM*, its referenced memory block $r.m_{L2}$ will definitely become the youngest memory block in the corresponding cache set. Therefore, r serves as a boundary beyond which we stop increasing the potential additional intra-task interference to any positive reference to $r.m_{L2}$. To indicate that we stop expanding the potential interference set for a reference r',

we set the $\alpha(r')_2$ as false.

If r is a positive reference, we need to track its potential additional intra-task interference from this point during the backward-flow analysis, so we set $\alpha(r)_1$ as an empty set and set $\alpha(r)_2$ as true. If $r.c_{L1}$ is classified *AH*, the WCET analysis treats it as *never* accessing L2 cache. Therefore, if the *AH* classification may not hold in the presence of preemption, it may access L2 cache so that it may cause additional intra-task interference to the following positive references. However, if r's *may* age is less than the *must* or *persistence* ages of a memory block referenced by a positive reference, the effect on aging this memory block has been taken into account. Therefore, for a positive reference, we only track the *AH* references whose referenced memory blocks have never affected the *must* or *persistence* ages of the memory block referenced by this positive reference.

The *join* function $\mathscr{J}_{\text{UPR}} : \mathbb{D}_{\text{UPR}} \times \mathbb{D}_{\text{UPR}} \to \mathbb{D}_{\text{UPR}}$ is used to merge information at a join point in the backward-flow analysis (namely at a branching point of the program). The function is defined as:

$$\mathscr{J}_{\text{UPR}}(\alpha_1, \alpha_2) = \Big[r \mapsto \begin{cases} \bot & \text{if } \alpha_1(r) = \bot \bigwedge \alpha_2(r) = \bot \\ \alpha_1(r) & \text{if } \alpha_1(r) \neq \bot \bigwedge \alpha_2 = \bot \\ \alpha_2(r) & \text{if } \alpha_1(r) = \bot \bigwedge \alpha_2 \neq \bot \\ \langle \alpha_1(r)_1 \cup \alpha_2(r)_1, \alpha_1(r)_2 \vee \alpha_2(r)_2 \rangle & \text{otherwise} \end{cases} \Big]$$

where we use set union to combine the potential interference set for each positive reference and use boolean OR operation to combine the indicators.

7.3.2 Preempting Task Analysis

When deriving the CRPD estimate, we need to know how much interference a preempting task can maximally cause to the preempted task. Therefore, we need to find an approach to over-approximate the amount of interference caused by the preempting task

(i.e. the maximum number of used cache blocks) in the context of multi-level inclusive caches.

The approach used in [104] first performs the *may* analysis of the preempting task on the multi-level non-inclusive cache. The maximum number of cache blocks used in each cache set at each cache level can be overestimated by the number of memory blocks in the corresponding abstract set state of *may* analysis at the exit point of the preempting task. This approach is sound, because memory blocks can only be evicted from a cache level in a multi-level non-inclusive cache. However, since there may be invalidations in multi-level inclusive caches, the amount of interference may be under-approximated if only considering how many memory blocks are in the abstract set state of *may* analysis.

Instead of the exact inter-task interference contents, we are only concerned about how much aging in each cache set at each cache level is maximally caused by the preempting task. Therefore, we can achieve this through checking the possible "watermarks" in each cache set due to loading the memory blocks of the preempting task. For a cache set, the maximal "watermark" can be interpreted as the maximum number of cache blocks ever used in that cache set by the preempting task.

To this end, we propose to use the abstract states of the *persistence* analysis at the exit point of the preempting task. Given the preempting task $\hat{\tau}$, let $\beta_{L1}^{\hat{\tau}}$ (resp. $\beta_{L2}^{\hat{\tau}}$) denote the abstract L1 (resp. L2) cache state of the *persistence* analysis at the exit point. For a memory block m in L1 cache, we have the over-approximated amount of interference caused by $\hat{\tau}$ in this cache set as:

$$ecb_{L1}^{\hat{\tau}}(m) = \begin{cases} A_{L1} & \text{if } \beta_{L1}^{\hat{\tau}}(i)(\top_{L1}) \neq \emptyset \\ \max\{j | \beta_{L1}^{\hat{\tau}}(i)(j) \neq \emptyset\} & \text{otherwise} \end{cases} \text{ where } m \text{ is mapped to the } i^{th} \text{ set}$$

For the memory block m, which is mapped to the i^{th} cache set, the $ecb_{L1}^{\hat{\tau}}$ function (named after the traditional "evicting cache blocks") gives A_{L1} if the $\top_{L1}$ position has memory blocks, which means the preempting task uses all the cache blocks in the i^{th} set; otherwise,

the function gives the biggest position which is mapped to a non-empty set of memory blocks by the abstract state $\beta_{L1}^{\hat{\tau}}$. Note that if the preempting task $\hat{\tau}$ never used the i^{th} cache set of L1 cache, $\{j|\beta_{L1}^{\hat{\tau}}(i)(j) \neq \emptyset\}$ would be an empty set; and we define $\max \emptyset = 0$. The corresponding $ecb_{L2}^{\hat{\tau}}$ function can be defined in the same manner.

