Precise analysis of Private and Shared Caches for tight WCET Estimates

A THESIS
SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy IN Computer Science

BY Kartik Nagar

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

May, 2016
Declaration of Originality

I, Kartik Nagar, with SR No. 04-04-10-12-12-1-09687 hereby declare that the material presented in the thesis titled

Precise Analysis of Private and Shared Caches for tight WCET Estimates

represents original work carried out by me in the Department of Computer Science and Automation at Indian Institute of Science during the years 2012-16.

With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Students Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Y N Srikant Advisor Signature
Acknowledgements

First of all, I would like to thank Prof. Y N Srikant for providing me the opportunity to perform this research, and also for keeping me on track for every step of the way. Thanks to the Department of CSA and IISc for providing an inspirational, productive environment. Many thanks to the CSA office staff for taking care of administrative matters quickly and efficiently. Thanks to MHRD and Microsoft Research India for funding this research through the Microsoft Research India PhD fellowship. A special thanks to all the members of the Compiler Lab for building an atmosphere of warmth and camaraderie, and also for all the technical discussions. Finally, thanks to my parents for their constant encouragement and support, without which this work would not have been possible.
Abstract

Worst Case Execution Time (WCET) is an important metric for programs running on real-time systems, and finding precise estimates of a program’s WCET is crucial to avoid over-allocation and wastage of hardware resources and to improve the schedulability of task sets. Hardware Caches have a major impact on a program’s execution time, and accurate estimation of a program’s cache behavior generally leads to significant reduction of its estimated WCET. However, the cache behavior of an access cannot be determined in isolation, since it depends on the access history, and in multi-path programs, the sequence of accesses made to the cache is not fixed. Hence, the same access can exhibit different cache behavior in different execution instances. This issue is further exacerbated in shared caches in a multi-core architecture, where interfering accesses from co-running programs on other cores can arrive at any time and modify the cache state. Further, cache analysis aimed towards WCET estimation should be provably safe, in that the estimated WCET should always exceed the actual execution time across all execution instances.

Faced with such contradicting requirements, previous approaches to cache analysis try to find memory accesses in a program which are guaranteed to hit the cache, irrespective of the program input, or the interferences from other co-running programs in case of a shared cache. To do so, they find the worst-case cache behavior for every individual memory access, analyzing the program (and interferences to a shared cache) to find whether there are execution instances where an access can suffer a cache miss. However, this approach loses out in making more precise predictions of private cache behavior which can be safely used for WCET estimation, and is significantly imprecise for shared cache analysis, where it is often impossible to guarantee that an access always hits the cache. In this work, we take a fundamentally different approach to cache analysis, by (1) trying to find worst-case behavior of groups of cache accesses, and (2) trying to find the exact cache behavior in the worst-case program execution instance, which is the execution instance with the maximum execution time.

For shared caches, we propose the Worst Case Interference Placement (WCIP) technique, which finds the worst-case timing of interfering accesses that would cause the maximum number
Abstract

of cache misses on the worst case execution path of the program. We first use Integer Linear Programming (ILP) to find an exact solution to the WCIP problem. However, this approach does not scale well for large programs, and so we investigate the WCIP problem in detail and prove that it is NP-Hard. In the process, we discover that the source of hardness of the WCIP problem lies in finding the worst case execution path which would exhibit the maximum execution time in the presence of interferences. We use this observation to propose an approximate algorithm for performing WCIP, which bypasses the hard problem of finding the worst case execution path by simply assuming that all cache accesses made by the program occur on a single path. This allows us to use a simple greedy algorithm to distribute the interfering accesses by choosing those cache accesses which could be most affected by interferences. The greedy algorithm also guarantees that the increase in WCET due to interferences is linear in the number of interferences. Experimentally, we show that WCIP provides substantial precision improvement in the final WCET over previous approaches to shared cache analysis, and the approximate algorithm almost matches the precision of the ILP-based approach, while being considerably faster.

For private caches, we discover multiple scenarios where hit-miss predictions made by traditional Abstract Interpretation-based approaches are not sufficient to fully capture cache behavior for WCET estimation. We introduce the concept of cache miss paths, which are abstractions of program path along which an access can suffer a cache miss. We propose an ILP-based approach which uses cache miss paths to find the exact cache behavior in the worst-case execution instance of the program. However, the ILP-based approach needs information about the worst-case execution path to predict the cache behavior, and hence it is difficult to integrate it with other micro-architectural analysis. We then show that most of the precision improvement of the ILP-based approach can be recovered without any knowledge of the worst-case execution path, by a careful analysis of the cache miss paths themselves. In particular, we can use cache miss paths to find the worst-case behavior of groups of cache accesses. Further, we can find upper bounds on the maximum number of times that cache accesses inside loops can exhibit worst-case behavior. This results in a scalable, precise method for performing private cache analysis which can be easily integrated with other micro-architectural analysis.
Publications based on this Thesis


2. **Path Sensitive Cache Analysis using Cache Miss Paths.** Kartik Nagar and Y.N. Srikant. 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), 2015.


Contents

Acknowledgements i
Abstract ii
Publications based on this Thesis iv
Contents v
List of Figures viii
List of Tables x

1 Introduction 1
   1.1 Shared Cache analysis ................................................. 5
   1.2 Private Cache Analysis ................................................ 9
   1.3 Organization of this thesis ............................................. 11

2 Preliminaries 12
   2.1 Abstract Interpretation .................................................. 12
   2.2 WCET estimation ......................................................... 14
   2.3 Cache Analysis .......................................................... 14
      2.3.1 Cache Architecture ................................................. 14
      2.3.2 Predicting cache behavior ....................................... 18

3 Shared Cache Analysis 25
   3.1 Setup ....................................................................... 26
   3.2 Literature Survey ......................................................... 27
   3.3 Example .................................................................. 29
   3.4 Cache hit paths .......................................................... 32
3.5 ILP-based WCIP ......................................................... 37
   3.5.1 Notation ......................................................... 37
   3.5.2 Objective Function .......................................... 37
   3.5.3 Access Constraints .......................................... 39
   3.5.4 Handling Loops ............................................. 42
   3.5.5 Interference Budget Constraints ......................... 43
   3.5.6 Handling Code/Data sharing ............................... 44
   3.5.7 Proof of Safety ............................................. 45
3.6 Complexity of WCIP .............................................. 46
3.7 Approximate WCIP ............................................... 56
   3.7.1 Setup ......................................................... 57
   3.7.2 Motivation ................................................... 57
   3.7.3 The overlapping effect .................................... 58
   3.7.4 Interference Distribution Algorithm .................... 62
   3.7.5 Algorithm analysis ....................................... 64
3.8 Interaction with the shared bus ................................ 65
3.9 Experimental Evaluation ........................................ 68
   3.9.1 Setup ......................................................... 68
   3.9.2 Results ....................................................... 69
   3.9.3 Hits on the WC path and the maximum OF ............... 71
   3.9.4 Changing the cache size ................................... 72
   3.9.5 Comparison with Simulated WCET ....................... 73
   3.9.6 Cache partitioning ....................................... 74
   3.9.7 Interaction with TDMA-based shared bus ............... 76

4 Private Cache Analysis ............................................. 78
   4.1 Literature Survey ............................................. 78
   4.2 Examples ....................................................... 81
   4.3 Cache miss paths ............................................. 85
   4.4 Algorithmic Approach ....................................... 91
      4.4.1 The precision issue with Must analysis .............. 91
      4.4.2 The precision issue with Persistence analysis ....... 92
      4.4.3 Finding maximum number of cache misses in a basic block .... 92
      4.4.4 Complexity ............................................. 96
      4.4.5 Finding worst-case profiles of basic blocks ........ 98
## CONTENTS

4.5 ILP-based Approach .................................................. 103
4.6 Experimental Evaluation ............................................. 106
  4.6.1 Setup .................................................................... 106
  4.6.2 Mälardalen Benchmarks .......................................... 109
  4.6.3 Benchmarks with Floating Point Operations ................. 112
  4.6.4 DEBIE Benchmarks ................................................ 113

5 Conclusion and Future Work ......................................... 117

Bibliography ................................................................. 120
List of Figures

1.1 Typical multi-core architecture ................................................. 5
2.1 General cache architecture ..................................................... 15
2.2 Example demonstrating LRU cache update ............................... 16
2.3 Example demonstrating join and transfer functions of Must analysis ........ 20
2.4 Example demonstrating join and transfer functions of May analysis ........ 21
2.5 Example demonstrating join and transfer functions of Persistence analysis .... 23
3.1 Example to show the imprecision of current approach. (a) Program fragment (b) Shared cache states (c) Shared cache states assuming 1 interfering access .......... 30
3.2 An instruction ........................................................................ 34
3.3 A Simple branched program ...................................................... 47
3.4 Program $P_{K_P}$ .................................................................... 53
3.5 Example to illustrate the overlapping effect ............................... 59
3.6 Example which demonstrates that the greedy method based on eviction distance is not optimal ......................................................... 60
3.7 Approximate WCIP .................................................................. 62
3.8 Precision improvement of WCET obtained using (1) ILP-based WCIP and (2) Approximate WCIP over Hardy et. al’s approach ............................... 70
3.9 Graph showing the relation between WCET obtained using WCIP and the amount of shared cache interference ............................................. 72
3.10 Precision improvement of the WCET for two different cache configurations (a) 2 KB L2 cache (b) 8 KB L2 cache ......................................................... 73
3.11 WCET overestimation ratio as compared with simulated WCET, for (1) private L2 cache, (2) shared L2 cache analyzed using ILP-based WCIP, (3) shared L2 cache analyzed using approximate WCIP and (4) shared L2 cache analyzed using Hardy et. al.’s approach ................................................. 74
LIST OF FIGURES

3.12 Graph showing the percentage increase in WCET obtained using (1) Approximate WCIP, (2) Horizontal cache partitioning and (3) assuming all shared cache accesses as misses ........................................ 75

4.1 Precision issue with Must Analysis .................................................. 81
4.2 Precision issue with Persistence Analysis ........................................... 82
4.3 Example illustrating difference between worst-case behavior of individual accesses and a group of accesses ........................................ 83
4.4 Example illustrating frequency of worst-case behavior for accesses in loops ................................. 84
4.5 Example illustrating impact of Worst case execution path on cache analysis ........................................ 84
4.6 Program $P_A$ ................................................................................. 97
4.7 Graph showing precision improvement in WCET of Mälardalen benchmarks ................................. 110
4.8 Graph showing ratio of WCET obtained using ILP/Algorithmic approach as compared with AI-based approach for benchmarks with floating point operations 113
4.9 Graph showing precision improvement in WCET of DEBIE-1 Tasks .................... 114
List of Tables

3.1 Notation used in ILP-based WCIP .................................................. 36

4.1 AI-based Miss Path Analysis applied to the example of Figure 4.1 .......... 89
4.2 Data flow analysis \( \mathcal{D}_v \) to determine conflict information for basic block \( v \) .... 94
4.3 Notation used in ILP-based private cache analysis .......................... 104
4.4 Mälardalen Benchmarks and their code sizes .................................. 109
4.5 Break-down of precision improvement ........................................... 111
4.6 DEBIE-1 Tasks  .............................................................................. 114
4.7 Break-down of precision improvement ........................................... 115
4.8 Analysis Time (in seconds) ........................................................... 116
Chapter 1

Introduction

Real time and embedded systems have to respond to physical events occurring in their environment in a prompt and deadline-constrained manner. Examples of real-time systems are software and control systems used in medical devices, automobiles, aviation, etc. In these systems, tasks are created in response to events occurring in the environment, and these tasks must finish their execution in a fixed, pre-defined interval of time (called deadlines) to ensure that the system continues to function correctly. A good example to understand the importance of time is the anti-lock braking systems used in modern cars. When brakes are applied, to ensure that there is no locking, the speeds of all the four tyres should decrease in a uniform manner. The anti-lock braking system monitors the speeds of the tyres, and determines the braking pressures required to ensure uniform speeds. For such a system to function correctly, it must respond immediately (within a fixed deadline) to any non-uniformity of speeds and determine the appropriate braking pressures, because otherwise, the changes in the speeds will ultimately lead to locking. Hence, providing timing guarantees for programs running on such a system is as important as proving correctness of the programs.

For safety-critical hard real-time systems (for example, medical devices or aviation systems), missing a deadline could be a matter of life and death. In general, a real-time system is modeled as a collection of tasks, where each task is associated with a deadline and worst case execution time (WCET). For a multi-tasking system, the real-time scheduling policy creates a schedule (either on-line or off-line) of tasks on the processor(s) to ensure that all deadlines of all tasks are met. A lot of work has been carried out in the area of real-time scheduling to explore various design alternatives (pre-emptive [44] vs non pre-emptive [43], global multi-processor scheduling [17] vs partitioned multi-processor scheduling [27], etc.), to ensure maximum processor utilization and to find schedulability tests [43, 37, 45, 16] to check whether deadlines of all tasks can be met, but all these works require one basic assumption: the actual execution time of a
task must never exceed its estimated WCET. As a result, timing analysis of programs, which is responsible for estimating a program’s WCET, occupies an important role in real-time and embedded systems.

All approaches to timing analysis can be broadly classified into two groups [63]: (1) Static analysis-based approaches [64], which generally use an abstraction of the program (for example its Control Flow Graph) and an abstraction of the architecture on which the program is to run, to derive theoretically safe upper bounds on the WCET, and (2) Measurement-based approaches [18, 70, 55], which typically execute (or simulate) the program (or parts of the program) on the architecture and use the maximal observed execution time to derive the WCET. An analogy from the field of software verification is that static timing analysis can be compared with program verification which can rigorously prove absence of bugs at the cost of false positives, while measurement based timing analysis can be compared with software testing, which may miss some bugs, but will only report actual bugs. Examples of tools based on static timing analysis are aiT [1] of AbsInt Angewandte Informatik, Germany, Bound-T [2] of Tidorum, Finland, Heptane [3] of IRISA, Rennes, Chronos [41] from National University of Singapore, SymTA/P [7] Tool of TU Braunschweig, Germany. Example of a tool based on measurement-based timing analysis is RapiTime [5] of Rapita Systems Ltd., UK. SWEET (SWEdish Execution Time tool) [6] uses a mixture of both static and measurement based approaches.

There are pros and cons to both approaches. The major advantage of static timing analysis over measurement-based analysis is that it can provide a formally provable guarantee that that actual execution time will never exceed the WCET, assuming that the architectural model that it uses is an abstraction of the actual architecture (i.e. all behaviors of the real architecture must be captured in the abstract model). On the other hand, measurement-based analysis can only provide probabilistic assurances based on number of test runs, input coverage, etc. For hard real-time systems, a formal guarantee on safety is a necessary requirement, but to ensure the safety of the WCET, static analysis-based techniques often forgo on another important requirement: precision. Since static analysis-based techniques use abstractions of the program and the architecture, they only determine an upper bound on the actual WCET, and the precision of this upper bound is defined by how close it is to the actual WCET of the program. On the other hand, measurement-based techniques approach the actual WCET from the other direction, i.e. they find the maximum observed execution time among a number of actual test runs, which is guaranteed to be less than or equal to the program’s actual WCET. Ideally, one would want a timing analysis technique which can match the precision of measurement-based approaches and provide safety assurance of static analysis based approaches. The work presented in this thesis is a step in that direction, by statically trying to precisely capture the
impact of an important architectural component (the hardware cache) on the execution time of programs.

It must be noted that there are certain scenarios where only one of the two approaches may be actually feasible. For example, there may not be enough details available about the architecture based on which an abstraction can be created and used for static timing analysis [8], in which case, one may be forced to use measurement-based approaches. Similarly, in multi-core architectures with shared resources, the chances of finding the worst-case execution instance (which causes the WCET) by repeated test runs would be very small, since it may require a very specific interleaving of shared resource access requests from different cores. In such cases, the actual execution times reported by measurement-based approaches may be nowhere close to the actual WCET, and a static search for the worst-case interleaving may become necessary.

In this work, we will only focus on static timing analysis. Conceptually, a program may run for different durations of time due to two main reasons: (1) Different program inputs, which may trigger different paths in the program (2) Different execution times of an individual program instruction, due to different hardware states. In the absence of the second reason, i.e. in a hypothetical scenario where execution time of every instruction is a constant, the exact WCET of a structured, reducible program \(^1\) can be efficiently determined with time complexity polynomial in the size of the program [56]. The problem of finding the WCET in this case reduces to finding the longest weighted path in an expanded control flow graph of the program, which encodes all the feasible paths, and where the weights of the nodes represent the execution times of individual instructions.

Unfortunately, modern architectures use various features such as hardware caches, pipelined execution, shared resources, etc. to speed up execution, and these features introduce variability in the execution time of a program instruction, so that the same instruction may take different number of processor cycles for completion in different execution instances. In the face of such variable execution time, static analysis-based approaches, in addition to finding the longest path in the program, also have to determine safe estimates of the execution time of individual program instructions. Picking the minimum among actual execution times of an instruction will provide maximum precision, but clearly jeopardizes safety, while picking the maximum execution time will guarantee safety, but will decrease the precision of the analysis.

The reason behind importance of finding precise WCET estimates is that hardware resources such as processor time are typically allocated to tasks by scheduling policies for the entire duration of their estimated WCETs. Hence, imprecise and bloated WCET can lead to severe

\(^1\)Assuming no recursion, all loops have statically known bounds, information about all feasible paths in the program is available
wastage of hardware resources. This could potentially nullify any advantage of using complex architectural features to improve the performance in the first place. In other words, to truly obtain any benefit from using complex architectural features in real-time systems, their impact on a program’s worst case execution time must be precisely modeled.

The hardware-controlled on-chip cache is one of the most important architectural components affecting a program’s execution time, since all of a program’s code and data must be fetched (often multiple times) from slow off-chip main memory. Caches can store a small portion of the main memory contents, and provide access speeds which are comparable to the processor speed, and which are orders of magnitude greater than the main memory speed. Every memory access made by a processor first goes to the cache, and if the accessed memory block is present in the cache, then it can be serviced 10 to 100 times faster than if the access has to go all the way to the off-chip main memory. As a result, accurate prediction of a program’s cache behavior can lead to substantial reduction in its estimated WCET. The aim of cache analysis is to statically find memory accesses made by a program which hit the cache, so that the cache latency can be used for such accesses while determining the program’s WCET.

To take advantage of temporal locality of memory accesses made by programs, cache contents are generally controlled by the hardware using the Least Recently Used (LRU) policy. According to this policy, all memory blocks are sorted according to the order of their last accesses, and only a fixed number of the most recently accessed blocks are allowed to be in the cache. It is easy to see that the cache behavior of an access thus depends on the access history, and in particular on the fact that whether the accessed block was also recently accessed in the past. The access history depends on the program path taken to reach a particular access, and hence in a multi-path program, the same access can have different cache behavior (i.e. hit or miss the cache) in different execution instances of the program. This raises the question of what cache behavior should be considered for such accesses while determining the WCET of the program.