7.3.3 CRPD Estimation

When we complete the analyses of the preempted task τ and the preempting task $\hat{\tau}$, we can estimate the CRPD by counting how many positive references may be declined at each program point. The maximum number of possibly declined positive references multiplied by the corresponding miss penalties is the CRPD estimate. The algorithm is given in Algorithm 7.1.

At a program point p, given the computed UPR state α and the computed inter-task interference functions $ecb_{L1}^{\hat{\tau}}$ and $ecb_{L2}^{\hat{\tau}}$, we estimate the CRPD $crpd_p$ by checking each positive reference to see if it can preserve its "positiveness". The estimated CRPD is stored in a variable $crpd_p$ which may be increased by one or more L1 cache reload times t_{L1} and/or L2 cache reload times t_{L2} for a positive reference.

We start checking each positive reference r having the smallest potential interference set (e.g. a positive reference r whose $\alpha(r)_1 = \emptyset$ will be processed first) (line 2). Since a positive reference r may be executed multiple times, line 4 – line 5 derive the upper bound on the number of times r may be declined. Recall that $dec(\mathscr{L}, r)$ gives the number of possible declination of r if r is in the loop $\mathscr{L}$. We use variables age_{L1}^{must}, age_{L2}^{must}, age_{L1}^{pers}, and age_{L2}^{pers} to capture the *must* and *persistence* ages of the referenced memory blocks in the presence of preemption (line 6 – line 12). If $\alpha(r)_1 \neq \emptyset$, there are cyclic dependencies since r has the smallest potential interference set. We break this cycle by pessimistically assuming every potential interference in the set may cause the ages to increase (line 8 – line 9). If $\alpha(r)_2 = true$, r has not met a *DAM* reference to $r.m_{L2}$ yet. Therefore, it may be affected by the inter-task interferences. We overestimate the *must* and *persistence* ages by

Algorithm 7.1: Estimate the CRPD at a program point p of τ preempted by $\hat{\tau}$