With multi-cores being widely adopted in general-purpose computer systems, and the increasing demand of computational power in real-time systems, it is imperative that real-time systems also use multi-cores. However, accurate prediction of WCET in multi-core systems is much more difficult as compared to traditional single-core architectures, due to the presence of shared hardware resources such as the shared cache and the shared bus. In particular, due to the large difference between the shared cache latency and the main memory latency, predicting shared cache hits of a program is crucial for providing precise WCET estimates. The shared cache behavior of a program running on one core, though, is affected by the interfering accesses made by co-running programs on other cores. These interfering accesses can not only evict cache blocks of the program under analysis and thus cause extra shared cache misses, but their
Figure 1.1: Typical multi-core architecture

arrival timing also cannot be determined statically. Hence, it is not easy to provide accurate upper bounds on the damage caused by interfering shared cache accesses.

While predicting cache hits, one must ensure that the safety of the estimated WCET is maintained, i.e. the estimated WCET must be greater than the actual execution time of the program across all execution instances. As a result, the state-of-the-art approaches to private cache analysis [29, 36] and shared cache analysis [35, 67, 23] target memory accesses which are guaranteed to always hit the cache, across all execution instances. However, this approach cannot be used to predict certain frequently occurring cache behavior of private caches, and in fact can be significantly imprecise for shared caches.

1.1 Shared Cache analysis

Figure 3.1 shows the a simplified version of the typical architecture found in multi-core systems. Every core has a small, private L1 cache, and there is a comparatively larger L2 cache which is shared between all the cores. The access latency of the shared L2 cache is higher than the private L1 cache, whose access latency typically matches the processor frequency. All memory blocks requested by a core are first searched in its private L1 cache, and on a L1 cache miss, the
shared L2 cache is searched. If the requested memory block is also not present in the L2 cache, it is brought from the main memory (thus incurring the main memory latency) and written to both the L1 cache of the requesting core and the L2 cache. If there is no free space in the cache, then an existing cache block is evicted using the LRU policy.

Interfering memory accesses coming from other cores can significantly impact the shared cache behavior and the execution time of a program running on one of the cores. In our experiments, we have found that the execution time of programs running on one of the cores in a multi-core architectures can show increase of up to 3 times compared to the execution time of the same program running on single-core architecture with the same cache hierarchy. Others have reported maximum increase of 12X due to shared cache interference \(^1\). The worst-case increase in execution time would potentially be even higher, thus nullifying any advantage of using multi-cores. Uncontrolled sharing of the last-level shared cache can thus lead to large increase in both average-case and worst-case execution times \([9]\).

The primary hardware-based technique to avoid such performance loss is Cache Partitioning \([51]\), which divides the shared cache into private areas assigned to each core, in such a way that a core can only access the portion of shared cache assigned to it. This can achieved by two mechanisms: (1) Page Coloring \([69]\), where memory pages are assigned different colors based on some fixed portion of their addresses and pages with different colors map to different portions of the cache, and the required portion of a cache accessed by a task are pre-allocated to the task or (2) Way Partitioning where a subset of ways in all cache sets are pre-allocated to cores, so that a core can only evict cache blocks from its assigned subset of ways.

However, while unlimited sharing of cache space is one extreme, total isolation and complete partitioning of the shared cache into private areas is the opposite extreme. In particular, the on-chip shared cache space is very limited, and dividing it among the cores would further reduce the effective space available to each core. Moreover, some tasks may be more memory-intensive than others, and this raises the question of fair division of the cache space across cores. Further, a task may not require the same amount of cache space across its entire execution time, and assigning a fixed amount of cache space to a task for its entire execution lifetime could potentially result in wastage of cache space which would have been useful to a task running on another core.

Total partitioning could be just as bad as total sharing, and the best performance can be obtained by some form of limited sharing. A core can be assigned some amount of isolated cache space, but some portion of the cache could also be shared among cores. This would allow better utilization of the cache space, since a task can use the extra shared space in the cache

\(^1\)http://www.sei.cmu.edu/cyber-physical/research/timing-verification/multicore-scheduling.cfm
when necessary, and not all co-running tasks would likely require more cache space at the same time. From the point of view of WCET estimation, this mechanism can be used to control the maximum amount of cache interference that a task can suffer. However, for such a framework to be useful for WCET estimation, we also need a precise shared cache analysis technique which works well for limited amount of cache interference.

Since this would lead to smaller number of actual shared cache misses caused by interference during runtime, we should also be able to predict more shared cache hits statically. Unfortunately, the current state-of-the-art approach to shared cache analysis [35] does not have this property, and often fails to guarantee even a single cache hit even for very small amount of cache interference. This is because it tries to find the worst-case behavior of every cache access in the presence of interferences, separately for each individual access, which is the only way to find shared cache accesses which always hit the cache. For a shared cache with associativity $A$ (which is usually 4 or 8), an access can become a cache miss if the number of interfering accesses exceeds the associativity. Hence, no shared cache access can be guaranteed to hit the cache even for very small amount of interference (equal to the associativity).

The principal issue in trying to find the worst-case behavior of every individual access is that the effect of the same interfering accesses is considered multiple times, even though the interfering accesses themselves would only arrive once during actual execution. A more intuitive approach is to find the worst-case behavior of a group of accesses, by distributing the interfering accesses to cause the maximum number of misses. Even though this approach would not be able to predict which accesses are guaranteed to hit the cache, it will be able to predict how many accesses are guaranteed to hit the cache in the presence of interferences, which is enough to determine the program’s WCET.

An interference to the shared cache can cause any number of misses between 0 and the cache associativity, depending on the timing of its arrival. Hence, the same number of interferences can cause different number of cache misses for the program under analysis. The worst-case arrival of interferences is the arrival that causes the maximum number of shared cache misses, and hence the maximum increase in execution time. We propose the Worst Case Interference Placement (WCIP) approach, which tries to find the worst-case interference arrival along the worst-case execution path in the program.

WCIP is safe and is theoretically the most precise method to estimate the shared cache behavior, because one must consider the possibility of a program run, which will traverse the worst-case path and experience the worst-case arrival of interferences from other cores, and thus have an execution time equal to the WCET as calculated using WCIP\(^1\). In practice, the

\(^1\)Note that because of infeasible paths, imprecision of private cache analysis, etc., it is possible that the
WCETs obtained using WCIP are much lower than previous techniques used for shared cache analysis.

An important property of WCIP is that the number of estimated cache misses are directly proportional to the number of interferences. This is an important property, because for cache hits inside loops, it is possible that the interferences from other cores may not be enough to cause misses in all iterations, but may only cause misses in a subset of the iteration space. Previous approaches would not be able to handle such cases, but WCIP can identify them and accordingly find the maximum number of misses. As a result, WCIP can be used in conjunction with cache partitioning techniques to guarantee low increase in WCETs for low amount of shared cache interference.

**Our Contributions**: We perform WCIP by first characterizing each cache hit (guaranteed without interference) with **cache hit paths**, which are abstractions of program paths along which an interference can cause damage to the cache hit, and **eviction distance**, which is the minimum number of interferences required to cause a cache miss. We use cache hit paths, eviction distances, structural information about the program under analysis (in the form of its Control Flow Graph), number of interfering accesses and number of distinct interfering cache blocks accessed by co-running programs on other cores to form a **Integer Linear Program** (ILP) whose solution gives the maximum WCET of the program under analysis in the presence of interferences.

The ILP-based approach gives significantly more precise WCET estimates as compared to previous approaches, but it does not scale well for large programs. Hence, we investigate the WCIP problem in detail, and prove that performing WCIP is **NP-Hard** by reducing the 0-1 Knapsack problem. The difficulty in WCIP arises from the difference in the execution times and number of shared cache hits, along different program paths. A program path with high execution time may not have enough shared cache hits to ‘use’ all the interferences, while there may be program paths with large number of shared cache hits but lower execution times. To bypass this problem, we assume that all the shared cache hits are present on the worst-case path (calculated assuming no interferences). We then propose a **greedy algorithm** to perform approximate WCIP, which picks those cache hits in the program which are most likely to becomes misses due to interferences, since the cache blocks they access are already close to eviction without interferences. The total time complexity of this approach is linear in the program size. Experimental results show that for low amount of interference, both approximate and ILP-based WCIP give similar WCETs, with the former being significantly faster than the actual WCET of the program may be lower than the WCET obtained using WCIP. However, this issue is orthogonal to WCIP, which itself will not introduce any imprecision.
latter. Further, approximate WCIP can also be used to express the maximum increase in WCET due to interferences as a piece-wise linear function of the number of interferences.

### 1.2 Private Cache Analysis

The first-level cache, which is closest to the processor (and which is also the smallest and provides the lowest access latency) is almost always private. In a multi-core architecture with a cache hierarchy, each core gets its own first-level cache, and while the last level shared cache has a higher impact on the execution time (due to the larger gap between the last level cache latency and main memory latency), it is reasonable to expect that a fair percentage of the memory accesses of a program will actually be satisfied by the first-level cache. Just like with shared cache analysis, the goal of private cache analysis is to predict as many cache hits as possible, statically, which can be used later for Worst Case Execution Time estimation. Private caches do not suffer from the adverse effects of interfering accesses made by other cores, but the uncertainty in private cache analysis arises due to different cache behavior of an access along different paths in a program.

Abstract Interpretation (AI) based approaches such as Must, May [11] and Persistence analysis [36] are generally used to predict private cache behavior. However, to ensure that the predicted cache behavior can be safely used for WCET estimation, these approaches make a number of conservative choices, which has a negative effect on the precision of the predicted cache behavior and ultimately, the precision of the estimated WCET. In particular, Must analysis tries to find those cache accesses which will always hit the cache across all execution instances, while Persistence analysis tries to find cache blocks which are never evicted from the cache across all execution instances. To do so, these approaches try to find the worst-case behavior of individual accesses, i.e. Must analysis finds whether there exists an execution instance where an access might miss the cache, while persistence analysis tries to find an execution instance where a cache block may get evicted from the cache.

In order to improve the precision of private cache analysis, we use some of the same ideas used in our proposed approach for shared cache analysis. Instead of trying to find the worst-case behavior individually for every access, it may be more beneficial to find the worst-case behavior of a group of cache accesses. It is possible that the worst-case behavior of two memory accesses may never occur simultaneously in the same execution instance. This could happen, for example, when the accesses responsible for causing misses may never be executed together since they are in different branches of a conditional statement. In such cases, even though we do not know which access will cause a miss, we know that only one miss can occur among the two accesses. For memory accesses inside loops, it is possible that multiple iterations of
the enclosing loop are required to realize the worst-case behavior (i.e. to cause a cache miss). This could happen when the accesses responsible for causing misses cannot be executed in a single iteration of the enclosing loop. By detecting such cases, we can provide a more accurate bound on the cumulative number of misses caused by an access (or a group of accesses) inside a loop. Cache Hit-Miss Classifications such as Always-Hit or Persistent determined using Must and Persistence analysis are not enough to capture such cache behavior.

Further, there are precision issues with Must analysis and Persistence analysis themselves, as it is possible that they may fail to identify cache accesses which actually always hit the cache or are persistent. Finally, while it is safe to target the worst-case cache behavior, the actual aim of cache analysis is to find the cache behavior in the worst-case execution instance of the program, which is the execution instance with the maximum execution time. Information about the Worst Case Execution Path (WCEP) can be used to find this cache behavior, but it requires cache analysis and path analysis (used to find the WCEP) to be carried out in an integrated fashion. Finding the actual cache behavior along the WCEP is the most precise method for performing cache analysis.

**Our Contributions**: In order to solve these precision issues, we introduce the concept of **cache miss paths**, which are simply abstractions of program paths along which an access suffers a cache miss. The main insight behind the applicability of cache miss paths is that the cache behavior of an access mostly depends on the memory accesses made in the immediate vicinity of the access, and hence, in most cases, it is sufficient to analyze a small, fixed size neighborhood of an access with maximum precision to obtain complete information about the cache behavior of the access. We differentiate between every path in this small neighborhood, along which the access could miss the cache. Various analyses can then be performed on the cache miss paths of accesses to refine their cache behavior prediction.

For example, we analyze the program control flow graph (CFG) to find simple properties about the cache miss paths of different accesses which could either prevent them from occurring in the same execution instance or in the same iteration of an enclosing loop. For basic blocks inside loops, our approach finds **worst-case profiles** of the form $<\text{Max\_misses}, \text{Iters}>$, which denotes that the basic block can suffer $\text{Max\_misses}$ number of cache misses, only if it is executed once for every $\text{Iters}$ number of iterations of its innermost enclosing loop. We also show how to integrate the worst-case profiles in the Integer Linear Programming (ILP) based Implicit Path Enumeration Technique (IPET) to find the final WCET.

We demonstrate how miss paths can be used to easily solve the precision issues with Must and Persistence analysis, and thus propose the most precise approach for finding Always-Hit and Persistent accesses. Finally, we show how cache miss paths can directly integrated into the IPET.
approach to find the exact cache behavior in the worst-case execution instance, thus proposing
the most precise method for performing private cache analysis. Experimentally, we ran our
proposed approaches over a wide range of benchmarks and demonstrate substantial precision
improvement in the WCET over AI-based approaches. Further, we also show that in most cases,
the worst-case cache behavior actually matches the cache behavior in the worst-case execution
instance, and thus we can avoid the computationally expensive approach of integrating cache
miss paths into the IPET ILP, in favor of the comparatively faster algorithmic analysis of cache
miss paths. We also show how our approach can be integrated with an existing approach for
pipeline analysis, which allows us to utilize available instruction parallelism to further improve
the precision of the WCET.

1.3 Organization of this thesis

In chapter 2, we cover the basics of Abstract Interpretation and cache analysis, which form the
building blocks for our proposed approaches. In chapter 3, we present our proposed approaches
for precise analysis of shared caches in a multi-core architecture. After reviewing existing works
in the area, we demonstrate the imprecision of existing shared cache analysis approaches using
a simple example. We then define cache hit paths, which summarize the complete information
required to perform shared cache analysis. We then outline the main idea behind our approach
- called Worst Case Interference Placement - and present an ILP-based approach to perform
precise shared cache analysis. We analyze the complexity of WCIP, and use the resulting
observations to propose an efficient but approximate algorithmic approach to perform WCIP.
Finally, we present the results of our experimental evaluation, demonstrating the precision
improvement in the WCET obtained using WCIP as compared to previous approaches.

In chapter 4, we present our proposed approaches for precise analysis of private caches.
After a survey of existing works in this area, we demonstrate the precision issues in existing
approaches using a series of examples. We then define cache miss paths, which form the spine
of our proposed techniques. We then systematically show how cache miss paths can be used
to solve all the precision issues in existing approaches, proposing a series of algorithms in the
process. We also propose an ILP-based approach to integrate cache miss paths in the existing
IPET formulation to find WCET. Finally, we present the results of our experimental evaluation,
demonstrating the precision improvement in WCET obtained using our proposed approaches, as
compared to the WCET obtained using AI-based analysis. We also discuss about the pros and
cons of the algorithmic and ILP-based approaches, with the higher precision of the ILP-based
approach offset-ed by the ease of integrating the algorithmic approach with pipeline analysis.
We conclude and also present some avenues for future work in Chapter 5.
Chapter 2

Preliminaries

In this chapter, we explain the basic concepts related to program analysis and cache analysis, which form the building blocks for the proposed approaches in later chapters.

In our work, we will generally represent programs in the form of Control Flow Graphs (CFG). Each vertex in the CFG corresponds to a basic block. The edges of the CFG correspond to transitions across basic blocks. A basic block is a sequence of program instructions with a single entry and a single exit. All the instructions in a basic block are guaranteed to execute every time the first instruction of the basic block is executed. The standard procedure to divide a program into basic blocks is to identify jump and call instructions and their targets. The first instruction of a basic block is either the first instruction of a function, the target of a jump instruction, or the instruction immediately following a jump/call instruction. The last instruction of a basic block is either the last instruction of a function or a jump/call instruction. We assume function calls to be virtually inlined in the CFG. This means that every function call will be replaced by the entire CFG of the function itself. This helps in avoiding the use of imprecise inter-procedural program analysis techniques at the cost of increase in the size of the CFG.

2.1 Abstract Interpretation

To statically find program properties of interest which hold across all possible execution instances of a program, static program analysis techniques such as Abstract Interpretation [24] are used. This approach requires two ingredients: (1) The abstract lattice \((D, \preceq)\) and (2) Transfer functions \(D \to D\) associated with each basic block in the program. \(D\) is a finite set which contains all possible values for the program property of interest, along with a partial order \(\preceq\) under which any subset of \(D\) must have a least upper bound. The aim is to associate
an element of $D$ at the start of each basic block, which is the least upper bound of all values of $D$ possible at the start of the basic block during actual execution. Here, we formally define Abstract Interpretation in the backward direction, which we use in our work.

Let $G = (V, E)$ be the CFG of the program. For each basic block $v \in V$, we associate two elements $IN_v, OUT_v \in D$. $IN_v$ is associated with the end of $v$, while $OUT_v$ is associated with the start of $v$. Let $[v] : D \rightarrow D$ be the transfer function of basic block $v$. $[v]$ encodes the impact of the instructions in $v$ to the program property of interest (in reverse program order).

Let $v_{end}$ be the unique end basic block of the program (i.e. $\nexists v$ such that $(v, v_{end}) \in E$). If $\pi = v_1v_2\ldots v_k$ is a walk in $G$ (i.e. $(v_i, v_{i+1}) \in E$ for all $i$), then $[\pi] = [v_1] \circ [v_2] \circ \ldots \circ [v_k]$ is the cumulative transfer function of $\pi$ (in reverse order), where $\circ$ is the functional composition operator. For basic block $v$, let $\Sigma_v$ be the set of all walks from $v$ to $v_{end}$ in $G$. We assume that $IN_{v_{end}} = d_{in} \in D$. For $D' \subseteq D$, let $\bigcup D'$ be the least upper bound of the elements in $D'$ (obeying the partial order $\preceq$). Then, we require the following property:

$$OUT_v = \bigcup \{[\pi](d_{in}) : \pi \in \Sigma_v\} \quad (2.1)$$

**Algorithm 1: Algorithm to find fix-points**

1. for basic block $v \in V \setminus \{v_{end}\}$ do
2. \hspace{1em} $OUT_v \leftarrow \bot$
3. end for
4. $OUT_{v_{end}} = [v_{end}](d_{in})$
5. change_flag $\leftarrow 1$
6. while change_flag $= 1$ do
7. \hspace{1em} change_flag $\leftarrow 0$
8. \hspace{1em} for all basic blocks $v$ visited in reverse topological order do
9. \hspace{2em} $IN_v \leftarrow \bigcup \{OUT_w : (v, w) \in E\}$
10. \hspace{2em} if $OUT_v \neq [v](IN_v)$ then
11. \hspace{3em} change_flag $\leftarrow 1$
12. end if
13. $OUT_v \leftarrow [v](IN_v)$
14. end for
15. end while

Algorithm 1 shows the fix-point based approach used to determine the $OUT_v$ values. After initializing the $OUT_{v_{end}}$ to $[v_{end}](d_{in})$, and the $OUT_v$ value of all other basic blocks to $\bot$ (the smallest element in $D$), the algorithm repeatedly updates the $IN_v$ value of each basic block $v$ by taking the least upper bound of the $OUT_w$ values of its successors $w$, and then applies the transfer function $[v]$ on $IN_v$ to obtain $OUT_v$. The algorithm terminates when there is no
change in the $\text{OUT}_v$ values of all basic blocks.