Input: p, α, $ecb^{\hat{\tau}}_{L1}$, $ecb^{\hat{\tau}}_{L2}$
Result: $crpd_p$

$crpd_p \leftarrow 0$;
repeat
  get r whose $\alpha(r)_1$ has the fewest elements ;
  $num \leftarrow 1$;
  if *r is in a loop $\mathscr{L}$* **then** $num \leftarrow dec(\mathscr{L}, r)$;
  $age^{must}_{L1} \leftarrow r.a^{must}_{L1}$ and $age^{must}_{L2} \leftarrow r.a^{must}_{L2}$;
  $age^{pers}_{L1} \leftarrow r.a^{pers}_{L1}$ and $age^{pers}_{L2} \leftarrow r.a^{pers}_{L2}$;
  if $\alpha(r)_1 \neq \emptyset$ **then**
    $age^{must}_{L2} \leftarrow age^{must}_{L2} + |\alpha(r)_1|$ and $age^{pers}_{L2} \leftarrow age^{pers}_{L2} + |\alpha(r)_1|$;
  if $\alpha(r)_2 =$ true **then**
    $age^{must}_{L1} \leftarrow age^{must}_{L1} + ecb^{\hat{\tau}}_{L1}(r.m_{L1})$ and $age^{must}_{L2} \leftarrow age^{must}_{L2} + ecb^{\hat{\tau}}_{L2}(r.m_{L2})$;
    $age^{pers}_{L1} \leftarrow age^{pers}_{L1} + ecb^{\hat{\tau}}_{L1}(r.m_{L1})$ and $age^{pers}_{L2} \leftarrow age^{pers}_{L2} + ecb^{\hat{\tau}}_{L2}(r.m_{L2})$;
  if $r.c_{L1} = AH \bigwedge r.c_{L2} = AH$ **then**
    $flag \leftarrow$ false ;
    if $age^{must}_{L1} > A_{L1} \bigwedge age^{must}_{L2} \leq A_{L2}$ **then**
      $crpd_p \leftarrow crpd_p + t_{L1}$;
      $flag \leftarrow$ true ;
    else if $age^{must}_{L2} > A_{L2}$ **then**
      $crpd_p \leftarrow crpd_p + (t_{L1} + t_{L2}) \times num$;
      $flag \leftarrow$ true ;
    foreach *r' with $r \in \alpha(r')_1$* **do**
      remove r from $\alpha(r')_1$;
      if $flag =$ true **then**
        if $r'.c_{L2} = AH$ **then** $r'.a^{must}_{L2} \leftarrow r'.a^{must}_{L2} + 1$;
        else $r'.a^{pers}_{L2} \leftarrow r'.a^{pers}_{L2} + 1$;
  else if $r.c_{L1} = PS \bigwedge r.c_{L2} = AH$ **then**
    if $age^{pers}_{L1} > A_{L1} \bigwedge age^{must}_{L2} \leq A_{L2}$ **then**
      if *p is in the same loop $\mathscr{L}$ as r* **then** $crpd_p \leftarrow crpd_p + t_{L1}$;
    else if $age^{must}_{L2} > A_{L2}$ **then**
      if *p is in the same loop $\mathscr{L}$ as r* **then**
        if $num \neq 1$ **then** $crpd_p \leftarrow crpd_p + t_{L1} \times (num - 1) + t_{L2} \times num$;
        else $crpd_p \leftarrow crpd_p + t_{L1} + t_{L2}$;
      else $crpd_p \leftarrow crpd_p + t_{L1} \times (num - 1) + t_{L2} \times num$;
  else if $r.c_{L2} = AH$ **then**
    if $age^{must}_{L2} > A_{L2}$ **then** $crpd_p \leftarrow crpd_p + t_{L1} \times num$;
  else if $r.c_{L1} = PS \bigwedge r.c_{L2} = PS$ **then**
    if $age^{pers}_{L1} > A_{L1} \bigwedge age^{pers}_{L2} \leq A_{L2}$ **then**
      if *p is in the same loop $\mathscr{L}$ as r* **then** $crpd_p \leftarrow crpd_p + t_{L1}$;
    else if $age^{pers}_{L2} > A_{L2}$ **then**
      if *p is in the same loop $\mathscr{L}$ as r* **then**
        if $num \neq 1$ **then** $crpd_p \leftarrow crpd_p + (t_{L1} + t_{L2}) \times (num - 1)$;
        else $crpd_p \leftarrow crpd_p + t_{L1} + t_{L2}$;
      else $crpd_p \leftarrow crpd_p + (t_{L1} + t_{L2}) \times (num - 1)$;
  else if $r.c_{L2} = PS$ **then**
    if $age^{pers}_{L2} > A_{L2}$ **then**
      if *p is in the same loop $\mathscr{L}$ as r* **then**
        if $num \neq 1$ **then** $crpd_p \leftarrow crpd_p + t_{L2} \times (num - 1)$;
        else $crpd_p \leftarrow crpd_p + t_{L2}$;
      else $crpd_p \leftarrow crpd_p + t_{L2} \times (num - 1)$;
  $\alpha(r) \leftarrow \perp$;
until $\forall r \in \mathbb{R}^+ : \alpha(r) = \perp$;

adding the number of memory blocks introduced by $\hat{\tau}$.

Next we compute the possible contribution of r to CRPD estimate according to the type of r (shown in Tab. 7.2). The case of Type 1 (i.e. L1 *AH* and L2 *AH*) is considered in line 13 – line 25. Since a positive reference of this type may also cause an additional L2 intra-task interference to other positive references, we use a $flag$ to indicate whether this additional intra-task interference is possible to happen. If the condition $age_{L1}^{must} > A_{L1} \bigwedge age_{L2}^{must} \leq A_{L2}$ holds, r may not be L1 *AH* due to the preemption. Thus, this positive reference is possible to contribute an additional t_{L1} to $crpd_p$. Since it may access L2 cache in the presence of preemption, the $flag$ is set true to indicate it may increase the amount of intra-task interference to some other references. If the condition $age_{L2}^{must} > A_{L2}$ holds, r may not be L1 *AH* and L2 *AH* due to the inclusion property. Thus, we add $t_{L1} + t_{L2}$ by the number of times r may be declined to $crpd_p$. Similarly, the $flag$ is set true to indicate its negative impact on the "positiveness" of some other references. Since r may be in other positive references' potential interference sets, we remove it from these sets (line 21 – line 25). If the $flag$ was set true, when we remove r from other positive references' potential interference sets, we also increase the corresponding memory block age by 1 to take into account r's effect.