Transfer function $f : D \rightarrow D$ is called distributive if $\forall D' \subseteq D$, $f(\bigcup D') = \bigcup f(D')$. The following result provides the condition under which Algorithm 1 terminates and gives the correct result.

**Result.** If the transfer functions $[v]$ for all $v \in V$ are distributive, then the $\text{OUT}_v$ values calculated using Algorithm 1 will satisfy Equation (2.1).

### 2.2 WCET estimation

After over a decade of research, a standard structure has emerged static analysis-based WCET estimation. First, the binary executable of the program is analyzed to reconstruct the Control Flow Graph. While it may be possible to begin from the source code of the program (written in a high-level language such as C), the binary executable is preferred, since this is version which will actually run on the architecture, it includes the effect of all the compiler optimizations which could affect the execution time, and it is also easier to obtain accurate information about the actual addresses of the memory accesses from the final executable.

In the next step, value analysis is carried out to determine bounds on the loops and infeasible paths. Value analysis is also used for address analysis, which determines the addresses of all possible memory accesses made by instructions in the program. The next step is micro-architectural analysis, which uses details about the architecture on which the program will run, to estimate the execution time (in cycles) of instructions in the program. The last step is path analysis, which uses the output of all the preceding steps to find the worst-case execution path, and thus the WCET of the program. For a detailed survey on WCET analysis tools and techniques, we refer to [63].

### 2.3 Cache Analysis

#### 2.3.1 Cache Architecture

**Cache Structure:** A hardware cache is characterized by following parameters: cache block size, capacity, cache associativity and cache replacement policy. In order to benefit from spatial locality, all transfer between cache and main memory happens in fixed-size chunks of memory called cache blocks, with the most commonly found sizes being 32 or 64 bytes. It is beneficial to view the complete memory accessed by a program as divided into contiguous chunks of cache blocks. As a result, if the cache block size is $\text{block}$, then a memory word at address $\text{addr}$ will be present in the cache block with address $\text{addr}/\text{block}$.

A cache of size $\text{capacity}$ and cache block size $\text{block}$ can hold $\text{capacity}/\text{block}$ number of cache
blocks. The contents in a cache are organized in terms of cache sets. If the cache associativity is $A$, then the number of cache sets would be $sets = capacity/(block \times A)$. A cache block with address $addr$ would map to cache set $addr\%sets$, where $\%$ is the modulo operator. Each cache set can store at most $A$ cache blocks mapped to it, and the contents of each cache set are handled independently, so that an access to cache block $m$ can only cause a change in the contents of the cache set to which $m$ is mapped.

Figure 2.1 shows a schematic diagram of a general cache architecture, with $n$ cache sets and associativity $A$. Each way in a cache set can store a single cache block, and hence the entire cache can store maximum of $nA$ cache blocks.

The total number of cache blocks accessed by a program and mapped to a cache set would generally be much larger than the cache associativity $A$, and hence, the cache replacement policy decides which $A$ cache blocks should be present in the cache set. Over the years, a number of deterministic replacement policies such as LRU, FIFO, PLRU, MRU, etc. and random replacement policies such as Evict-on-miss, Evict-on-access, etc. have been proposed. Different considerations such as access latency, power, area on chip, etc. play a role in selecting the replacement policy, but for real-time systems and WCET estimation, predictability is also an important concern. Out of all the replacement policies, LRU (Least Recently Used) is the most predictable. We say that a memory access hits the cache if the accessed cache block is present in the cache, otherwise we say it misses the cache. Given a sequence of memory accesses, the cache hit-miss behavior of an access in the sequence can be determined by examining the suffix of the sequence ending at the access. Among all the deterministic replacement policies, the length of the suffix required to predict cache hit-miss behavior is smallest when the cache
replacement policy is LRU [54]. For the random replacement policies, only a probabilistic hit-miss classification can be made, which is not useful for finding the WCET. In this work, we will assume that the cache replacement policy is LRU.

The LRU policy conceptually sorts all the cache blocks present in a cache set according to the order of their last accesses. The age of a cache block is defined to be its position in this sorted order. Given that the cache associativity is \( A \), we use the following terminology: the most recently accessed cache block has an age of 1, the second most recently accessed block has age 2 and so on. In Figure 2.1, the youngest cache block will be stored in way 1, while the oldest cache block will be stored in the maximum way number (way \( A \) if the cache set is full). If a new cache block is to be brought into a full cache set, then the cache block with age \( A \) is evicted from the cache.

Figure 2.2 demonstrates how the LRU replacement policy updates the cache on memory accesses. In the example, the cache has only two cache sets and has an associativity of 2. In the beginning, cache blocks \( m_1, m_2 \) are present in cache set 1, while \( m_3, m_4 \) are present in cache set 2. On an access to cache block \( m \), which maps to cache set 1, the cache block \( m_2 \) which is the oldest cache block in the full cache set 1 is evicted. The age of \( m_1 \) is increased by 1, and the newly accessed cache block \( m \) is brought into the cache set with age 1. Note that the cache set 2 is not affected by this access. Next, there is an access to cache block \( m_4 \), which maps to cache set 2 and is already present in the cache. Hence, no cache block is evicted from the cache, and the ages of \( m_3 \) and \( m_4 \) are updated. Again, note that the cache set 1 remains unchanged due to this access.

**Instruction and Data Cache:** All memory accesses made by a program can be divided into two classes: instruction fetches and data load/stores. Further, the complete memory accessed by a program can also be divided into two regions: code and data. The instruction fetches access the code region, while the data load/stores access the data region. The addresses of accesses corresponding to instruction fetches can be predicted statically with much higher accuracy as compared to addresses of accesses corresponding to data load/stores. This is because instruction fetches either follow the program order or jump instructions in the program, whose target addresses can be statically determined. Every instruction in the program can thus
be statically associated with a unique memory access in the code region. For RISC architectures, the size of an instruction is constant, and the cache block size is generally chosen as a multiple of the instruction size, so that every instruction maps to a unique cache block.

In contrast, the addresses of a data load/store cannot always be predicted with high accuracy. Due to addressing modes such as register indirect, the same instruction can access different addresses during different executions (for example an instruction inside a loop which loads different elements of an array in different iterations). Hence, a data load/store instruction will be statically associated with a set of addresses in the data region. For accesses to data allocated on the heap, even the exact set of addresses may be hard to determine statically.

In most modern architectures, separate caches are used to store cache blocks from code region and data region, respectively called instruction cache (I-cache) and data cache (D-cache). Every instruction fetch issued by the processor goes to the I-cache, while load and store requests go to the D-cache. As a result, the cache behaviors of the instruction and data accesses can be analyzed separately. In our work, we will mostly focus on predicting the instruction cache behavior.

**Multi-level caches:** Modern architectures generally have multiple caches at different levels. These levels are ordered, so that a memory address requested by the processor is first searched in the lowest cache level, and on cache miss, it goes to the next higher level, and so on. If the requested memory block is not present in any of the cache levels, only then the main memory is accessed. The rationale behind a multi-level cache hierarchy is that the gap between the processor speed and the main memory speed is generally too large and a single cache is too small to provide fast access to most of the memory requests. By having progressively larger caches (albeit with progressively increasing access latencies), a higher percentage of memory accesses can be serviced with reasonably small access latency.

In a non-inclusive cache hierarchy, eviction of cache blocks across different levels happen independently. If a cache level is accessed, then the requested cache block is always brought into the cache (if it was not originally present), possibly evicting another cache block from the same level. However, the evicted memory block could potentially be present in the lower levels. On the other hand, since cache levels are searched from lowest to highest levels until the accessed memory block is found, if a cache block is brought at a level, it will also be brought at all lower levels. In this work, we assume a non-inclusive cache hierarchy.

In a multi-core architecture, the highest level cache (sometimes called the last level cache) is generally shared among the cores, which means that it services memory accesses generated by all cores. Since we assume a non-inclusive cache hierarchy, evictions from the shared cache will not affect the private lower level cache behavior, and hence the private cache behaviors can
be predicted separately for each core.

**Timing Anomalies:** One of the most common assumptions made by majority of cache analysis techniques is the lack of timing anomalies. Intuitively, a timing anomaly happens when a local worst-case event is either not the worst-case event from a global perspective, or its impact on the global worst-case is larger. From the perspective of cache analysis, the local worst-case event for an access is a cache miss, but due to the interaction of the cache with the processor pipeline, a cache hit may actually cause a higher increase in the final execution time of the program, or the increase in the final execution time due to a cache miss may be higher than the cache miss latency.

In our work, we will generally assume that cache analysis can be carried out independent of the processor pipeline analysis, and that the additional cache latencies caused due to cache misses can be safely added to the final WCET. Hence, we will generally require the absence of strong impact timing anomalies \[64\]. In out-of-order processors with multiple functional units, the increase in execution time of an instruction due to cache miss can cause an even greater increase in the final execution time than the extra cache miss penalty. This happens because the increase in execution time of an instruction can delay the execution of a dependent instruction, due to which another instruction which is ready for execution may take hold of a functional unit, ultimately resulting in missed opportunities for instruction parallelism.

Performing an integrated analysis of the cache and processor pipeline (for example, as done in \[40\] or \[23\]) requires a safe hit-miss prediction for every cache access. Guaranteeing that individual accesses hit the cache is almost impossible for shared caches suffering even a small amount of interference from other cores. Hence, integrated pipeline and cache analysis is not feasible in the presence of shared caches in multi-cores. The property of being able to separately analyze the impact of different architectural components on the execution time of a program, and then to safely compose the results to obtain the final execution time is called timing compositionality \[33\]. It is expected that processors used in real-time and embedded systems should satisfy timing compositionality to allow precise analyses of the individual architectural components.

### 2.3.2 Predicting cache behavior

The aim of cache analysis is to statically find memory accesses which hit the cache, so that the cache latency can be used while determining the WCET. The approach takes as input the CFG of the program, and the information about the sequence of cache accesses made by each basic block in the program, in conjunction with the details about the cache architecture. There are three Abstract Interpretation-based approaches used to predict behavior of a single cache:
Must analysis, May analysis and Persistence Analysis.

Given a cache $C$, with its parameters capacity $C$, block $C$ and associativity $A$, and program $P$, let $M$ be the set of cache blocks accessed by $P$. The abstract lattice $D$ for Must and May analysis is the set of all functions $M \rightarrow \{1, 2, \ldots, A\} \cup \{\bot\}$. Each element of $D$ is a function which maps cache blocks which are present in the cache to their abstract age. If the cache block is not present in the cache, then it is mapped to $\bot$. The meaning of the abstract age is different for Must and May analyses. Note that all the three analyses must be performed in the forward direction, and hence a slightly modified version of Algorithm 1 (which works in the forward direction) is used.

**Must Analysis**: Must analysis computes those cache blocks which are guaranteed to be present in the cache across all execution instances, and also determines the maximum age of such cache blocks. Given $s_1, s_2 \in D$, the join $s_1 \sqcup_{\text{Must}} s_2$ is defined as follows:

$$(s_1 \sqcup_{\text{Must}} s_2)(m) = \begin{cases} 
\bot & \text{if } s_1(m) = \bot \lor s_2(m) = \bot \\
\text{MAX}(s_1(m), s_2(m)) & \text{otherwise}
\end{cases}$$

Only those cache blocks which are present in both $s_1$ and $s_2$ are present in the join $s_1 \sqcup_{\text{Must}} s_2$, with their ages being the maximum of the age in $s_1$ and $s_2$. The transfer function mimics the LRU cache update. If cache access $a$ accesses cache block $m$, then the transfer function $[a]_{\text{Must}}$ is given by $[a]_{\text{Must}}(s) = s'$, where

$$s'(m') = \begin{cases} 
1 & \text{if } m' = m \\
s(m') & \text{if } s(m) \neq \bot \land s(m') \neq \bot \land s(m') \geq s(m) \\
s(m') + 1 & \text{if } s(m') \neq \bot \land s(m') < A \land (s(m) = \bot \lor (s(m) \neq \bot \land s(m') < s(m))) \\
\bot & \text{otherwise}
\end{cases}$$

Note that $m$ and $m'$ map to the same cache set. For all other $m''$ mapping to a different cache set than $m$, $s'(m'') = s(m'')$.

Figure 2.3 demonstrates the join and transfer functions of must analysis using an example. Assume that the cache associativity is 4. At the top of the figure, abstract must cache states $s_1$ and $s_2$ are symbolically depicted. For example, $s_1$ is the abstract cache state with the following mapping: $m_1 \rightarrow 1, m_2 \rightarrow 2, m_3 \rightarrow 4, m_4 \rightarrow 4$ (all other cache blocks are mapped to $\bot$). After the join, $s_1 \sqcup_{\text{Must}} s_2$ does not contain $m_4$ or $m_5$ (i.e. they are mapped to $\bot$), because these
cache blocks are only present in either \(s_1\) or \(s_2\) but not in both. For the rest of the cache blocks \((m_1, m_2, m_3)\), the join maps them to maximum of their ages in \(s_1\) and \(s_2\). The figure also shows the effect of the transfer function \([m_1]\) on \(s_1 \sqcup_{\text{Must}} s_2\). \(m_1\) becomes the youngest cache block, while the age of cache block \(m_2\) is increased, since it originally had a smaller age than \(m_1\).

**May Analysis**: May analysis computes those cache blocks which may be present in the cache across all execution instances, and also determines the minimum ages of such cache blocks. Given \(s_1, s_2 \in D\), the join \(s_1 \sqcup_{\text{May}} s_2\) is defined as follows:

\[
(s_1 \sqcup_{\text{May}} s_2)(m) = \begin{cases} 
  s_1(m) & \text{if } s_1(m) \neq \bot \lor s_2(m) = \bot \\
  s_2(m) & \text{if } s_2(m) \neq \bot \lor s_1(m) = \bot \\
  \text{MIN}(s_1(m), s_2(m)) & \text{if } s_1(m) \neq \bot \land s_2(m) \neq \bot \\
  \bot & \text{otherwise}
\end{cases}
\]

If a cache block is present in either \(s_1\) or \(s_2\), then it is included in the join, with the minimum of its age in \(s_1\) and \(s_2\). The transfer function again mimics the LRU cache update. If cache access \(a\) accesses cache block \(m\), then the transfer function \([a]_{\text{May}}\) is given by \([a]_{\text{May}}(s) = s'\), where \(m\) and \(m'\) map to the same cache set and
Figure 2.4 demonstrates the join and transfer function of May analysis using a simple example. Again, assume that the cache associativity is 4. $s_1$ and $s_2$ are the same abstract cache states used in the previous example (Figure 2.3). However, here they denote the may abstract cache states. After join, $s_1 \uplus_{May} s_2$ contains all the cache blocks present in either $s_1$ or $s_2$. Moreover, each cache block is assigned an age which is the minimum of its ages in $s_1$ and $s_2$ (if it is present in both). The access to $m_1$ increases the ages of all cache blocks whose original age is less than or equal to $m_1$ (in this case only $m_3$).

**Persistence Analysis**: Persistence analysis computes the maximum ages of all cache blocks which may be present in the cache across all execution instances. It is generally used to identify those cache blocks which are never evicted during execution. It can be separately conducted at different static scopes (e.g. different loop levels), but for simplicity, here we present the analysis at the scope of the entire program. The following version of Persistence analysis is based on the safe approach proposed by [36]. The abstract lattice $D_{Pers}$ for Persistence analysis is

\[
s'(m') = \begin{cases} 
1 & \text{if } m' = m \\
 s(m') & \text{if } s(m) \neq \bot \land s(m') \neq \bot \land s(m') > s(m) \\
 s(m') + 1 & \text{if } s(m') \neq \bot \land s(m') < A \land (s(m) = \bot \lor (s(m) \neq \bot \land s(m') \leq s(m))) \\
 \bot & \text{otherwise}
\end{cases}
\]
\( M \rightarrow \mathbb{P}(M) \cup \{\bot\} \). Each element maps a cache block in persistence cache (non-\(\bot\) value) to the set of 'younger' cache blocks, i.e. cache blocks which are more recently accessed. Given \( s_1, s_2 \in D_{\text{Pers}} \), the join \( s_1 \sqcup_{\text{Pers}} s_2 \) is defined as follows:

\[
(s_1 \sqcup_{\text{Pers}} s_2)(m) = \begin{cases} 
  s_1(m) & \text{if } s_1(m) \neq \bot \lor s_2(m) = \bot \\
  s_2(m) & \text{if } s_2(m) \neq \bot \lor s_1(m) = \bot \\
  s_1(m) \cup s_2(m) & \text{if } s_1(m) \neq \bot \land s_2(m) \neq \bot \\
  \bot & \text{otherwise}
\end{cases}
\]

If a cache block is present in either \( s_1 \) or \( s_2 \), then it is included in the join, with union of the younger cache blocks in both \( s_1 \) and \( s_2 \). If cache access \( a \) accesses cache block \( m \), then the transfer function \([a]_{\text{Pers}}\) is given by \([a]_{\text{Pers}}(s) = s'\), where \( m \) and \( m' \) map to the same cache set and

\[
s'(m') = \begin{cases} 
  \phi & \text{if } m' = m \\
  s(m') \cup \{m\} & \text{if } s(m') \neq \bot \\
  \bot & \text{otherwise}
\end{cases}
\]

Figure 2.5 demonstrates the join and transfer functions of Persistence analysis. Assume that the cache associativity is 4. Abstract cache states \( s_1 \) and \( s_2 \) (similar to the cache states in previous examples) map each cache block to a set of cache blocks which may be younger (i.e. more recently accessed). After the join, each cache block is simply mapped to the union of its younger sets in both \( s_1 \) and \( s_2 \). The access to \( m_1 \) sets the younger set of \( m_1 \) to the null set, and adds \( m_1 \) to the younger set of every other cache block.

The abstract cache states (i.e. elements of the abstract lattice) obtained using Must, May and Persistence analysis are used to obtain a safe cache hit-miss classification (CHMC) of cache accesses. Consider cache access \( a \) which accesses cache block \( m \), and let \( s_{\text{Must}}, s_{\text{May}}, s_{\text{Pers}} \) be the abstract cache states obtained using Must, May and Persistence analysis respectively, just before the access \( a \). Then, if \( s_{\text{Must}}(m) \neq \bot \), then access \( a \) is classified as Always-Hit (AH), which means that the access is guaranteed to hit the cache across all execution instances. Otherwise, if \( s_{\text{May}}(m) = \bot \), then \( a \) is classified as Always-Miss (AM), which means that the access is guaranteed to always miss the cache across all execution instances. Otherwise, if \( s_{\text{Pers}}(m) \neq \bot \) and \(|s_{\text{Pers}}(m)| < A\), then \( a \) is classified as Persistent (PS), which means that the access can
Figure 2.5: Example demonstrating join and transfer functions of Persistence analysis
cause at most one cache miss. If none of the above conditions are satisfied, than the access remains non-classified (NC).

Note that all three analyses take as input the sequence of cache accesses made by each basic block in the program. While obtaining this sequence is straightforward for the first level L1 cache (procured directly from the results of address analysis), the cache accesses to higher level caches (in particular the shared cache) depend on the cache hit-miss behavior of the lower level caches. Due to the filtering effect, accesses which hit the lower level caches never reach the higher levels. Hence, for each cache level, every access made by the program is also associated with a cache access classification (CAC), which denotes whether the access may reach the level, and whether the access needs to be considered while performing analysis of the cache level. For a 2-level cache hierarchy, given an access \( a \), if the CHMC of \( a \) at L1 is AH, then its CAC at L2 will be Never. If the CHMC of \( a \) at L1 is AM, then its CAC at L2 would be Always. Finally, if the CHMC of \( a \) at L1 is NC or PS, then its CAC at L2 would be Uncertain. The cache analysis at L2 should consider all accesses whose CAC is Always or Uncertain. For a detailed description of CAC in case of a multi-level hierarchy with more than 2 levels, we refer to [34].

Shared cache analysis: The state-of-the-art approach to shared cache analysis [35] analyzes the shared cache separately from each core’s perspective. It first performs Must and May analysis of the private L1 caches of all cores. Then, Must analysis of the shared L2 cache is carried out, assuming no interfering shared cache accesses from other cores. After this step, the CAC at the shared cache level, of all accesses made by programs running on all cores is known, along with the CHMC at the shared cache level for the accesses made by the program under analysis. Due to our assumption that the cache hierarchy is non-inclusive, the CAC at the shared cache level is not affected by the interfering accesses (since these accesses can only cause evictions at the shared cache level, and the private L1 cache behavior is independent of the L2 cache behavior).

The approach then finds the maximal set of interfering cache blocks \( I \), by considering all cache accesses made by programs running on other cores, whose CAC at the shared cache level is Always or Uncertain. For the accesses, made by the program under analysis, whose CHMC is AH, the abstract age of the accessed cache block as determined using Must analysis (i.e. \( s_{Must}(m) \)) is used, and the approach checks if the sum of \(|I|\) and this abstract age is greater than or equal to the cache associativity. If yes, then the access is no longer guaranteed to hit the cache and hence is re-classified as NC.
Chapter 3

Shared Cache Analysis

In this chapter, we present our proposed approach for shared cache analysis. The purpose of shared cache analysis is to statically identify the memory accesses of a program which hit the shared cache. However, in a multi-core architecture, if we want to determine the shared cache behavior of a program running on one core, then we must consider the effect of the shared cache accesses generated by other cores (henceforth called interfering accesses or interferences). This is because such interferences can evict the cache blocks of the program under analysis, causing extra shared cache misses, which the program would not have suffered in isolation.

The shared cache misses caused due to interferences will cause an increase in the program’s execution time, which must be accounted for while determining its WCET. The WCET of a program in a multi-core environment should be greater than the actual execution time of the program across all its runs, i.e. irrespective of the program input, and the timing of the interferences from other cores. While it would be safe to assume that all shared cache accesses cause a miss, this would introduce significant imprecision in the WCET estimate. Our goal is to safely predict as many shared cache hits as possible, so that precise WCET estimates can be obtained. It should be clear that to perform shared cache analysis, knowledge about the shared accesses made by other cores, and hence knowledge about the programs running on other cores should be available.

The most general strategy for shared cache analysis is to analyze the shared cache behaviour separately from each core’s perspective. While performing the analysis from one core’s perspective, an abstraction of the shared cache accesses made by the other cores is used. This abstraction could either be just the number of accesses, or the number of distinct cache blocks accessed by other cores, etc. Previous approaches to shared cache analysis just used the latter abstraction, but the shared cache behaviour is also highly dependent on the former, and our approach uses both to perform precise shared cache analysis.
3.1 Setup

We assume a standard multi-core architecture, where each core has one (or more) private caches at lower levels and a shared cache (shared between all cores) at the highest level. We assume a timing anomaly-free architecture [46], since our approach is targeted towards maximizing the number of cache misses, and hence cache misses must cause the maximum increase in final execution time. The cache hierarchy is non-inclusive, so evictions at different levels are independent of each other, with the cache replacement policy being LRU at all levels. We limit our attention to instruction caches, and assume separate instruction caches at all levels. Even for private caches, data cache analysis is less precise than instruction cache analysis [52], due to the imprecision of address analysis which results in a set of accessed cache blocks for each instruction. While there have been some efforts at data cache analysis for shared caches [39], and it may be possible to use our approach for shared data cache analysis, it will require a non-trivial extension and is beyond the scope of this work.

Even though our approach can be applied to the shared cache at any level (with the restriction that all lower levels should be private), for simplicity, we will assume a 2-level cache hierarchy, with private L1 caches and a shared L2 cache. In the rest of the chapter, when we say a cache access, we mean an access to the L2 cache, and when we use cache hit, we mean a memory access which hits the L2 cache. We assume a shared bus interconnecting all the cores to the shared cache (and the main memory). We note that even though the shared bus can cause extra delays to any shared cache access, it will interfere neither with the internal workings of the cache nor with the memory accesses made to the shared cache.

Our technique requires knowledge about the maximum number of interferences that can be generated by all other cores, during a single execution instance of the program under analysis. For this, we need to know the mapping of tasks to cores, since the number of interferences generated by a task can be determined using analysis of the private caches. If programs are periodic, then the maximum number of instances of an interfering program, during a single instance of the program under analysis, may need to be determined.

We perform the standard Must and May analysis (as shown in Chapter 2) of the private L1 cache. As a first step, the shared cache is analyzed in isolation, assuming that no interferences arrive from other cores. As a result, for each instruction in the program, we have a Cache Access Classification (CAC) and Cache Hit Miss Classification (CHMC) at the shared cache level. The CAC determines whether the instruction will access the shared cache, and can be Always (A), Uncertain (U) or Never (N). In the first two cases, the access has to be considered at the shared cache, and in the last case, the access can be ignored. Note that interferences
from other cores will not affect the CAC.

The CHMC at the shared cache level can be Always-Hit (AH), Always-Miss (AM) or Uncertain (U) (note that for shared caches, we do not perform persistence analysis, and hence no access is classified as PS). The CHMC will be determined without considering interfering accesses from other cores. Accesses classified as AM or U will not be affected by interferences, since they are already counted as misses, and interfering accesses will only increase the number of misses. Hence, we will concentrate only on those accesses which are classified as AH at the shared level.

3.2 Literature Survey

Most of the variants of shared cache analysis in the literature are based on Hardy et. al.’s approach [35]. To find the effect of interfering accesses, this approach essentially uses the entire set of interfering cache blocks (i.e. cache blocks accessed by other cores) for each cache set to update the shared cache state at every program point. Since the timing of interfering accesses is not known, it is assumed that all interfering accesses can occur between any two accesses of the program under analysis. Hence the shared cache state obtained after the must analysis at each program point is updated by increasing the age of all cache blocks in the shared cache by the total number of interfering cache blocks. The updated shared cache states are then used to obtain the new (and safe) CHMC at the shared cache level. The obvious drawback of this approach is that effect of the same interfering accesses will be considered multiple times in the program, even though these accesses themselves will only happen once during actual program execution. We will demonstrate this point using a simple example in the next section.

Yan and Zhang [66] introduced the ‘always-except-one’ classification for instructions inside loops which access the shared cache. If there is only one interfering access and it is not inside any loop, then instructions inside a loop in the program under analysis will suffer maximum of one cache miss due to the interference and can be classified as ‘always-except-one’. Clearly, their approach is highly restrictive and its advantage will be implicitly covered by our approach. Moreover, their approach still considers the effect of the interfering access on all shared cache accesses of the program under analysis. In a later work [67], they take into account the sequence of interfering accesses and the sequence of accesses in the program under analysis to rule out certain misses arising due to infeasible interleaving. This is a complementary issue since we do not consider the sequence of interfering accesses in our approach. Moreover, the effect of the feasible interferences is still considered at all program points in [67].

[35] proposes an approach to find cache blocks which are only accessed once during the entire execution, and then uses this information in a hardware mechanism which forces accesses to
such static single usage cache blocks to bypass the shared cache. This could not only improve
the shared cache behaviour of the program itself (by decreasing the number of conflict and
capacity misses), but it would also reduce the shared cache interference to programs running on
other cores. However, not all accesses can be bypassed, and a precise analysis is still required
to consider the impact of the interfering accesses which are not bypassed.

Yan Li et al. [42] proposed a timing analysis for message-passing programs running concurrently on different cores. In this scenario, a partial order on the execution lifetimes of tasks can be determined based on the sent and received messages. Tasks which do not have overlapping execution lifetimes (because they are connected by some chain of sent and received messages) will not cause shared cache interference to each other. This information is used while performing shared cache analysis, but again, for tasks with overlapping lifetimes, they use Hardy et. al.’s approach.

The shared bus is another important hardware resource which introduces unpredictability in timing analysis, and a number of works ([38, 23, 22]) have looked at precise analysis of shared bus, and integrated analysis of shared bus and cache. However, the internal workings of the shared cache are not affected by the timing delays caused due to the shared bus, and hence they can be analyzed more or less independently. Moreover, we later also show how to integrate the results of our proposed approach for shared cache analysis into an existing shared bus analysis framework [38]. [21] uses Model checking to find out accesses which occur on infeasible paths in the interfering programs, and hence can be safely ignored while identifying interfering accesses. For interferences on feasible paths, Hardy et. al.’s approach is again used for shared cache analysis.

Hardware approaches ([60, 51, 68, 20]) focus on making the multi-core architecture prediction-friendly by using techniques such as cache locking and cache partitioning. Cache partitioning can be used to effectively allocate a private region of the shared cache to each core, which would reduce the shared cache analysis problem to be equivalent to analysis of private caches. Cache locking can be used to lock and prevent the eviction of selected cache blocks, so that interfering cache accesses will have no effect on these locked blocks, and accesses to the locked cache blocks can be safely predicted as Always-Hit. However, the on-chip shared cache space is limited, and cache partitioning/locking would further reduce the actual hardware-controlled cache space available to the cores.

The effectiveness of these approaches highly depend on size of the partitions allocated to the cores or the selection of locked cache contents. Moreover, they also require support from the hardware. These approaches also assume that the entire memory contents accessed by a task can be stored in the cache, but this assumption is unlikely to hold for large programs. The
cache requirements of a program change during its execution lifetime, and statically assigning fixed size partitions or locking cache blocks for the entire execution could have detrimental effects on the program’s actual execution time and its WCET, which could potentially be worse than the effect of interferences in a fully shared cache. In fact, our experiments with horizontal cache partitioning provide evidence of this fact. Finally, the schedulability analysis of multiple tasks on a multi-core architecture with cache partitioning also suffers, as the constraints on task period and execution time imposed by the schedulability test become more stringent [30], preventing many task sets to be scheduled.

On the other hand, cache locking or cache partitioning can be effectively used to control the amount of shared cache interference suffered by a task. In particular, all the applications of these hardware approaches so far have had to necessarily ensure that the amount of shared cache interference is zero, by allocating private partitions of the shared cache to each task. The reason for this extreme use of locking/partitioning is that previous shared cache analysis approaches gave highly pessimistic WCET estimates even when the cache interference was slightly more than zero. However, our approach guarantees a linear increase in the WCET with respect to the amount of interference, and hence a relaxed form of locking/partitioning which guarantees low, non-zero cache interference can be used to avoid wastage of cache space and infeasible requirements on the total size of memory contents accessed by a task.

We note that there have been recent attempts [53] to capture the WCET of a program in the form of an expression whose value depends on hardware parameters, which treats the WCET estimation process itself as a blackbox, and varies the hardware parameters to infer the relationship of the WCET with them. Our approach to shared cache analysis provides a similar relationship between the WCET of a program and the amount of shared cache interference suffered by it.

### 3.3 Example

In this section, we identify the precision issue with Hardy et al.’s approach to shared cache analysis [35] using a simple example. Consider the program fragment in Figure 3.1(a). a, b, c, d are cache blocks in the shared cache mapped to the same cache set, and also indicate the accesses to these cache blocks. Assume that the associativity of the shared cache is 2. Figure 3.1(b) shows the shared cache states obtained after must analysis, assuming no interferences (the position of a cache block indicates its age, with the age increasing from left to right). Using these cache states, the CHMC of the second access to a and the second access to c will be AH (since both a and c are present in the cache just before their accesses). Now, suppose the program fragment is running in a multi-core environment, and there is one interfering access
Figure 3.1: Example to show the imprecision of current approach. (a) Program fragment (b) Shared cache states (c) Shared cache states assuming 1 interfering access

coming from other cores. Figure 3.1(c) shows the updated cache states, obtained by increasing the age of each cache block in the original cache states by 1. This is the cache update made by [35]. Both the second accesses to \( a \) and \( c \) will now be classified as a miss, since they are not present in the updated cache state just before their accesses.

However, notice that if the interfering access comes at program points \( P \) or \( Q \), it will only affect the access to \( a \). Similarly, if it arrives at \( S \) or \( T \), it will only affect the access to \( c \), while if it arrives at \( R \), neither of the two accesses will be affected. Hence, one interfering access can cause at most one miss in the program, and one of the two accesses is guaranteed to remain a cache hit. Note that this example can be easily expanded to contain more accesses of the form \( m - m' - m \), so that the second access to \( m \) would be a cache hit. The approach of [35] will still report each such access to be a cache miss with just one interfering access, while during actual execution, at most one cache miss would be caused due to interferences. Similarly, if the entire program fragment is enclosed in a loop, then accesses in all iterations will reported as a miss, while during actual execution, the interfering access will come only during one iteration, causing at most one cache miss.

Hence, it is clear that assuming all interferences occur at all program points results in a highly pessimistic analysis and can blow up the WCET estimate. While it is true that the interfering accesses can arrive at any program point during actual execution, and thus no access
can be safely classified as AH, we are not really interested in which accesses are guaranteed to be hits. We are only interested in the maximum possible execution time, in the presence of interfering accesses, which can obtained by estimating the maximum number of misses caused by interferences. We have just seen that interfering accesses arriving at different program points can cause different number of cache misses, and the damage caused depends only on the sequence of accesses in the program being analyzed. If we can statically distribute the interfering accesses across the program such that they cause the maximum possible number of misses, then we can use the resultant shared cache states to obtain a safe CHMC. The worst-case arrival of interferences is the arrival that causes the maximum number of shared cache misses, and hence the maximum increase in execution time. The Worst Case Interference Placement (WCIP) technique tries to find the worst-case interference arrival along the worst-case path in the program.

WCIP is safe and is theoretically the most precise method to estimate the shared cache behavior, because one must consider the possibility of a program run, which will traverse the worst-case path and experience the worst-case arrival of interferences from other cores, and thus have an execution time equal to the WCET as calculated using WCIP\(^1\). Effectively, Hardy et. al.’s approach to shared cache analysis finds the local worst-case arrival of interferences for every individual cache hit, by simply assuming that all interferences occur before every cache hit. On the other hand, WCIP aims to find the global worst-case arrival for all the accesses on the worst-case path. An important advantage of this strategy is that the number of estimated cache misses becomes directly proportional to the number of interferences. This is an important property, because for cache hits inside loops, it is possible that the interferences from other cores may not be enough to cause misses in all iterations, but may only cause misses in a subset of the iteration space. Previous approaches would not be able to handle such cases, but WCIP can identify them and accordingly find the maximum number of misses.

We first propose an Integer Linear Programming (ILP) based approach to solve this optimization problem. ILP is already an integral part of the WCET estimation process, as most WCET analyzers use the IPET formulation of ILP \(^{[57]}\) to determine the worst case path in the program. ILP has also been used for cache hit-miss analysis, by constructing the Cache Conflict Graph (CCG) of programs \(^{[58], [59]}\). However, because of scalability issues and the success of the Abstract Interpretation based approaches, most WCET analyzers employ AI-based techniques for cache analysis. The IPET formulation requires the WCET of each instruction in

\(^{[57]}\)Note that because of infeasible paths, imprecision of private cache analysis, etc., it is possible that the actual WCET of the program may be lower than the WCET obtained using WCIP. However, this issue is orthogonal to WCIP, which itself will not introduce any imprecision.

31
the program as a constant. In our formulation, we relate the WCET of each instruction to the number of interfering accesses before the instruction.

For the example of Figure 3.1(a), we associate binary variables \( x_a \) and \( x_c \) with the second accesses to \( a \) and \( c \) respectively, to store their hit/miss status. Integer variables \( z_1, z_2, z_3, z_4, z_5 \) are associated with the program points \( P, Q, R, S, T \) respectively, and they store the number of interfering accesses occurring at those program points. Now, for the second access to \( a \) to become a miss, there needs to be at least one interfering access at \( P \) or \( Q \), and for the second access to \( c \) to become a miss, there needs to be at least one interfering access at \( S \) or \( T \). Consider the following ILP:

\[
\begin{align*}
\text{Maximize } & \quad x_a + x_c, \\
\text{subject to } & \quad x_a \leq z_1 + z_2 \\
& \quad x_c \leq z_4 + z_5 \\
& \quad z_1 + z_2 + z_3 + z_4 + z_5 \leq 1 
\end{align*}
\] (3.1) (3.2) (3.3)

The objective function maximizes the number of misses caused due to interferences. Equations 3.1 and 3.2 depict the access constraints, which state the minimum number of interferences to cause the access to be a miss. Equation 3.3 is the interference budget constraint, which encodes the maximum number of interferences available. Solving the above ILP will give the maximum value of the objective function to be 1, with either one of \( z_1, z_2, z_4, z_5 \) assigned as 1.

### 3.4 Cache hit paths

The concept of cache hit paths plays a central role in our proposed approach. Cache hit paths are abstractions of program paths along which an access is guaranteed to experience a cache hit. Only those interferences which occur on the cache hit path can cause an access to miss the cache. In this section, we give a formal definition of cache hit paths, and also present an Abstract Interpretation based approach to automatically find them.

Let \( G = (V, E) \) be the Control Flow Graph (CFG) of the program under analysis. \( V = \{b_1, b_2, \ldots, b_n\} \) are the basic blocks of the program, while \( E \subseteq V \times V \) denotes the control-flow among them. With each basic block \( b_i \), we associate the set of instructions \( Acc_{b_i} = \{a_{i1}, a_{i2}, \ldots, a_{il_i}\} \), whose CAC is Always or Uncertain at the shared cache level. Basic block \( i \) contains \( l_i \) such instructions. We ignore accesses which are satisfied by the private caches (i.e. accesses whose CAC is Never), since they will not be affected by the interferences from other
cores. Let $\text{Acc} = \bigcup_{i=1}^{n} \text{Acc}_{b_i}$ be the set of all shared cache accesses, and $\text{Acc}_H \subseteq \text{Acc}$ be the set of accesses whose CHMC is $\text{AH}$ (without interferences).