The case of Type 2 (i.e. L1 *PS* and L2 *AH*) is considered in line 26 – line 33. If the condition $age_{L1}^{pers} > A_{L1} \bigwedge age_{L2}^{must} \leq A_{L2}$ holds, r may suffer an additional t_{L1} only if p is in the same loop as r (i.e. the loop $\mathscr{L}$) and the preemption occurs at p during the iterations other than the first iteration (recall that *PS* classification implies the reference is in a loop). This is because if p is out of the loop $\mathscr{L}$, the WCET analysis has already taken into account an L1 cache miss for r. If the condition $age_{L2}^{must} > A_{L2}$ holds, we need to take into account the additional $t_{L1} + t_{L2}$ for one possible declination multiplied by the declination bound. If p is in the loop $\mathscr{L}$, even if *num* is 1, we would also need to add $t_{L1} + t_{L2}$ to $crpd_p$ once, since the preemption may happen at p during the iterations other than the first iteration; if *num* is not 1 (i.e. the loop bound), we add delays to $crpd_p$ which correspond to that not considered by the WCET analysis. If p is not in the loop, the delays that added to $crpd_p$

include one more t_{L2} than t_{L1}, since the WCET analysis has already considered one L1 cache miss for r due to L1 *PS* classification.

The cases of Type 3 (i.e. L1 *AM* and L2 *AH*) and Type 4 (i.e. L1 *NC* and L2 *AH*) are considered in line 34 – line 35. In both cases, the WCET analysis treats all the r's L1 accesses as cache misses. Thus, we only need to consider the possible additional delays to reload $r.m_{L2}$ into L2 cache when the condition $age_{L2}^{must} > A_{L2}$ holds.

The case of Type 5 (i.e. L1 *PS* and L2 *PS*) is considered in line 36 – line 43. If the condition $age_{L1}^{pers} > A_{L1} \bigwedge age_{L2}^{pers} \leq A_{L2}$ holds, it (line 37 – line 38) is the same as that considered in line 27 – line 28 in the case of Type 2. If the condition $age_{L2}^{pers} > A_{L2}$ holds, it (line 39 – line 43) is also similar to that considered in line 29 – line 33 in the case of Type 2, except we add one less t_{L2} to $crpd_p$ when p is not in the loop $\mathscr{L}$ or p is in the loop but *num* is not 1. This is because the WCET analysis has taken into account one L2 cache miss for r which is in the loop $\mathscr{L}$.

The cases of Type 6 (i.e. L1 *AM* and L2 *PS*) and Type 7 (i.e. L1 *NC* and L2 *PS*) are considered in line 44 – line 49. Other than the WCET analysis has treated all the r's L1 accesses as cache misses, the rest is much similar to that considered in line 39 – line 43 in the case of Type 5.

In the end, we mark r as processed (line 50), and the for-loop processes the next positive reference whose potential interference set is empty.

7.4 Conclusion and Future Work

In this chapter, we investigate how to bound CRPD in the context of multi-level inclusive caches. We show that there are different challenges in CRPD analysis for inclusive cache hierarchies, so the traditional methods of CRPD analysis for single-level caches and non-inclusive cache hierarchies cannot be used directly. In order to analyze CRPD for multi-level inclusive caches, we propose to use a concept of useful positive references. At a program point, the set of reachable useful positive references is derived through a

backward-flow analysis. We define two functions (the *update* function and the *join* function) for this backward-flow analysis. For a positive reference in a loop, we give an upper bound on the number of times this reference may not preserve its "positiveness". We also propose a "watermark" method to analyze the preempting task in the context of inclusive cache hierarchies. In the end, we describe how to derive the CRPD estimate according to the performed analyses.

In the future, we plan to complete this study by evaluating our approach on a set of benchmarks and we also plan to investigate the effect of CRPD on the schedulability analysis of fixed-priority scheduling algorithms (e.g. rate-monotonic) in the context of multi-level inclusive caches.

Chapter 8

Conclusion

In this thesis, we have studied how to analyze and verify CPS software using static analysis. Since CPS software must be analyzed at a level of abstraction that captures details of the computational platform, we focus on its low-level code. Two types of properties are investigated – one type is of numeric values and the other one is of timing.

In terms of numeric analysis, we choose to perform value-set analysis (VSA) on the low-level code of CPS software. We extend the original strided-interval domain to more precisely track the set of structured numbers, and also define the operations on this extended strided-interval domain. In order to achieve generic VSA, we present the syntax and concrete semantics of an intermediate language, which can be used to encode the instructions of different instruction set architectures. We define the abstract semantics for the intermediate language, and discuss how to use it in VSA.