We define a program path $\pi$ to be a sequence of instructions (i.e. $\pi \in \text{Acc}^*$) of the program, which follow the program order. By program order, we mean that if $\pi$ contains the sub-sequence $a_1a_2$, then either $a_1$ and $a_2$ occur in the same basic block (in that order), or if they are in different basic blocks $b_{i1}$ and $b_{i2}$ respectively, then $(b_{i1}, b_{i2}) \in E$, $a_1$ is the last instruction in $b_{i1}$ and $a_2$ is the first instruction in $b_{i2}$. Given an instruction $a$ which accesses the cache block $m$ mapped to cache set $s$, a program path $\pi$ is called a cache hit path of $a$ if

1. $\pi$ begins with instruction $a'$ which also accesses $m$,
2. $\pi$ ends with instruction $a$ and has no other accesses to $m$ other than $a$ and $a'$, and
3. the number of distinct cache blocks (other than $m$) mapped to $s$ and accessed by instructions in $\pi$ is less than the cache associativity ($A$).

Note that $a$ and $a'$ could be the same instruction. Intuitively, if the actual execution of the program reaches instruction $a$ by flowing along a cache hit path of $a$, then the cache block $m$ (accessed by $a$) is guaranteed to be in the cache, and hence the access by $a$ will be a cache hit. This is because after the instruction $a'$, the cache block $m$ will be present in the cache and will be the most recently accessed block. Now, at least $A$ distinct cache blocks (mapped to set $s$) need to be brought into the cache to evict $m$. But since the number of distinct cache blocks accessed after $a'$ is strictly less than $A$, $m$ is guaranteed to escape eviction, and hence instruction $a$ will be a cache hit.

Conversely, if the instruction $a$ experiences a cache hit, then the execution must have passed through a cache hit path of $a$. Again, since $a$ is a cache hit, $m$ must be in the cache, just before the execution of $a$. Consider the last instruction in the execution flow, that brought $m$ to the cache. The path starting from this instruction and ending at $a$ would be a cache hit path.

Cache hit paths will be extensively used in the both the ILP-based approach and the approximate algorithmic approach for WCIP. Finding cache hit paths of accesses outside any loop is straightforward, as we can simply traverse the program in the reverse direction starting from the access, until we reach an access to the same cache block. However, for accesses inside loop, we may have to take the back edge and traverse the loop multiple times, and in general, it is not clear how many times one should do that, to ensure that all cache hit paths have been discovered. Hit paths for accesses inside loops can span multiple iterations of the loop.

We propose an Abstract Interpretation [24] based approach, which builds the hit paths (HP) of all shared cache hits in the program until a fix-point is reached. This program analysis is
carried out in the *backward* direction. Since we only need HPs of those accesses which are guaranteed shared cache hits (without interferences), we will only concentrate on the accesses whose CHMC is AH. The general idea is that we traverse backward in the CFG starting from the access and keep track of the cache blocks encountered along different paths. Eventually, another instruction accessing the same cache block will be encountered, and the hit paths can be completed. Since we only consider guaranteed cache hits, all the program paths leading to the access must end with cache hit paths of the access.

While the cache hit path was defined to be a sequence of accesses following program order, the sequence itself is not important when hit paths are used for WCIP. Hence, we will only maintain the set of accesses in every hit path. We also use a special symbol ⊣ to indicate that the hit path has been completed, i.e., the start instruction of the hit path has been encountered. Thus, for every hit path \( \pi \), \( \pi \in \mathcal{P}(\text{Acc} \cup \{\top\}) \).

A cache hit can have multiple hit paths, and hence we maintain a set of hit paths for each cache hit. Our abstract lattice is the set of all functions \( F = \{f \mid f : \text{Acc}_H \to \mathcal{P}(\mathcal{P}(\text{Acc} \cup \{\top\}))\} \). For \( f_1, f_2 \in F \), we say that \( f_1 \preceq f_2 \) iff \( \forall h \in \text{Acc}_H, f_1(h) \subseteq f_2(h) \). This is the standard power-set formulation of abstract lattice, with the join being defined as the point-wise union. Hence, \( (f_1 \sqcup f_2)(h) = f_1(h) \cup f_2(h) \).

We now define the transfer function for all instructions in \( \text{Acc} \). The transfer function for the rest of the instructions will be the identity function. Let \( cb(a) \) and \( cs(a) \) denote the cache block and the cache set accessed by \( a \), respectively. \( \text{CAC}(a) \) denotes the cache access classification of \( a \) at the shared cache level. As shown in Figure 3.2, suppose shared cache access \( a \) accesses cache block \( m \) mapped to cache set \( s \). For \( f_P \in F \), the transfer function \( T_{PQ} \) for this access is defined as \( T_{PQ}(f_P) = f_Q \), where
The transfer function operates separately on each hit path of every cache hit. The presence of a path \( \pi \) in \( f_P(h) \) indicates that \( \pi \) is a path from the access \( a \) to cache hit \( h \), and it is a subpath of some cache hit path of \( h \).

First, we consider the hit paths of the access \( a \) itself (if \( a \in \text{Acc}_H \)). We add the path \( \{a\} \), to begin the collection of hit paths of \( a \), while all existing paths of \( a \) are completed by adding \(-\). An existing path of \( a \) will be present when \( a \) is inside a loop, and it has already been encountered once during an earlier AI iteration.

In the second case, we consider the paths of those instructions \( h \) which access the same cache block \( cb(a) \). If the CAC of \( a \) is Always, then \( a \) is the instruction guaranteed to bring \( cb(a) \) into the cache, which will eventually be accessed by \( h \), causing \( h \) to be a cache hit. Hence, any existing paths of \( h \) which do not contain \(-\) are completed, while existing paths of \( h \) which have already been completed are retained.

In the third case, we consider the paths of those instructions \( h \) which access a different cache block \( cb(h) \), mapped to the same cache set \( cs(a) \). In this case, cache block \( cb(a) \) will conflict with \( cb(h) \) and therefore the access \( a \) is added to any existing path of \( h \), which has not been completed, while all completed hit paths are retained. Finally, paths of instructions which do not access the cache set \( cs(a) \) are not modified.

It is easy to see that the transfer function is distributive, since it operates individually on each hit path of every cache hit. It either adds a new path, or adds new accesses to an existing path, but this depends solely on the properties of the access or the path itself. Formally, if \( f_1 \preceq f_2 \), then \( f_1(h) \subseteq f_2(h) \). Since all hit paths in \( f_1(h) \) are also present in \( f_2(h) \), after applying
Symbol | Explanation
--- | ---
y_i | Integer variable storing the execution count of basic block \( b_i \)
x_{ij}^b | Integer variable storing the number of shared cache hits of instruction \( a_{ij} \)
x_{ij}^m | Integer variable storing the number of shared cache misses of instruction \( a_{ij} \)
x_{ij}^\pi | Integer variable storing the number of shared cache misses of instruction \( a_{ij} \) along path \( \pi \)
z_{ij} | Integer variable storing the total number of interfering accesses occurring just before instruction \( a_{ij} \)
w_{ij} | Integer variable storing the execution count of edge between basic blocks \( b_i \) and \( b_j \)
p_{ij}^\pi | Eviction distance of instruction \( a_{ij} \) along path \( \pi \)
c_i | Execution time of basic block \( b_i \) not affected by interferences
B_s | Number of interfering accesses mapped to cache set \( s \)
B_s^b | Number of interfering cache blocks mapped to set \( s \)

Table 3.1: Notation used in ILP-based WCIP
the transfer function, the transformed hit paths in $T_{PQ}(f_1)(h)$ will also be present in $T_{PQ}(f_2)(h)$. Moreover, the abstract lattice $F$ is finite, hence, the fixed-point based approach is guaranteed to terminate. All the completed cache hit paths of accesses in $H$ will be gathered at the start of the program in the fixpoint solution.

Theoretically, the AI-based approach has a time complexity exponential in the program size, as both the number and size of cache hit paths can be exponentially large. However, in practice, the fixpoint is quickly reached, and in general, takes the same amount of time as AI-based Must and May cache analysis.

### 3.5 ILP-based WCIP

#### 3.5.1 Notation

We now give a general description of our ILP for performing WCIP. We associate a constant $c_i$ with basic block $b_i \in V$, which is the total execution time of those instructions in $b_i$ which are not affected by interferences. This includes accesses which are satisfied by the private caches, non-memory-accessing instructions, etc. Let $cb_{ij}$ be the cache block in the shared cache accessed by instruction $a_{ij}$ in basic block $b_i$.

Let $A$ be the associativity of the shared cache. Let $M$ be the set of all cache blocks that can be stored in the shared cache. $cachestate_{ij} : M \rightarrow \{1, 2, \ldots, A, \bot\}$ is the abstract shared cache state as determined by the must analysis (ignoring the interfering accesses), just before instruction $j$ of basic block $i$. If $cachestate_{ij}(m) = h$ (where $h \neq \bot$), then $h$ is the age of $m$ just before $a_{ij}$. Note that in such a scenario, the instruction $a_{ij}$ will be a guaranteed shared cache hit (without interferences). Table 4.2 lists the variables and constants that will be used in our ILP formulation.

#### 3.5.2 Objective Function

Before specifying the objective function of our ILP formulation, we briefly explain the IPET formulation, which is used to find the worst case path in a program. Let $e_1, \ldots, e_n$ be the WCET of basic blocks $b_1, \ldots, b_n$. $y_1, \ldots, y_n$ are integer variables storing the execution count of basic blocks, and $w_{ij}$ stores the execution count of the edge between $b_i$ and $b_j$. The IPET
formulation is:

\[
\text{Maximize } \sum_{i=1}^{n} e_i y_i, \quad \text{subject to} \\
\forall i = 1, \ldots, n, \ y_i = \sum_{j \in \text{pred}(b_i)} w_{ji} = \sum_{k \in \text{succ}(b_i)} w_{ik} \\
\text{Loop Constraints} \ldots
\]

\(\text{pred}(b)\) and \(\text{succ}(b)\) give the predecessors and successors of basic block \(b\) respectively. The objective is to find the execution counts of basic blocks which maximizes the execution time of the program. The execution counts are constrained by the program structure, which basically places the restriction that the number of times execution enters a basic block must be the same as the number of times execution leaves the basic block, and this will also be the execution count of the basic block. The variable \(w_{ij}\) stores the number of times execution left basic block \(i\) and entered basic block \(j\).

The loop constraints give an upper bound on the execution count of the header basic block of each loop in the program, and are typically supplied by the programmer. The WCET of basic blocks in the above formulation are assumed to be constants and are obtained using micro-architecture analysis. After solving the ILP, the basic blocks whose execution counts are non-zero are considered to be on the worst-case path of the program, while the maximum value of the objective function will be the WCET of the program.

In our formulation, the WCET of basic blocks are not constants. The execution time of an instruction which accesses the shared cache depends on the interferences arriving from other cores. Our objective is to distribute the interferences across the program, such that they cause the maximum increase in the execution time of shared cache-accessing instructions, and all
interferences occur on the worst-case path. Consider the following ILP:

\[
\text{Maximize } \sum_{i=1}^{n} (c_i y_i + \sum_{j=1}^{l} (e^h x_{ij}^h + e^m x_{ij}^m))
\]

subject to

\[
\forall i = 1, \ldots, n, \quad y_i = \sum_{j \in \text{pred}(b_i)} w_{ji} = \sum_{k \in \text{succ}(b_i)} w_{ik}
\]

\[
\forall i = 1, \ldots, n, \quad \forall j = 1, \ldots, l_i, \quad x_{ij}^h + x_{ij}^m = y_i
\]

Loop Constraints ...
Access Constraints ...
Interference Budget Constraints ...

\(e^h\) and \(e^m\) are the execution time of an instruction, in the event of a shared cache hit and shared cache miss respectively. In the objective function, we have separated out the constant portion of the execution time of a basic block (i.e. \(c_i y_i\)), that is not affected by interferences. For each instruction accessing the shared cache, we have associated two variables \((x_{ij}^h\) and \(x_{ij}^m\)), which will store the hit and miss counts of the instruction in the shared cache. These variables will depend on the number of interferences affecting the instruction.

Then, we have the constraints on the execution count of basic blocks and the loop constraints, which are the same as those in the IPET formulation. Since \(e^m > e^h\), in the absence of any other constraints, to maximize the objective function, every instruction accessing the shared cache will be assumed to incur a miss. The access constraints will place an upper bound on the number of shared cache misses (i.e. \(x_{ij}^m\)) experienced by an instruction. The interference budget constraints will place an upper bound on the number of interferences for each cache set. The objective function will ensure that the interferences are distributed in such a manner that they cause the maximum number of shared cache misses. In the next few subsections, we give a detailed explanation of the access and interference budget constraints. First, we will define the concept of a cache hit path of an instruction, which will allow us to link the number of interferences with its hit/miss counts.

### 3.5.3 Access Constraints

We use cache hit paths to formulate the access constraints. Consider instruction \(a \in Acc_H\) in basic block \(b\), which accesses cache block \(m\) mapped to cache set \(s\), and let \(\pi\) be a cache hit path of this access. If the number of distinct cache blocks accessed on path \(\pi\) (excluding \(m\)) is \(h\), then the eviction distance of instruction \(a\) along \(\pi\) is defined to be \(A - h\). The eviction
distance gives minimum number of extra cache blocks required to be accessed on \( \pi \) to cause instruction \( a \) to be a cache miss. Thus, at least \( A - h \) interfering accesses, which are mapped to set \( s \) and coming from other cores, must arrive during the execution of the cache hit path \( \pi \) to cause instruction \( a \) to be a cache miss.

In the example in Figure 3.1, the cache hit path of the second access to \( a \) is \( a - b - a \), and its eviction distance is 1. Similarly, the cache hit path of the second access to \( c \) is \( c - d - c \), and its eviction distance is also 1. In general, let \( \pi = a' - a_1 - \ldots - a_k - a \), where \( a \) and \( a' \) access cache block \( m \) (which is mapped to cache set \( s \)). \( a_1, \ldots, a_k \) are the intervening instructions on the path, which access cache blocks mapped to set \( s \). Since \( \pi \) is a cache hit path, the number of distinct cache blocks accessed by the intervening instructions will be less than the cache associativity. Let \( p_\pi^a \) be the eviction distance. Let \( x_a \) be a binary variable, and let \( z_{a_1}, \ldots, z_{a_k}, z_a \) be integer variables storing the number of interfering accesses (mapped to \( s \)) occurring just before instructions \( a_1, \ldots, a_k, a \) respectively. Consider the following inequality:

\[
p_\pi^a x_a \leq z_{a_1} + z_{a_2} + \ldots + z_{a_k} + z_a
\]

First, note that \( z_{a_1}, \ldots, z_{a_k}, z_a \) capture all the interferences which may occur on the cache hit path \( \pi \). If an interfering access occurs on \( \pi \), it will happen after \( a' \) and before \( a \), and thus before any of the intervening accesses. Thus, the right hand side of the above equation captures the total number of interfering accesses that may occur on \( \pi \). If \( x_a \) is 1 in the above equation, then the total number of interfering accesses will be greater than or equal to the eviction distance. This will result in the access \( a \) becoming a cache miss. Thus we have encoded an access constraint which relates the integer variables storing the number of interferences with the binary variable which indicates a cache hit/miss due to interferences.

Interferences occurring before instruction \( a' \) will not have any effect on the cache hit-miss status of instruction \( a \), and hence they can be ignored in the above equation. Note that interferences mapped to set \( s \) will not affect the contents of other cache sets. In the above formulation, we have assumed that interfering accesses arrive just before instructions which access the set \( s \). This is safe because even if an interfering access arrives earlier, its effect will only be seen on the instructions accessing the set \( s \).

Also, for shared caches, the cache hit path of any instruction should begin with an instruction which is guaranteed to access the shared cache. In other words, the CAC of the start instruction of a cache hit path must be Always. The intervening instructions on the cache hit path (including the instruction \( a \)) can have a CAC of Always or Uncertain. If instruction \( a \) is inside a loop and has a CAC of Uncertain, we allow its cache hit path to start with \( a \).
If an instruction is classified as AH in the shared cache (assuming no interferences), then for all paths from the start of the program to the instruction, it will experience a cache hit. In other words, every path from the start of the program to the instruction should contain a cache hit path of the instruction. For the instruction to incur a cache miss due to interferences, the number of interfering accesses should exceed the eviction distance on at least one cache hit path (in fact, on the hit path which is part of the worst-case execution path).

Let instruction \( j \) of basic block \( i \) (denoted by \( a_{ij} \)) be a cache hit in the shared cache, and let \( \pi_1, \ldots, \pi_r \) be the cache hit paths of this instruction. For the moment, assume that \( a_{ij} \) is not inside any loop. Let the cache hit path \( \pi_l = a_{ij'} - a_{i1j1} - \ldots - a_{ikjk} - a_{ij} \). Let \( p_{ij}^{\pi_l} \) be the eviction distance along this path. Then, the access constraints for path \( \pi_l \) are:

\[
p_{ij}^{\pi_l} x_{ij}^{\pi_l} \leq z_{i1j1} + \ldots + z_{ikjk} + z_{ij} \tag{3.4}
\]

\[
x_{ij}^{\pi_l} \leq y_{ij'} \tag{3.5}
\]

\( x_{ij}^{\pi_l} \) stores the number of shared cache misses along the path \( \pi_l \). Equation 3.4 gives the relation between \( x_{ij}^{\pi_l} \) and the total number of interfering accesses on \( \pi_l \), as we saw in the last subsection. Equation 3.5 ensures that the instruction \( a_{ij'} \) occurs on the worst case path. If \( a_{ij'} \) is not on the worst case path, the cache hit path \( \pi_l \) itself is not on the worst case path, and hence no shared cache misses can be caused along the path. Remember that we also want to distribute the interferences only on the worst case path, and hence, we have to ensure that interferences not on the worst case path do not contribute in converting a cache hit to a cache miss. Hence, we add the following constraint for all \( a_{ij} \):

\[
z_{ij} \leq Ay_i \tag{3.6}
\]

If \( A \) interfering accesses arrive at a program point, then the entire shared cache will be emptied (from the perspective of the program under analysis). Hence, a valid upper bound for the number of interfering accesses at a program point is \( A \) multiplied by the number of times the program point is reached on the worst case path. The above constraint also ensures that if \( b_i \) is not on the worst case path, then no interferences will be assigned before instructions inside \( b_i \).