In terms of timing analysis, we focus on how to safely and precisely derive the worst-case execution time (WCET) and preemption delay for a task in the presence of single-level or multi-level caches. Since most of the execution time of a task is spent in loops, we need to perform safe and precise cache *persistence* analysis. We identify the sources of pessimism of state-of-the-art approaches for cache *persistence* analysis and propose methods to eliminate these sources in order to improve the analysis precision. When the task runs on a processor equipped with multi-level inclusive caches, invalidations between cache levels may happen, which make precise WCET estimation much harder. We propose two approaches to improve the precision of the WCET estimation in the presence of inclusive cache hierarchies and also prove the proposed methods are still safe. In addition, CPS software usually consists of multiple tasks which are scheduled in a preemptive manner. Therefore, we also investigate how to bound the cache-related preemption delay for a task

due to interferences introduced by other tasks.

As stated in the introduction, analysis and verification of CPS software is very challenging. When analyzing CPS software in its low-level form, different properties may require the consideration of different underlying hardware details. For functional property verification, we may only consider the semantics of each instruction, and there can be a generic approach applied to different underlying hardware architectures, namely we may not care about how a value is computed but what the value may be. For non-functional property verification, we need to take into account different micro-architectural features and their interactions during the computation. Therefore, for one specific micro-architectural feature, we may need one method to take into account this feature; and for a processor with multiple features, we need to compose a specific approach to perform the analysis.

In the future, we plan to study how to utilize value analysis to verify many numeric properties of control software. We also plan to take into account more micro-architectural features in the timing analysis.

Appendix A

Software

The static analyzer we are developing is called "Vandalizer" (Vanderbilt anali(y)zer). The main components that have already been developed include:

- Program Representation
  - The binary is manipulated and disassembled using BFD library.
  - Each instruction is represented according to the encoding (see Chapter 3).
  - Two types of graphs are built for the binary.
    - A control-flow graph is built for each procedure (see [18]).
    - A context-sensitive call graph is built for the program (see [125]).
  - A dominator tree is built for identifying loops (see [126, 127]).
- Program Analysis
  - Worklist algorithm is used to derived fixed-points.
  - Data values of a location can be derived (see Chapter 3).
  - Multi-level cache behavior can be analyzed (see Chapter 5 and 6).
    - LRU replacement policy is currently supported (see [53] and Chapter 4).
  - Implicit path enumeration is used to derive WCET (see [83]).
    - ILP in AMPL is generated (any solver which supports AMPL can be used).

BIBLIOGRAPHY

[1] CPS Summit Report.

[2] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 363–369, 2008.

[3] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and Shige Wang. Toward a Science of Cyber-Physical System Integration. *Proceedings of the IEEE*, 100(1):29–44, January 2012.

[4] Zhenkai Zhang, Joseph Porter, Emeka Eyisi, Gabor Karsai, Xenofon Koutsoukos, and Janos Sztipanovits. Co-simulation Framework for Design of Time-triggered Cyber Physical Systems. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 119–128, 2013.

[5] Zhenkai Zhang, Emeka Eyisi, Xenofon Koutsoukos, Joseph Porter, Gabor Karsai, and Janos Sztipanovits. A co-simulation framework for design of time-triggered automotive cyber physical systems. *Simulation Modelling Practice and Theory*, 43(0):16 – 33, 2014.

[6] Emeka Eyisi, Zhenkai Zhang, Xenofon Koutsoukos, Joseph Porter, Gabor Karsai, and Janos Sztipanovits. Model-Based Control Design and Integration of Cyber-Physical Systems: An Adaptive Cruise Control Case Study. *Journal of Control Science and Engineering*, 2013.

[7] William Landi. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[8] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[9] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, 1977.

[10] Zhenkai Zhang and Xenofon Koutsoukos. Generic Value-Set Analysis on Low-Level Code. In *Proceedings of the 5th Analytic Virtual Integration of Cyber-Physical Systems Workshop*, AVICPS '14, 2014.

[11] Zhenkai Zhang and Xenofon Koutsoukos. Improving the Precision of Abstract Interpretation Based Cache Persistence Analysis. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM*, LCTES'15, pages 10:1–10:10, 2015.

[12] Zhenkai Zhang and Xenofon Koutsoukos. Top-Down and Bottom-Up Multi-Level Cache Analysis for WCET Estimation. In *Proceedings of the 2015 IEEE 21st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '15, pages 24–35, 2015.

[13] Zhenkai Zhang and Xenofon Koutsoukos. Precise Multi-Level Inclusive Cache Analysis for WCET Estimation. In *Proceedings of the 2015 IEEE 36th Real-Time Systems Symposium*, RTSS '15, 2015.

[14] Cristina Cifuentes and Mike Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Proceedings of the 7th International Workshop on Program Comprehension*, IWPC '99, pages 192–, 1999.

[15] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural Static Slicing of Binary

Executables. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 188–, 1997.

[16] Daniel Kästner and Stephan Wilhelm. Generic Control Flow Reconstruction from Assembly Code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, pages 46–55, 2002.

[17] Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, P Keyngnaert, and B Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2, CSREA Press, Las Vegas, 2000, 1013-1019*, 2000.

[18] 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–, 2000.

[19] Johannes Kinder, Florian Zuleger, and Helmut Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 214–228, 2009.

[20] Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural Control Flow Reconstruction. In Kazunori Ueda, editor, *Programming Languages and Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 188–203. 2010.

[21] Johannes Kinder and Dmitry Kravchenko. Alternating Control Flow Reconstruction. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 267–282, 2012.

[22] Thomas Reinbacher and Jörg Brauer. Precise Control Flow Reconstruction Using Boolean Logic. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 117–126, 2011.

[23] Edd Barrett and Andy King. Range and Set Abstraction Using SAT. *Electron. Notes Theor. Comput. Sci.*, 267(1):17–27, October 2010.

[24] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Generalized Type Unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 77–94, 1977.

[25] T. Hickey, Q. Ju, and M. H. Van Emden. Interval Arithmetic: From Principles to Implementation. *J. ACM*, 48(5):1038–1068, September 2001.

[26] Antoine Miné. The Octagon Abstract Domain. *Higher Order Symbol. Comput.*, 19(1):31–100, March 2006.

[27] Francesco Logozzo and Manuel Fähndrich. Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 184–188, 2008.

[28] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Sci. Comput. Program.*, 72(1-2):3–21, June 2008.

[29] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[30] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation Recovery from Low-level Code. In *Proceedings of the 2006 ACM SIGPLAN Sym-*

posium on Partial Evaluation and Semantics-based Program Manipulation, PEPM '06, pages 100–111, 2006.

[31] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. CodeSurfer/x86 – A Platform for Analyzing x86 Executables. In *Proceedings of the 14th International Conference on Compiler Construction*, CC '05, pages 250–254, 2005.

[32] Rathijit Sen and Y. N. Srikant. Executable Analysis Using Abstract Interpretation with Circular Linear Progressions. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE '07, pages 39–48, 2007.

[33] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *Proceeding of CanSecWest*, 2009.

[34] Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 357–366, 2011.

[35] Julian Kranz, Alexander Sepp, and Axel Simon. GDSL: A Universal Toolkit for Giving Semantics to Machine Language. In Chung-chieh Shan, editor, *Programming Languages and Systems*, volume 8301 of *Lecture Notes in Computer Science*, pages 209–216. Springer International Publishing, 2013.

[36] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, 2011.

[37] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena.

BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, December 2008.

[38] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The BINCOA Framework for Binary Code Analysis. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 165–170, 2011.

[39] Jörg Brauer, René Rydhof Hansen, Stefan Kowalewski, Kim G. Larsen, and Mads Chr. Olesen. Adaptable Value-Set Analysis for Low-Level Code. In *6th International Workshop on Systems Software Verification*, volume 24 of *OpenAccess Series in Informatics (OASIcs)*, pages 32–43, 2012.

[40] AbsInt. aiT WCET Analyzers.

[41] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[42] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, 2011.

[43] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[44] N. Zhang, A. Burns, and M. Nicholson. Pipelined Processors and Worst Case Execution Times. *Real-Time Syst.*, 5(4):319–343, October 1993.

[45] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, July 1995.

[46] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 334–, 1998.

[47] Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, pages 35–44, 1999.

[48] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling Out-of-order Processors for WCET Analysis. *Real-Time Syst.*, 34(3):195–227, November 2006.

[49] Antoine Colin and Isabelle Puaut. Worst Case Execution Time Analysis for a Processor withBranch Prediction. *Real-Time Syst.*, 18(2/3):249–274, May 2000.

[50] Iain Bate and Ralf Reutemann. Worst-Case Execution Time Analysis for Dynamic Branch Predictors. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, pages 215–222, 2004.

[51] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling Control Speculation for Timing Analysis. *Real-Time Syst.*, 29(1):27–58, January 2005.