For each cache hit path \( \pi_l \), we thus obtain an upper bound on \( x_{ij}^{\pi_l} \). The access \( a_{ij} \) would become a cache miss if \( x_{ij}^{\pi_l} \) is non-zero on at least one cache hit path. In fact, \( x_{ij}^{\pi_l} \) will be non-zero for at most one cache hit path, since at most one cache hit path can be on the worst case path, and we have ensured that all interferences are to be distributed on the worst case
path only. The following constraint imposes the upper bound on $x^m_{ij}$:

$$x^m_{ij} \leq \sum_{l=1}^{r} x^{\pi_l}_{ij}$$

(3.7)

The above constraints are added for instructions which were classified as AH, assuming no interferences. For instruction $a_{ij}$ which accesses the shared cache and is classified as AM or U, we add the following constraint:

$$x^m_{ij} = y_i$$

(3.8)

### 3.5.4 Handling Loops

If an instruction inside a loop is classified as AH, then in every iteration of the loop, the instruction experiences a cache hit. Whenever it experiences a cache hit, execution must have passed through a cache hit path. The cache hit path for the first iteration may begin outside the loop, while the cache hit path for the rest of the iterations will begin within the loop itself.

The cache hit path inside a loop may be traversed multiple times (e.g. in multiple iterations), but each time, the number of interferences should exceed the eviction distance of the cache hit path to cause a shared cache miss. This is because in each iteration, the start instruction of the cache hit path will bring the cache block to the shared cache and make it the most recently accessed block. Hence, in every iteration, the age of the accessed cache block will be reset to 1 as the execution enters the cache hit path, and so every iteration requires interferences to cause a cache miss. In other words, if $p^\pi_a$ is the eviction distance of instruction $a$ along path $\pi$, and if $\pi$ is inside a loop with $T$ iterations, at least $Tp^\pi_a$ interferences are required to arrive on the path $\pi$ to cause $a$ to be a cache miss on all iterations.

Equivalently, if a total of $Z$ interferences are supposed to arrive on the path $\pi$, which is inside a loop, then these $Z$ interferences should be distributed across iterations such that exactly $p^\pi_a$ interferences will arrive each iteration, so that the number of cache misses incurred by $a$ due to interferences will be $\lceil Z/p^\pi_a \rceil$. This is the maximal number of misses that $Z$ interferences on $\pi$ can cause for instruction $a$.

Let us now look at the access constraints defined in the previous subsection from the perspective of instructions inside loops. If $a_{ij}$ is not inside a loop, then $x^m_{ij}$ will be a binary variable, which will be 1 if the instruction experiences a cache miss due to interferences. This is ensured by the access constraints added for each cache hit path. On the other hand, if the instruction is inside a loop, then $x^{\pi_l}_{ij}$ will give the maximal number of misses that can be caused by the assigned interferences along the cache hit path $\pi_l$. This is because, according to Equation 3.4,
\[ x_{ij}^\pi = \left\lfloor \frac{(z_{i_1j_1} + \ldots + z_{i_kj_k} + z_{ij})}{p_{ij}^\pi} \right\rfloor = \left\lfloor \frac{Z}{p_{ij}^\pi} \right\rfloor, \]

where \( Z \) would be the total number of interferences on the cache hit path. We just argued earlier that this is the maximum number of misses that \( Z \) interferences along a cache hit path can cause for an instruction inside a loop.

If an instruction is not inside any loop, then at most one cache hit path of the instruction will be on the worst case path. However, consider an instruction inside any arbitrarily nested loop. For each parent loop of the instruction, a cache hit path originating from the parent loop could be on the worst case path. Hence, \( x_{ij}^\pi \) may be non-zero for multiple cache hit paths \( \pi_i \). Equation 3.7 ensures that the contribution of each cache hit path will be considered while finding the total number of misses caused due to interferences. Also, the maximum number of misses caused by a cache hit path would be upper bounded by the number of times the first instruction of the cache hit path is executed (i.e. the number of times the cache hit path itself is executed). This is ensured by Equation 3.5.

We note that this method of counting interferences occurring inside loops introduces slight imprecision in the analysis. For program points inside a loop, we only keep count of the total number of interferences arriving at the program point, but the distribution of interferences across iterations is not part of the ILP. We assume that if the interferences cause misses to some instruction, then they will arrive optimally w.r.t. that instruction, so that the maximum number of misses are caused. However, the optimal distribution may be different for different instructions.

For example, suppose a total of 6 interferences occur at some program point, which is on the cache hit paths of two instructions \( a_1 \) and \( a_2 \). Suppose the eviction distance of \( a_1 \) is 2, while the eviction distance of \( a_2 \) is 3. Both \( a_1 \) and \( a_2 \) are inside the same loop. Now, the optimal distribution for \( a_1 \) would be 2 interference per iteration, which will result in 3 misses for \( a_1 \). The optimal distribution for \( a_2 \) would be 3 interference per iteration, which will result in 2 misses for \( a_2 \). Hence, our ILP will count a total of 5 misses for \( a_1 \) and \( a_2 \), but this will never occur during actual execution. The feasible optimal distribution is 3 interferences for 2 iterations, resulting in a total for 4 misses (2 for \( a_1 \) and \( a_2 \)). Experimentally, we found substantial precision improvement with our approach, which indicates that the above problem may not have a significant impact on precision.

### 3.5.5 Interference Budget Constraints

Since we know the tasks running on all the cores, we can determine the number of interferences that are generated by each core. Each instruction in the program whose CAC is Always or Uncertain at the shared cache level is considered to make an interfering access to the shared cache. If such an instruction is inside a loop, then the number of interfering accesses will be
the iteration count of the loop. For our program under analysis, let $B_s$ be the total number of interferences mapped to cache set $s$. $B_s$ would be the sum of interferences generated by every other co-running task. Let $B_{cb}^s$ be the number of distinct interfering cache blocks mapped to set $s$. Let $CCB_s$ be the set of interfering cache blocks mapped to set $s$.

Let $Acc_s \subseteq Acc$ be the set of instructions in the program whose CAC is Always or Uncertain, and which access cache blocks mapped to set $s$. For every cache set $s$, the interference budget constraint is:

$$\sum_{a_{ij} \in INST_s} z_{ij} \leq B_s \quad (3.9)$$

Since $z_{ij}$ will be non-zero only for instructions which are on the worst case path (ensured by Equation 3.6), the above constraint ensures that all $B_s$ interferences will be distributed across instructions in $Acc_s$ which are on the worst case path.

In general, the number of interfering accesses, $B_s$ could be much larger than the number of interfering cache blocks, $B_{cb}^s$. While distributing interferences across the entire program, we have to use $B_s$, but while distributing accesses on a single cache hit path, we can use $B_{cb}^s$. An instruction will suffer a cache miss due to interferences on a cache hit path only if both the number of interfering accesses and the number of interfering cache blocks exceeds the eviction distance of the cache hit path. Hence, we only add the access constraint for the cache hit path $\pi$ of an instruction $a_{ij}$, if $B_{cb}^s$ is greater than or equal to the eviction distance $p_{ij}^\pi$. If $B_{cb}^s < p_{ij}^\pi$ for all cache hit paths $\pi$ of the instruction, no access constraint will be added for any cache hit path, and instead we will add the constraint $x_{ij}^h = y_i$.

### 3.5.6 Handling Code/Data sharing

So far, we have assumed that interfering cache blocks will be different from the cache blocks accessed by the program under analysis, so that every interfering access will increase the age of all cache blocks in the shared cache. However, if programs running on different cores use shared libraries, it is possible that same cache blocks may be accessed by multiple cores. A similar scenario would occur for data caches, if programs on different cores share data variables.

We can simply ignore the sharing of code/data during our analysis, since this only affects the precision of the analysis. In the presence of sharing, interfering cache blocks may already be present in the cache, in which case they will not increase the ages of older cache blocks. Consider instruction $a_{ij}$, which accesses cache block $m$ mapped to cache set $s$, and let $\pi$ be a cache hit path of the instruction. Let $M^\pi$ be the set of cache blocks mapped to set $s$, and accessed by the instructions in path $\pi$. Interfering accesses which access cache blocks in $CCB_s \cap M^\pi$ will not contribute in increasing the age of $m$, since these cache blocks will also be
accessed by instructions in the cache hit path. Hence, we calculate the set $CCB_s \setminus M^{\pi}$, and if $|CCB_s \setminus M^{\pi}| \geq p_i^{\pi}$, only then we add the access constraint for $\pi$. Again, if $|CCB_s \setminus M^{\pi}| < p_i^{\pi}$ for all cache hit paths $\pi$ of instruction $a_{ij}$, then we conclude that this access can never cause a miss due to interferences, and hence add the constraint $x_{ij}^{h} = y_i$.

### 3.5.7 Proof of Safety

Given an interference Budget $\{B_s\}_{s \in S}$ for each cache set and program $P$ running on core $c$, we would like to show that the WCET obtained using our ILP would be greater than the execution time of any actual execution instance of $P$ in the presence of interferences from the budget. The objective function of the ILP finds a program path such that the execution time of the program along the selected path would be maximum after assigning interferences on the path which generate the most number of misses. The ILP tries to maximize the execution time subject to the access constraints, and the access constraints give an upper bound on the number of shared cache misses due to interferences. If every feasible shared cache miss due to interferences is also allowed by the access constraints, this would mean that the ILP will maximize execution time taking into account every feasible shared cache miss. Hence, we will show that if an arrival of interferences can cause a shared cache miss during actual execution, then the same assignment of interferences in our ILP will also result in a shared cache miss.

An instruction can suffer a feasible shared cache miss only if it experiences a shared cache hit without interferences. Hence, the CHMC of the instruction without interferences should be U or AH. First, consider an instruction whose CHMC is U and which experiences a shared cache miss due to interferences during actual execution. Such instructions are already considered as shared cache misses in our ILP (Equation 3.8).

Now, consider an instruction $a$ whose CHMC is AH. Suppose $a$ experiences a shared cache miss due to interferences along the cache hit path $\pi$ during actual execution. Assume that $\pi$ begins at instruction $a'$. Suppose the interferences arrive before instructions $a_1, \ldots, a_k$ on the path. If the CAC of instruction $a'$ is A, then there will be an access constraint for $\pi$ in our ILP. The CAC of instructions $a_1, \ldots, a_k$ must be either A or U, so the interference variables $z_{a_1}, \ldots, z_{a_k}$ will be part of the access constraint. If the sum of interferences exceeds the eviction distance during actual execution, then the same assignment of interferences to $z_{a_1}, \ldots, z_{a_k}$ will also exceed the eviction distance in the access constraint. Thus, the shared cache miss will be allowed and will be taken into consideration by the ILP.

If the CAC of $a'$ is U, then there will be cache hit path $\pi'$ such that $\pi$ is a sub-path of $\pi'$. This is because $a$ is classified as AH, and hence there must be some instruction $a''$ before $a$ which accesses the cache block accessed by $a$ and has a CAC of A. The cache hit path starting
from $a''$ (call it $\pi'$) will contain the accesses $a_1, \ldots, a_k$. Hence, $z_{a_1}, \ldots, z_{a_k}$ will be part of the access constraint for $\pi'$, while $p_{\pi'} \leq p_{\pi}$, which will allow the shared cache miss for the same assignment of interferences.

### 3.6 Complexity of WCIP

The ILP-based approach gives us a way to perform WCIP but solving an ILP is an NP-Hard problem, and this is reflected in the fact that ILP-based WCIP fails to produce WCETs for some programs in a reasonable duration (as demonstrated later by experimental results). This raises the challenge of finding efficient techniques to perform WCIP. In this section, we will show that performing WCIP is a NP-Hard problem. Specifically, we will show that finding the worst-case path in a program, in the presence of interferences to the shared cache, is NP-Hard by reducing the 0-1 Knapsack problem.

Given the WCET and shared cache behavior of the program in isolation, as well as the information about interferences to the shared cache generated by other cores, the WCIP problem is to find the maximum increase in the WCET due to the shared cache misses caused by interferences. This can be decomposed into two inter-dependent problems: (1) find the program path with the maximum execution time in the presence of interferences and (2) find a distribution of interferences on this program path which causes the maximum number of shared cache misses. A distribution of interferences will assign disjoint subsets of interferences at each program point in the path. It is only essential to keep track of the number of interferences assigned at a program point.

To find the worst-case path in the presence of interferences, we must know the worst-case distribution which will cause the maximum increase in the execution time of the path. However, to find this worst-case distribution, we have to know the entire path along which the interferences are to be distributed. A naïve approach to WCIP would be take every complete program path from the beginning of the program to its end, find the worst-case distribution and hence the WCET of the path in the presence of interferences, and then select the path with the maximum WCET.

Finding the worst-case path in a program without any shared cache interferences can be done efficiently. The cache behaviour can be estimated without interferences using the AI-based techniques, which would lead to an estimate of the WCET of each basic block in the program. Given the WCET of each basic block, and the program CFG, there are techniques which can find the WCET of the program in time polynomial in the size of the CFG, by using a scoped version of Dijsktra’s algorithm to find longest path in directed acyclic graphs [12]. Interferences will cause shared cache misses and increase the WCET of basic blocks, and we will show that
finding the worst-case path in the presence of interferences becomes NP-Hard.

To focus on the problem of finding the worst-case path, we will make the worst-case distribution problem simpler by assuming a direct-mapped (DM) shared cache with a single cache set. DM caches contain a single cache block per cache set, and since we are assuming a single cache set, our entire cache will contain only one cache block, which will be the most recently accessed block. A cache hit will occur when the block present in the cache is accessed by the program. An interference from another core could evict the block in the cache and thus cause a cache miss if there is an access to the evicted cache block, after the interference.

We define a straight-line program to be one without any loops or branches. For our purposes, a straight-line program simply consists of a sequence of accesses to the shared cache. For this simplified shared cache architecture, WCIP in a straight-line program becomes trivial.

**Lemma 1.** Given a straight-line program with \( H \) number of shared cache hits and \( B \) number of interferences coming from other cores, and assuming a direct-mapped shared cache with one cache set, the maximum number of shared cache misses caused due to interferences would be \( \min(H, B) \).

Since at most one cache block will be present in the DM cache at a time, an interference can only affect the next access to this cache block. Hence, \( B \) interferences can cause at most \( B \) cache misses. If \( H \) is the number of shared cache hits in the program, and if \( H \leq B \), then every cache hit will become a miss by assigning one interference before the access. On the other hand if \( H > B \), then we can select any \( B \) cache hits and assign one interference before each selected cache hit, causing a total of \( B \) misses.

We now add one layer of complexity, and consider programs with single level of branching and no loops. An example of such a program is given in Figure 3.3. The program has \( n \) segments, where each segment is an if-then-else branch. For our purposes, each branch of a segment is just a sequence of shared cache accesses. \( t_i^l \) (\( t_i^r \)) is the execution time, without interferences, of the left (right) branch of the \( i \)th segment. \( h_i^l \) (\( h_i^r \)) is the number of shared cache hits in the left (right) branch of the \( i \)th segment.

These are the accesses which are guaranteed to hit the shared cache, without any interferences. Assume WLOG that \( \forall i, t_i^l \geq t_i^r \). Hence, the WCET without interferences would be \( \sum_{i=1}^{n} t_i^l \), obtained by taking the left branch of each segment.

For simplicity, assume a shared cache miss penalty of 1 cycle. If \( B \) interferences to the
shared cache come from other cores, then they can cause at most \( B \) cache misses. Hence, if there are at least \( B \) cache hits among the left branches, then the WCET with interferences would be \( \sum_{i=1}^{n} t_i^l + B \). However, if that is not the case, i.e. if \( \sum_{i=1}^{n} h_i^l < B \), then the maximum execution time with interferences by picking the left-hand branch in each segment would be \( \sum_{i=1}^{n} (t_i^l + h_i^l) \). If for some \( i \), \( h_i^r > h_i^l \), then by taking the right-hand branch, we would able to increase the execution time by \( h_i^r - h_i^l - (t_i^l - t_i^r) \), by making use of \( h_i^r - h_i^l \) extra interferences.

Notice that the WCIP problem in this case boils down to finding the segments where the right-hand branch must be taken, i.e. finding the worst-case path in the program. Once the worst-case path is known, finding the distribution of interferences is trivial, as we can simply assign one interference before each cache hit on the path, until we run out of interferences or cache hits.

We can now see the resemblance to the 0-1 Knapsack Problem (KP), in which there are \( n \) objects each with value \( v_i \) and weight \( w_i \) and a total weight budget of \( W \), and the problem is to select a subset of objects whose total weight is at most \( W \) and which maximizes the total value. Taking object \( i \) in KP as segment \( i \) in WCIP, selecting object \( i \) would be equivalent to selecting the right-hand branch of segment \( i \), which would result in a ‘value’ of \( h_i^r - h_i^l - (t_i^l - t_i^r) \), with an associated ‘weight’ of \( h_i^r - h_i^l \). The total weight budget would be the extra interferences which were not used on the left-hand branches, i.e. \( B - \sum_{i=1}^{n} h_i^l \).

The only issue with this reduction is that in WCIP, it is not necessary that all the cache hits in a selected branch may be converted to cache misses due to interferences. In other words, \( h_i^r - h_i^l - (t_i^l - t_i^r) \) is only the maximum value available by selecting the right-hand branch in segment \( i \), and the worst-case distribution may result in a lower value, if it does not distribute interferences before all cache hits on the right-hand branch. On the other hand, in 0-1 KP, if an object \( i \) is selected, it is guaranteed to increase the total value by \( v_i \). To solve this dilemma, we will use the fact that there always exists a worst-case distribution which will try to convert all the cache hits to cache misses in a selected branch (Lemma 2), to formally show the reduction.

Let us first define the decision version of the WCIP problem in the restricted setting:

**WCIP Problem:** Given a simple branched program \( \mathcal{P} \) with \( n \) segments (shown in Figure 3.3), an interference budget \( B \) and a target execution time \( T \), does there exist a path in the program (represented by the function \( \sigma : \{1, \ldots, n\} \rightarrow \{l, r\} \)), and assigned interferences \( b_1, \ldots, b_n \) on
this path such that

\[ \forall i, 1 \leq i \leq n, \quad b_i \leq h_i^{\sigma(i)} \]  
(3.10)

\[ \sum_{i=1}^{n} b_i \leq B \]  
(3.11)

\[ \sum_{i=1}^{n} t_i^{\sigma(i)} + b_i \geq T \]  
(3.12)

An interference distribution which obeys equations 3.10 and 3.11 is called a valid distribution. The following lemma states that given any path in the program and a valid interference distribution, there exists another path and a valid distribution, which will result in equal or higher execution time, and which will try to distribute interferences such that all cache hits in a selected segment will become misses.

**Lemma 2.** Given a program \( P \) with \( n \) segments (as shown in Figure 3.3), an interference budget \( B \), assume that \( \sum_{i=1}^{n} h_i^l \leq B \). Given a path represented by \( \sigma \) and assigned interferences \( b_1, \ldots, b_n \) which form a valid distribution, there exists another path \( \hat{\sigma} \), and valid distribution \( \hat{b}_1, \ldots, \hat{b}_n \), with the following properties:

1. If \( \hat{\sigma}(i) = l \), then \( \hat{b}_i = h_i^l \).

2. There exists at most one \( j \) such that \( \hat{\sigma}(j) = r \) and \( \hat{b}_j < h_j^r \).

3. \[ \sum_{i=1}^{n} t_i^{\sigma(i)} + b_i \leq \sum_{i=1}^{n} t_i^{\hat{\sigma}(i)} + \hat{b}_i \]

**Proof.** We will perform a series of redistribution of interferences from the initial distribution to ensure that properties (1) and (2) are met. After each redistribution, we will also ensure that the total execution time either remains the same or it increases (i.e. property (3) remains true).

First, suppose \( \exists i, \sigma(i) = l \) and \( b_i < h_i^l \). Let \( b_{rem} = h_i^l - b_i \).

If \( \forall j, \sigma(j) = l \), then we can take \( \hat{\sigma}(j) = l, \hat{b}_j = h_j^l \) for all \( j \). This satisfies property (1) and (2). Also, \( \hat{b}_i = h_i^l > b_i \), hence the total execution time has increased. Hence, \( \hat{\sigma} \) and \( \hat{b}_i \) satisfy all the properties of the lemma and we are done.

Suppose \( \exists j, \sigma(j) = r \). Then \( b_j - h_j^r > t_j^l - t_j^r \).

Let \( \hat{b}_j = b_j - b_{rem} \).

**Case - 1:** If \( \hat{b}_j - h_j^l > t_j^l - t_j^r \), consider the new distribution \( \hat{\sigma}(k) = \sigma(k), \forall k, \hat{b}_i = h_i^l, \hat{b}_j = b_j^l \),
\( \hat{b}_k = b_k \) for all other \( k \).

\[
\sum_{k=1}^{n} \hat{b}_k = h_i^l + b_j - b_{rem} + \sum_{k \neq i,j} b_k \\
= b_i + b_j + \sum_{k \neq i,j} b_k \\
= \sum_{k=1}^{n} b_k \leq B
\]

\[
\sum_{k=1}^{n} (t_{\tilde{\sigma}(k)}^\ast + \hat{b}_k) - \sum_{k=1}^{n} (t_{\sigma(k)}^\ast + b_k) \\
= t_i^l + h_i^l + t_j^r + b_j - b_{rem} - t_i^l - b_i - t_j^r - b_j \\
= 0
\]

In this case, we have redistributed the interferences such that \( \tilde{\sigma}(i) = l \) and \( \hat{b}_i = h_i^l \), while ensuring the total execution time remains the same.

**Case - 2:** If \( b_j' - h_j^l \leq t_i^l - t_j^r \), consider \( \tilde{\sigma}(j) = l \), \( \hat{S}(k) = S(k) \) for all other \( k \), \( \hat{b}_j = h_j^l \), \( \hat{b}_i = b_i + (b_j - h_j^l) \), \( \hat{b}_k = b_k \) for all other \( k \). Again,

\[
\sum_{k=1}^{n} \hat{b}_k = b_i + b_j - h_j^l + h_j^l + \sum_{k \neq i,j} b_k \\
= \sum_{k=1}^{n} b_k
\]

\[
\sum_{k=1}^{n} (t_{\tilde{\sigma}(k)}^\ast + \hat{b}_k) - \sum_{k=1}^{n} (t_{\sigma(k)}^\ast + b_k) \\
= t_i^l + b_i + b_j - h_j^l + t_j^r + h_j^l - t_i^l - b_i - t_j^r - b_j \\
= t_j^l - t_j^r \geq 0
\]

In this case, by redistributing the interferences, we have \( \tilde{\sigma}(j) = l \), \( \hat{b}_j = h_j^l \), while \( \hat{b}_i > b_i \). By repeating the process with other \( j' \) such that \( \sigma(j') = r \), we can continue to increase \( \hat{b}_i \) until it reaches \( h_i^l \). If there is no such \( j' \), then we can simply set \( \hat{b}_i \) to \( h_i^l \), since \( \sum_{k=1}^{n} h_k^l \leq B \).

Thus, we have shown that we can always redistribute the interferences used on the right...
branch of some segment to the left branch of another segment. It is optimal to use as many interferences as possible on the left branches.

After performing the above transformations, we can now assume property 1. We will now give the redistribution strategies for proving property 2.

Suppose \( \exists i, j \) such that \( \sigma(i) = r, b_i < h_i^r, \sigma(j) = r, b_j < h_j^r \). Then \( b_i - h_i^r > t_i^l - t_i^r \) and \( b_j - h_j^r > t_j^l - t_j^r \). Let \( b_i^{rem} = h_i^r - b_i \) and \( b_j^{rem} = h_j^r - b_j \).

**Case - 1:** If \( b_i^{rem} \geq b_j - h_j^l \), then let \( \hat{\sigma}(j) = l, \hat{b}_j = h_j^l, \hat{\sigma}(i) = r, \hat{b}_i = b_i + b_j - h_j^l \leq b_i + b_i^{rem} = h_i^r \), for all other \( k \), \( \hat{\sigma}(k) = \sigma(k) \) and \( \hat{b}_k = b_k \).

\[
\sum_{k=1}^{n} \hat{b}_k = b_i + b_j - h_j^l + h_i^l + \sum_{k \neq i,j} b_k
\]

\[
= \sum_{k=1}^{n} b_k \leq B
\]

\[
\sum_{k=1}^{n} (t_{k}^{\hat{\sigma}(k)} + \hat{b}_k) - \sum_{k=1}^{n} (t_{k}^{\sigma(k)} + b_k)
\]

\[
= t_i^r - b_i + b_j - h_i^l + h_i^l - t_i^r - b_i - t_j^r - b_j
\]

\[
= t_j^l - t_j^r \geq 0
\]

In this case, by redistribution, we have only one segment \((i)\) where \( \hat{\sigma}(i) = r \) and \( \hat{b}_i < h_i^r \). Similarly, if \( b_j^{rem} \geq b_i - h_i^l \), a similar proof will work.

**Case - 2:** Now, suppose that neither of the two conditions are true. In particular, \( b_i^{rem} < b_j - h_j^l \). There are two subcases to consider:

**Subcase - 1:** \( b_i^{rem} \leq b_j - h_j^l - (t_j^l - t_j^r) \). Hence, \( (b_j - b_i^{rem}) - h_j^l > t_j^l - t_j^r \). Let \( \hat{\sigma}(i) = r, \hat{b}_i = h_i^r, \hat{\sigma}(j) = r, \hat{b}_j = b_j - b_i^{rem} \), and for all other \( k \), \( \hat{\sigma}(k) = \sigma(k) \) and \( \hat{b}_k = b_k \).

\[
\sum_{k=1}^{n} \hat{b}_k = h_i^r + b_j - b_i^{rem} + \sum_{k \neq i,j} b_k
\]

\[
= \sum_{k=1}^{n} b_k \leq B
\]
\[
\sum_{k=1}^{n}(t_k^{\hat{\sigma}(k)} + \hat{b}_k) - \sum_{k=1}^{n}(t_k^{\sigma(k)} + b_k) \\
= t_i^r + h_i^r + t_j^r + b_j - b_i^{rem} - t_i^r - b_i - t_j^r - b_j \\
= 0
\]

Hence, by redistribution, we now have only one segment \((j)\) where all cache hits are not converted to misses on the right branch.

**Subcase - 2:** \(b_i^{rem} > b_j - h_j^l - (t_j^l - t_j^r)\). Let \(\hat{\sigma}(i) = r, \hat{b}_i = h_i^r, \hat{\sigma}(j) = l, \hat{b}_j = h_j^l,\) and for all other \(k, \hat{\sigma}(k) = \sigma(k), \hat{b}_k = b_k.\)

\[
\sum_{k=1}^{n} b_k - \sum_{k=1}^{n} \hat{b}_k = b_i + b_j - h_i^r - h_j^l \\
= b_j - h_j^l - b_i^{rem} > 0 \text{ (by assumption)}
\]

Hence, \(\sum_{k=1}^{n} \hat{b}_k < \sum_{k=1}^{n} b_k \leq B.\)

\[
\sum_{k=1}^{n}(t_k^{\hat{\sigma}(k)} + \hat{b}_k) - \sum_{k=1}^{n}(t_k^{\sigma(k)} + b_k) \\
= t_i^r + h_i^r + t_j^r + h_j^l - t_i^r - b_i - t_j^r - b_j \\
= b_i^{rem} - (b_j - h_j^l - (t_j^l - t_j^r)) > 0
\]

Again, we have ensured that all cache hits are converted to misses in segments \(i\) and \(j.\) Thus, we have showed that by redistributing the interferences, we can always ensure that whenever the left branch is taken in a segment, all the cache hits are converted to misses, and there is at most one segment where the right branch is taken, but all cache hits are not converted to misses.

Property 1 of Lemma 2 guarantees that all cache hits in left-hand branches which are selected by \(\hat{\sigma}\) will be converted to misses. It may not be possible to guarantee the same about the right-hand branches, because the number of cache hits on the right-hand branches can be arbitrarily high, but property 2 of Lemma 2 guarantees that there will be at most one right-hand branch, where all cache hits are not converted to misses. Property 3 shows that the new distribution
will yield equal or higher execution time.

The proof essentially uses a series of redistribution of interferences from the initial distribution to ensure that properties 1 and 2 are met, while not decreasing the execution time. The main idea is that if all cache hits are not converted to misses in some selected segment, then interferences can be borrowed from some other selected segment, and this can be done until all hits becomes misses. We can safely assume that any optimal distribution of interferences will follow the rules set down by Lemma 2. We now define the decision version of the 0-1 Knapsack problem, which is known to be NP-Hard [48].

**0-1 Knapsack Problem:** Given a set of \( n \) items each with value \( v_i \) and weight \( w_i \) \((1 \leq i \leq n)\), a weight budget \( W \) and a target value \( V \), does there exist a subset \( P \) of items such that

\[
\sum_{i \in P} w_i \leq W \tag{3.13}
\]

\[
\sum_{i \in P} v_i \geq V \tag{3.14}
\]

We assume that the values and weights are positive integers. Given an instance of the knapsack problem, we first convert it to another knapsack problem where the weights are greater than the values for all items. Let \( v_m \) be the maximum value among all the \( n \) items. In the new problem, item \( i \) will have the same value \( v_i \) and a new weight \( w'_i = v_m w_i \). The weight budget will be \( W' = v_m W \), while the target value remains \( V \). Since \( w_i > 0 \), \( v_m w_i \geq v_m \geq v_i \) for all \( i \). It is easy to see that any solution of the original knapsack problem will also be a solution of the modified problem, and vice versa.

We now construct a simple-branched program \( P_{KP} \) (shown in Figure 3.4), along with the interference budget and target execution time, in such a way the solution to the WCIP problem in this program corresponds to the solution of the modified knapsack problem. \( P_{KP} \) has \( n \) segments, with \( t'_i = w'_i - v_i \), \( h'_i = 0 \), and \( t''_i = 0 \), \( h''_i = w'_i \), for all \( i \). The interference budget is \( B = W' \), and the target execution time is \( T = \sum_{i=1}^{n} (w'_i - v_i) + V \). Note that \( t''_i \) can also be taken as any constant \( C \), in which case \( t'_i \) should be \( w'_i - v_i + C \). The selection of execution times and shared cache hits should ensure that the ‘profit’ of taking the right-side branch in segment \( i \) should be \( v_i \).

Note that \( \sum_{i=1}^{n} (w'_i - v_i) \) is the execution time of the program, if the left branch is taken in all segments. However, no interferences can be used on the left branch, and taking the right
branch in segment $i$ will have a profit (i.e. increase in execution time) of $v_i$, but associated
weight (i.e. interferences used) of $w'_i$. Let us prove the reduction formally:

\textbf{Theorem 1.} The WCIP problem in a simple-branched program, assuming a direct-mapped
shared cache with single cache set, is NP-Hard.

\textit{Proof.} We need to show that: $\exists P \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in P} w'_i \leq W'$ and $\sum_{i \in P} v_i \geq V \iff$

There exists a path $\sigma$ and a valid interference distribution $b_1, \ldots, b_n$, such that $P_{KP}$ achieves
the target execution time $T = \sum_{i=1}^{n} (w'_i - v_i) + V$.

($\Rightarrow$) We are given a solution to KP, which achieves the target value. To obtain a solution
to the WCIP problem, we simply pick the right-hand branch in those segments $i$ where the
corresponding item $i$ has been selected in solution to KP. We will also assign interferences
before all cache hits in the selected right-side branches. We define $\sigma : \{1, \ldots, n\} \rightarrow \{l, r\}$ and
$b_i$ as follows:

\[
\sigma(i) = \begin{cases} r & \text{if } i \in P \\ l & \text{otherwise} \end{cases}
\]

\[
b_i = \begin{cases} w'_i & \text{if } i \in P \\ 0 & \text{otherwise} \end{cases}
\]

First, we need to show that this is a valid interference distribution. Equation 3.10 is trivially
satisfied.

\[
\sum_{i=1}^{n} b_i = \sum_{i \in P} w'_i \leq W'
\]

This shows that Equation 3.11 is also satisfied (Note that $B = W'$). We now show that this
interference distribution achieves the target execution time $T$.

$$
\sum_{i=1}^{n} t_{i}^{\sigma(i)} + b_i = \sum_{i \in P} w_i + \sum_{i \notin P} w_i - v_i \\
= \sum_{i \in P} (w_i - v_i + v_i) + \sum_{i \notin P} w_i - v_i \\
$$

(adding and subtracting $v_i$, $\forall i \in P$)

$$
= \sum_{i=1}^{n} (w_i - v_i) + \sum_{i \in P} v_i \\
$$

(rearranging terms)

$$
\geq \sum_{i=1}^{n} (w_i - v_i) + V = T
$$

Thus, $P_{KP}$ achieves the target execution time under $S$ and $b_1, \ldots, b_n$.

($\Leftarrow$) We are given $\sigma, b_1, \ldots, b_n$ such that the target execution time $T$ is achieved and the interference distribution is valid (i.e. it satisfies equations 3.10,3.11,3.12). Note that if $\sigma(i) = l$, then $b_i = 0$, and by Lemma 2, we can assume there is at most one $j$ such that $\sigma(j) = r$ and $b_j < w_j'$. To obtain the solution of KP, we will pick the item $i$ if the right-hand branch has been taken in the corresponding segment $i$. Hence $P = \{i|\sigma(i) = r\}$. First, we show that this selection of items meets the target value $V$.

$$
\sum_{i=1}^{n} t_{i}^{\sigma(i)} + b_i \geq \sum_{i=1}^{n} (w_i' - v_i) + V \\
\Rightarrow \sum_{i: \sigma(i) = l} (w_i' - v_i) + \sum_{i: \sigma(i) = r} b_i \geq \sum_{i=1}^{n} (w_i' - v_i) + V \\
\Rightarrow \sum_{i: \sigma(i) = r} v_i \geq V + \sum_{i: \sigma(i) = r} (w_i' - b_i)
$$

Since $\forall i, \sigma(i) = r \Rightarrow b_i \leq w_i'$, $\sum_{i: \sigma(i) = r} v_i \geq V$. Hence, we have shown that the target value is met. Next, we will show that $\sum_{i: S(i) = r} w_{i}' \leq W'$. We consider two cases:

Case 1: If $\forall i, \sigma(i) = r \Rightarrow b_i = w_i'$, then

$$
\sum_{i: \sigma(i) = r} b_i \leq w_i' \Rightarrow \sum_{i: \sigma(i) = r} w_{i}' \leq W'.
$$

Case 2: Suppose $\exists j$, such that $\sigma(j) = r$ and $b_j < w_j'$ (by Lemma 2 this means that $b_i = w_i'$,
for all $i$ such that $\sigma(i) = r$ and $i \neq j$). Note that in this case, $b_j > w'_j - v_j$. Otherwise, if $b_j \leq w'_j - v_j$, then we can modify the worst-case path to take the left-hand branch in segment $j$, i.e. $\sigma(j) = l$, which will only increase the execution time. Now, since for all other $i$, $\sigma(i) = r \Rightarrow b_i = w'_i$ implies the result (by case 1).

Hence, $b_j > w'_j - v_j \Rightarrow v_j > w'_j - b_j$. We know that $w'_j = v_m w_j$. We will now show that $b_j$ should also be a multiple of $v_m$.

Again, in this case, $\sum_{i: \sigma(i) = r} b_i = W'$, because otherwise if $\sum_{i: \sigma(i) = r} b_i < W'$, we can increase $b_j$ such that either $b_j$ becomes $w'_j$ or the sum becomes equal to $W'$. We will only be increasing the execution time, and we have already proved the result if $b_j$ becomes equal to $w'_j$ (in case 1).

$$\sum_{i: \sigma(i) = r} b_i = W' \Rightarrow b_j = W' - \sum_{i: \sigma(i) = r \neq j} b_i$$

$$\Rightarrow b_j = W' - \sum_{i: \sigma(i) = r \neq j} w'_i$$

$$\Rightarrow b_j = v_m W' - \sum_{i: \sigma(i) = r \neq j} v_m w_i$$

$$\Rightarrow b_j = v_m p \quad \text{(where } p > 0)$$

But, $v_j > w'_j - b_j \Rightarrow v_j > v_m w_j - v_m p \Rightarrow v_j > v_m q$. This is a contradiction, as $q > 0$ and $v_m$ is the maximum of all values. Hence there cannot exist $j$ such that $\sigma(j) = r$ and $b_j < w'_j$. Hence, $\forall i, \sigma(i) = r \Rightarrow b_i = w'_i$. Hence, the set $P = \{i | \sigma(i) = r\}$ is a solution to KP.

Although we have only looked at WCIP for simple-branched programs, assuming a DM cache with a single cache set, for the most general setting, we have to consider programs with nested branches and loops, and set-associative caches with multiple cache sets. These additions are only going to increase the complexity of the problem.

### 3.7 Approximate WCIP

In this section, we present an efficient, approximate approach for WCIP. As shown in the previous section, the difficulty in WCIP arises from the difference in the execution times and number of shared cache hits, along different program paths. A program path with high execution time may not have enough shared cache hits to ‘use’ all the interferences, while there may be program paths with large number of shared cache hits but lower execution times. To bypass this problem, we assume that all the shared cache hits are present on the worst-case path.
(calculated assuming no interferences). We then propose a simple greedy algorithm to perform WCIP, which simply picks those cache hits in the program which have the highest chance of becoming misses, since the cache blocks they access are already close to eviction without interferences. The total time complexity of our approach is linear in the program size.

3.7.1 Setup

As with ILP-based WCIP, we first perform AI-based multi-level Must and May cache analysis for the instruction cache hierarchy [34] to obtain an initial cache access classification (CAC) and cache hit miss classification (CHMC) of all memory accesses at the shared cache level. We consider all program accesses with a CAC of Always or Uncertain and a CHMC of Always Hit at the shared cache level.

Let $Acc$ and $Acc_H$ be the set of all instructions in the program which may access the L2 cache and are guaranteed to cause a L2 cache hit, respectively. Hence, for all $a \in Acc$, CAC($a$) is Always or Uncertain, while for all $a \in Acc_H$, CHMC($a$) is Always Hit. Clearly, $Acc_H \subseteq Acc$.