[52] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 12–, 1999.

[53] Christian Ferdinand and Reinhard Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-TimeSystems. *Real-Time Syst.*, 17(2-3):131–181, December 1999.

[54] Daniel Grund and Jan Reineke. Toward Precise PLRU Cache Analysis. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, WCET '10, pages 23–35, 2010.

[55] David Griffin, Benjamin Lesage, Alan Burns, and Robert I. Davis. Lossy Compression for Worst-Case Execution Time Analysis of PLRU Caches. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 203:203–203:212, 2014.

[56] Daniel Grund and Jan Reineke. Abstract Interpretation of FIFO Replacement. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 120–136, 2009.

[57] Daniel Grund and Jan Reineke. Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection. In *Proceedings of the 2010 22Nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 155–164, 2010.

[58] Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. FIFO Cache Analysis for WCET Estimation: A Quantitative Approach. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 296–301, 2013.

[59] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET Analysis with MRU Caches: Challenging LRU for Predictability. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS '12, pages 55–64, 2012.

[60] Jan Reineke and Daniel Grund. Relative Competitive Analysis of Cache Replacement Policies. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on*

Languages, Compilers, and Tools for Embedded Systems, LCTES '08, pages 51–60, 2008.

[61] Damien Hardy and Isabelle Puaut. WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, pages 456–466, 2008.

[62] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET ANALYSIS OF MULTI-LEVEL SET-ASSOCIATIVE DATA CACHES. In *9th Intl. Workshop on Worst-Case Execution Time WCET Analysis*, page 2283, November 2009.

[63] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *Proceedings of 2009 IEEE 30th Real-Time Systems Symposium*, RTSS '09, pages 68–77, 2009.

[64] Damien Hardy and Isabelle Puaut. WCET Analysis of Instruction Cache Hierarchies. *J. Syst. Archit.*, 57(7):677–694, August 2011.

[65] Yerang Hur, Young Hyun Bae, Sung-Soo Lim, Sung-Kwan Kim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Minsuk Lee, Heonshik Shin, and Chong Sang Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, pages 308–, 1995.

[66] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proceedings of the 2Nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, RTAS '96, pages 230–, 1996.

[67] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 317–324, 1997.

[68] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, 1991.

[69] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact Analysis of the Cache Behavior of Nested Loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 286–297, 2001.

[70] Harini Ramaprasad and Frank Mueller. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, RTAS '05, pages 148–157, 2005.

[71] Randall T. White, Frank Mueller, Chris Healy, David Whalley, and Marion Harmon. Timing Analysis for Data and Wrap-Around Fill Caches. *Real-Time Syst.*, 17(2-3):209–233, December 1999.

[72] Christian Ferdinand and Reinhard Wilhelm. On Predicting Data Cache Behavior for Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 16–30, 1998.

[73] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-Aware Data Cache Analysis for WCET Estimation. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 203–212, 2011.

[74] Christoph Cullmann. Cache Persistence Analysis: A Novel Approach Theory and Practice. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '11, pages 121–130, 2011.

[75] Rathijit Sen and Y. N. Srikant. WCET Estimation for Executables in the Presence of Data Caches. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 203–212, 2007.

[76] Christoph Cullmann. Cache Persistence Analysis: Theory and Practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, March 2013.

[77] Sudipta Chattopadhyay and Abhik Roychoudhury. Unified Cache Modeling for WCET Analysis and Layout Optimizations. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 47–56, 2009.

[78] Tyler Sondag and Hridesh Rajan. A More Precise Abstract Domain for Multi-level Caches for Tighter WCET Analysis. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 395–404, 2010.

[79] Christine Rochange. An Overview of Approaches Towards the Timing Analysability of Parallel Architecture. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 32–41, 2011.

[80] Jun Yan and Wei Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '08, pages 80–89, 2008.

[81] Wei Zhang and Jun Yan. Accurately Estimating Worst-Case Execution Time for Multi-core Processors with Shared Direct-Mapped Instruction Caches. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, pages 455–463, 2009.

[82] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In

Proceedings of 2009 IEEE 30th Real-Time Systems Symposium, RTSS '09, pages 57–67, 2009.

[83] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, 1995.

[84] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 4(3):257–279, July 1999.

[85] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and Precise WCET Prediction by Separated Cache andPath Analyses. *Real-Time Syst.*, 18(2/3):157–179, May 2000.

[86] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 339–349, 2010.

[87] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures Using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, WCET '10, pages 101–112, 2010.

[88] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Shared Data Caches Conflicts Reduction for WCET Computation in Multi-Core Architectures. In *18th International Conference on Real-Time and Network Systems*, page 2283, November 2010.

[89] Jan Nowotsch and Michael Paulitsch. Leveraging Multi-core Computing Architectures in Avionics. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, pages 132–143, 2012.

[90] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the Suitability of the NGMP Multi-core Processor in the Space Domain. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 175–184, 2012.

[91] Reinhard Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 309–322. Springer Berlin Heidelberg, 2004.

[92] Alexander Metzner. Why Model Checking Can Improve WCET Analysis. In Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347. Springer Berlin Heidelberg, 2004.

[93] Mingsong Lv, Zonghua Gu, Nan Guan, Qingxu Deng, and Ge Yu. Performance Comparison of Techniques on Static Path Analysis of WCET. In *Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing - Volume 01*, EUC '08, pages 104–111, 2008.

[94] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. *IEEE Trans. Comput.*, 47(6):700–713, June 1998.

[95] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Bounding Cache-Related Preemption Delay for Real-Time Systems. *IEEE Trans. Softw. Eng.*, 27(9):805–826, September 2001.

[96] Hiroyuki Tomiyama and Nikil D. Dutt. Program Path Analysis to Bound Cache-

related Preemption Delay in Preemptive Real-time Systems. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, CODES '00, pages 67–71, 2000.

[97] Hemendra Singh Negi, Tulika Mitra, and Abhik 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 '03, pages 201–206, 2003.

[98] Jan Staschulat and Rolf Ernst. Multiple Process Execution in Cache Related Preemption Delay Analysis. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 278–286, 2004.

[99] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pages 41–48, 2005.

[100] Harini Ramaprasad and Frank Mueller. Bounding Preemption Delay Within Data Cache Reference Patterns for Real-Time Tasks. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, pages 71–80, 2006.

[101] Harini Ramaprasad and Frank Mueller. Tightening the Bounds on Feasible Preemption Points. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 212–224, 2006.

[102] Yudong Tan and Vincent Mooney. Timing Analysis for Preemptive Multitasking Real-time Systems with Caches. *ACM Trans. Embed. Comput. Syst.*, 6(1), February 2007.

[103] Sebastian Altmeyer and Claire Burguiere. A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay. In *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, ECRTS '09, pages 109–118, 2009.

[104] Sudipta Chattopadhyay and Abhik Roychoudhury. Cache-Related Preemption Delay Analysis for Multilevel Noninclusive Caches. *ACM Trans. Embed. Comput. Syst.*, 13(5s):147:1–147:29, July 2014.

[105] Charlie Miller, Juan Caballero, Noah M Johnson, Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Crash analysis with BitBlaze. *at BlackHat USA*, 2010.

[106] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, 2007.

[107] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 15:1–15:16, 2007.

[108] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 2–16, 2006.

[109] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A Timing Analyzer for Embedded Software. *Sci. Comput. Program.*, 69(1-3):56–67, December 2007.

[110] Sudipta Chattopadhyay, Abhijeet Banerjee, and Abhik Roychoudhury. Precise Micro-architectural Modeling for WCET Analysis via AI+SAT. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '13, pages 87–96, 2013.

[111] Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A Unified WCET Analysis Framework for Multi-core Platforms. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS '12, pages 99–108, 2012.

[112] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, February 2002.

[113] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems*, SEUS'10, pages 35–46, 2010.

[114] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In Björn Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, jul 2010. OCG.

[115] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. PapaBench: a Free Real-Time Benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, 2006.

[116] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of PLDI '11*, pages 283–294, 2011.

[117] Christian Ferdinand. A fast and efficient cache persistence analysis. Technical report, Universitat des Saarlandes, 1997.

[118] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and DavidA. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer Berlin Heidelberg, 1996.

[119] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46, 1997.

[120] Clément Ballabriga and Hugues Casse. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 341–350, 2008.

[121] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 73–80, 1988.

[122] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-Aware Multicore WCET Analysis Through TDMA Offset Bounds. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ECRTS '11, pages 3–12, 2011.

[123] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related Preemption Delay via Useful Cache Blocks: Survey and Redefinition. *J. Syst. Archit.*, 57(7):707–719, August 2011.

[124] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience Analysis: Tightening the CRPD Bound for Set-associative Caches. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '10, pages 153–162, 2010.

[125] Henrik Theiling. *Control flow graphs for real-time systems analysis: reconstruction from binary executables and usage in ILP-based path analysis*. PhD thesis, Universitätsbibliothek, 2003.

[126] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001.

[127] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.