Let $Age(a)$ be the age of the cache block accessed by $a$ in the L2 Must cache at the program point just before the access. The Eviction Distance of an access $a$ is given by $A - Age(a) + 1$. The eviction distance of an access is the minimum number of interferences required to evict the cache block just before the access. The concept of eviction distance is similar to the resilience of a cache block, as defined in [13]. For an access $a$ referencing cache block $m$, if $P_a$ is the program point just before the access $a$, and if $res_{P_a}(m)$ is the resilience of $m$ at $P_a$, then the eviction distance of $m$ would be $res_{P_a}(m) + 1$.

Any access to the shared L2 cache made by a program will act as an interference to the program(s) running on other core(s). Hence, we count all the actual accesses made by the interfering programs, i.e. the programs running on other cores, whose CAC at L2 is Always or Uncertain, to obtain the number of interfering accesses suffered by the program under analysis. If the access is inside a loop of the interfering program, then we use the loop bound to count the interferences caused by the access. In this way, we obtain the number of interferences, $B_s$ and the number of interfering cache block $B_{s}^{th}$ to every cache set $s$. Let $H$ be the actual number of cache hits of the program under analysis (obtained by counting every access in $Acc_H$, considering its loop bound if the access is inside a loop).

3.7.2 Motivation

The source of hardness for the WCIP problem (as shown in the previous section) lies in finding the worst case path in the presence of interferences. One way to bypass this issue is to first find the worst case path assuming no interferences (say $\pi_{wc}$), and then determine an upper bound
on the increase in execution time due to interferences across all paths. Interferences will convert some of the shared cache hits into misses, but we cannot simply consider the shared cache hits present on $\pi_{wc}$ to calculate the upper bound. Instead, a safe option would be to consider all the shared cache hits present in the program, and then find the maximum number of misses generated by interferences among them.

Let $\pi_{\text{Int}}$ be the worst-case path in the program in the presence of interferences, and let $T_{\text{wc}}^{\text{Int}}$ and $I_{\text{Int}}$ be its execution time without interferences, and the maximum increase in its execution time due to interferences, respectively. Hence $T_{\text{wc}}^{\text{Int}} + I_{\text{Int}}$ is the WCET of $\pi_{\text{Int}}$ (also the WCET of the program) in the presence of interferences. As mentioned earlier, $\pi_{\text{wc}}$ is the worst-case path in the program in the absence of interferences, and let $T_{\text{wc}}$ be its execution time.

Then, $T_{\text{wc}} \geq T_{\text{wc}}^{\text{Int}}$. Our strategy is to find the maximum number of cache misses due to interferences among all the shared cache hits in the program, and determine the resulting increase in execution time, $I$. Since this will also include all the shared cache hits present on $\pi_{\text{Int}}$, clearly, $I \geq I_{\text{Int}}$. Hence, $T_{\text{wc}} + I \geq T_{\text{wc}}^{\text{Int}} + I_{\text{Int}}$.

The worst case path and its WCET without interferences ($T_{\text{wc}}$) can be easily determined, and hence our objective now is to calculate $I$, for which we need to determine the maximum number of cache misses that interferences can cause among $\text{Acc}_H$. We know that the eviction distance of a cache hit is the minimum number of interferences required to convert it to a cache miss. Hence, if the number of interferences assigned just before a cache hit is equal to its eviction distance, then the access can miss the cache.

It is easy to see that the optimal strategy to maximize the number of cache misses, would be the greedy strategy of selecting cache hits in increasing order of their eviction distances. If cache hits with lower eviction distance are selected first, and interferences are assigned before them, then this would ensure that more interferences are available for later cache hits, thus maximizing the impact of every interference. However, before using this strategy, we must account for the overlapping effect of interferences.

### 3.7.3 The overlapping effect

We say that an interference affects a cache hit $a$, when it increases the age of the cache block $m$ which will eventually be accessed by $a$, without any intervening accesses to $m$ between the interference and $a$. Obviously, any interference which occurs just before the cache hit $a$ will affect it. However, any interference which occurs on a cache hit path of $a$ will also affect $a$. Since the cache hit path can contain other cache hits, interferences which occur before these hits can also affect the access $a$. The implication is that the greedy strategy will not work, because it only considers the impact of those interferences which are assigned just before the
Figure 3.5: Example to illustrate the overlapping effect

cache hit.

As an example, consider the program fragment shown in Figure 3.5, containing instructions $a, b, c$ which access cache blocks $m_1, m_2, m_3$ respectively, all mapping to the same cache set, and hitting the cache. Assume the cache associativity is 4. Since the age of all the three cache blocks in the Must cache will be 3, their eviction distance will be 2. Now, any interference which occurs just before the access $a$ is going to affect the accesses $b$ and $c$ as well. If the sequence of accesses made by the program is $b - a - c - b - a - c - b - \ldots$, then assigning 2 interferences just before the access $a$ will result in a cache miss for the second access to $b$. This shows that it is not enough to simply consider the interferences assigned just before the cache hit, but we must also consider the impact of interferences assigned before other cache hits.

The Overlapping Factor (OF) of a cache hit $a$ is defined as the maximum number of cache hits that a single interference just before $a$ can affect. An interference before $a$ will affect the next access to any cache block that is present in the cache set just before $a$. If, just before the cache hit $a$, the cache set is full (i.e. it contains $A$ cache blocks), then an interference before $a$ can affect the next access to each of the $A$ cache blocks. In the example program shown in Figure 3.5, the OF of each of the three cache hits $a, b$ and $c$ is 3. This is because an interference occurring before any of the three cache hits will affect all the three.

We can use cache hit paths to calculate the OF. Only those interferences which occur within a cache hit path of an access can affect that access. Hence, if a cache hit $h_1$ is present in a hit-path of another cache hit $h_2$, then any interference occurring before $h_1$ will affect $h_2$. The overlapping factor of cache hit $h$ will be the number of cache hits who have a cache hit path which contains $h$. In our example program, $a$ is present in the hit path $a - a$ of cache hit $a$,
Figure 3.6: Example which demonstrates that the greedy method based on eviction distance is not optimal

\[ b - a - b \text{ of } b, \text{ and } c - a - c \text{ of } c. \]  
Similar observations can be made about \( b \) and \( c \).

Different values of OF and the eviction distance complicate worst case distribution of interferences, because we cannot directly use the greedy strategy of considering cache hits in increasing order of eviction distances. For example, as shown in Figure 3.6, consider the cache hits \( h_1, h_2 \) and \( h_3 \) with eviction distances 2, 3 and 1 respectively. The cache hit paths of \( h_1 \) and \( h_2, h_2 \) and \( h_3 \) overlap. Both \( h_1 \) and \( h_2 \) have OF 2, while the OF of \( h_3 \) is 1. With an interference budget of 3, all the three cache hits can be converted to misses by assigning 2 interferences before \( h_1 \) and 1 interference before \( h_2 \). However, if we assign interferences based on increasing eviction distances, we will first assign 1 interference before \( h_3 \) and then 2 interferences before \( h_1 \), thus using up the interference budget and obtaining only 2 cache misses.

Other selection strategies such as decreasing order of overlapping factors, or increasing order of eviction distance of overlapped cache hits also do not work. This is where we make our second approximation, by removing the overlapping effect through an increase in the number of interferences. If a cache hit \( a \) has an OF of \( o \), and if \( z \) interferences occur before \( a \), then they will affect all the \( o \) cache hits, and the effect is equivalent to having \( oz \) interferences and assigning \( z \) interferences individually before each cache hit.

Remember that \( B_s \) is the interference budget for cache set \( s \). We find the maximum overlapping factor among all cache hits in the program mapped to \( s \). If \( o_s \) is the maximum OF,
Algorithm 2: Algorithm to find the maximum overlapping factor

**Input**: Cache hits mapped to each cache set $s$, Hit paths of every cache hit

**Output**: Maximum overlapping factor $o_s$ for every cache set $s$

1. **for every cache set** $s$ **do**

2. \[ o_s \leftarrow 0 ; \]

3. **for every cache hit** $a \in Acc_H$ **mapped to** $s$ **do**

4. \[ OF_a \leftarrow 0 \]

5. **end**

6. **for every cache hit** $a \in Acc_H$ **mapped to** $s$ **do**

7. **for every hit** $a' \in Acc_H$ **present in a hit path of** $a$ **do**

8. \[ OF_{a'} \leftarrow OF_{a'} + 1 \]

9. **end**

10. **end**

11. **for every cache hit** $a$ **mapped to** $s$ **do**

12. **if** $OF_a > o_s$ **then**

13. \[ o_s \leftarrow OF_a \]

14. **end**

15. **end**

16. **end**

then we take $o_s B_s$ to be the new interference budget for cache set $s$. We can now safely assume no overlapping while using the new interference budget. In other words, we can now assume that only those interferences which occur just before a cache hit will affect it. In the example of Figure 3.6, the maximum OF is 2, hence the interference budget would become 6. Now, the eviction distance of each cache hit can be met, thus resulting in 3 cache misses.

In general, this is safe because any distribution of interferences with the original budget can be converted into a new distribution with the new budget and assuming no overlap. Hence, for the worst-case distribution with the original budget and overlap, there will also exist a distribution with the new budget, which will not use any overlap. We will find the worst-case distribution with the new budget and assuming no overlap.

Algorithm 2 is used to find the maximum overlapping factor for each cache set. It goes through every hit path of every cache hit, and finds the overlapping factor of each cache hit, which is then used to find the maximum OF for the cache set. Since the size and number of hit paths are bounded, the inner for loop (lines 7-9) will run for a constant number of iterations. Hence, the algorithm has a complexity of $O(|Acc_H|)$, which will be linear in the code size.
3.7.4 Interference Distribution Algorithm

Since we are assuming that there is no overlapping effect, the optimal interference distribution strategy is to select cache hits in the increasing order of their eviction distances. Figure 3.7 shows the overall scheme for approximate WCIP. Given a program, we first perform the multi-level cache analysis, and calculate the WCET of each basic block in the program, assuming no interferences. By performing the cache analysis, we can determine all the shared cache hits in the program, as well as their eviction distances. We also obtain information about all the shared cache accesses made by the program, which will act as interferences to programs running on other cores.

We find the cache hit paths of all the cache hits, and then find the maximum overlapping factor $o_s$ for each cache set $s$, using Algorithm 2. The cache hit information, overlapping factor, and the interference information is used by the interference distribution algorithm (explained below), to obtain the maximum increase in WCET due to cache misses caused by interferences. This is simply added to the WCET obtained by the IPET (Implicit Path Enumeration Technique) ILP to obtain the final WCET.

We call a cache hit having an eviction distance of $k$ as a $k$-interference cache hit ($1 \leq k \leq A$). We count all the $k$-interference cache hits in the program, using the L2 must cache and loop bounds. Algorithm 3 shows our interference distribution strategy.

The algorithm assigns interferences to cache hits in increasing order of their eviction distances, for each cache set. First, we multiply the interference budget with the overlapping factor to get the new budget (line 3). Next, we check if the number of distinct cache blocks accessed by interferences is greater than $k$ (line 5). If this is not the case, then no cache misses can caused by interferences, because $k$-interference cache hits require interferences to at least $k$ distinct blocks to become misses.
Algorithm 3: Interference distribution algorithm

**Input:** Number of interferences $B_s$, Number of interfering cache blocks $B_s^{cb}$ for each cache set $s$, Number of $k$-interference cache hits $numhits_s^k$ in the program, for each cache set $s$ ($1 \leq k \leq A$), Overlapping factor $o_s$ for each cache set, Shared cache miss penalty $c_p$

**Output:** Maximum increase in WCET due to interferences, $I$

1. $I \leftarrow 0$;
2. for every cache set $s$ do
3.     $B_s \leftarrow o_s B_s$;
4. for $k \leftarrow 1$ to $A$ do
5.     if $B_s^{cb} \geq k$ then
6.         if $B_s \leq k \times numhits_s^k$ then
7.             $I \leftarrow I + (\lceil \frac{B_s}{k} \rceil \times c_p)$;
8.             $B_s \leftarrow 0$;
9.         else
10.            $I \leftarrow I + (numhits_s^k \times c_p)$;
11.            $B_s \leftarrow B_s - (numhits_s^k \times k)$;
12.        end
13.    end
14. end
15. end
Then, we check whether there are enough \( k - \text{interference} \) cache hits to use all interferences (line 6). If yes, then all interferences are assigned \((k \text{ interferences before each cache hit})\), resulting in a maximum of \( \lceil \frac{B_s}{k} \rceil \) misses. The cache miss penalty is added to the WCET for each of these misses, and the interference budget is updated to 0 (lines 7-8). If there aren’t enough \( k - \text{interference} \) cache hits to use all interferences, then the cache miss penalty is added for all \( k - \text{interference} \) cache hits, and the interference budget is decreased (lines 10-11), and we continue the interference distribution in the next iteration with \((k + 1) - \text{interference} \) cache hits and the remaining interferences.

### 3.7.5 Algorithm analysis

For a shared cache with associativity \( A \) and number of cache sets \( S \), Algorithm 3 has a time complexity of \( O(SA) \). Combined with Algorithm 2, the total complexity of our approach is linear in the program size and the shared cache size. The algorithm introduces imprecision, on account of the two approximations made to simplify the analysis. First, it assumes that all cache hits of the program are on the worst case path without interferences. However, it is possible that this worst case path may have very few cache hits, and non-worst-case paths with many shared cache hits may have small execution time without interferences. Second, we multiplied the original interference budget with the maximum overlapping factor \( o_s \), with the inherent assumption that every interference to set \( s \) affects \( o_s \) cache hits, which may not be true.

In general, the algorithm will compute a maximum WCET increase of \( (\sum_{\text{all sets } s} o_s B_s) c_p \) (if there are enough cache hits to use all interferences). An important property of the algorithm is that the maximum increase in execution time due to interferences is directly proportional to the number of interferences. This ensures that if the cache interference is low, then the increase in WCET due to cache interference will also be small. All previous approaches to shared cache analysis do not have this property. Also, the increase in execution time due to interferences is a multi-dimensional, piecewise linear function of the number of interferences. Specifically, for some cache set \( s \), if we plot the increase in WCET versus the number of interferences, then we will have a line with slope \( c_p \) until \( \text{numhits}_s^1 \) interferences, then a line with slope \( \frac{c_p}{2} \) until \( \text{numhits}_s^1 + 2\text{numhits}_s^2 \) interferences, and so on.

Since we cannot find the worst-case path in the presence of interferences efficiently, we must make the worst-case assumptions about it, which translates to the worst-case assumptions about the number of shared cache hits, maximum OF and eviction distance. As far as the approximations regarding the maximum OF, this value can be expected to be small (generally less than the cache associativity), because an interference at a program point will only affect
the accesses to the cache blocks which are guaranteed to be present at that program point. In our experiments, the OF for most of the benchmarks was 1, and it never exceeded the cache associativity.

Instead of considering all the shared cache hits in the program, we could also find the maximum number of cache hits that could happen on a program path. For this, we can use the IPET ILP, modifying the objective function to consider the number of cache hits in a basic block, instead of its execution time. In our experiments, this did not have any impact on the precision of WCIP. Note that to find the maximum overlapping factor, we must still consider all the cache hits in the program.

Handling Code Sharing: We can simply ignore the effect of code sharing during our analysis, since this only affects the precision of the analysis. In the presence of sharing, interfering cache blocks may already have been brought into the cache by the program under analysis, in which case they may not cause eviction of other cache blocks. Consider instruction \(a\), which accesses cache block \(m\) mapped to cache set \(s\), and let \(\pi\) be a cache hit path of the instruction. Let \(M^\pi\) be the set of cache blocks accessed in \(\pi\). Let \(M_s\) be the set of cache blocks accessed by the interfering program, and mapped to cache set \(s\) (hence, \(B^{ch}_s = |M_s|\)). Interfering accesses which access cache blocks in \(M_s \cap M^\pi\) will never cause eviction of \(m\), because these cache blocks will also be accessed by instructions on the hit path, and hence their impact on \(m\) would have already been considered (during private cache analysis). Hence, we calculate the set \(M_s \setminus M^\pi\), and we ignore the hit path \(\pi\) of \(m\) if \(|M_s \setminus M^\pi|\) is less than the eviction distance of \(m\). If this happens for all hit paths of an access, then we can safely conclude that the access will never experience a miss due to interferences. Otherwise, the hit paths will be ignored while determining the overlapping factor (in Algorithm 2).

### 3.8 Interaction with the shared bus

In most multi-core architectures, accesses to the shared cache have to go through the shared bus, which collects access requests from all cores and sends them to the shared cache. If requests from two different cores arrive at the same time, then one core must wait, because the shared cache can only fulfill one access request at a time. For predictability, it is desirable that the delay suffered due to this interference be bounded statically. One of the most commonly used arbitration policies to ensure bounded delays is the Time Division Multiple Access (TDMA) based round-robin policy. In this policy, each core is assigned a fixed slot of time, and all access requests arriving during this slot will be immediately forwarded to the shared cache (provided those requests can be fulfilled in the same slot). All slots are arranged in a fixed, static schedule which repeats itself. If a core generates an access request outside of its slot, then it must wait
for its next slot. Since the size of the slots as well as the schedule are known, the delay can be accurately bounded.

An obvious bound on the maximum bus delay would be the maximum time between two slots assigned to the same core, obtained by assuming that the access request arrives just after the slot assigned to the core has finished. Using this upper bound for every shared cache access, however, could result in over-approximation, and hence, previous works ([38, 22]) have proposed a more precise timing analysis, by using accurate bounds on the exact timing of each shared cache access.

The time at which a shared cache access happens depends on the hit-miss behavior of previous accesses, and hence, approaches for TDMA-based shared bus analysis require the safe hit-miss classification for every individual access to the shared cache. In our approach, we do not provide a safe hit-miss classification for every access to the shared cache, but instead only provide upper bound on the number of shared cache misses. Providing guarantees for every shared cache access is very difficult, since that would require considering the worst-case interference arrival individually for every access, resulting in high over-approximation of the number of misses. By considering the global worst-case interference arrival, we can guarantee substantially higher number of cache hits. However, we cannot exactly pinpoint where the hits and misses are going to happen during the program execution.

In this section, we show that knowing the maximum number of shared cache misses caused due to interferences is enough to find the maximum shared bus delay that these misses will cause. Hence, our approach for shared cache analysis can be safely integrated with TDMA-based shared bus analysis techniques to accurately bound the shared bus delay. Note that we only concentrate on shared instruction cache accesses, and assume separate busses for instruction and data traffic. Initially, the shared cache behavior of a program in isolation (which will provide precise hit-miss classification for individual accesses) would be used to find the WCET, taking into account the shared bus delays (for example, using the techniques described in [38, 22]). Then, we find the maximum number of shared cache misses caused due to interferences, using approximate WCIP. We can show that every shared cache miss can only cause a maximum bus delay equal to the twice the TDMA period (which is the sum of the length of slots assigned to each core). Hence, the maximum bus delay caused due to interferences would also be directly proportional to the number of shared cache misses, and can be found without pinpointing where the misses occur during execution.

Let $\pi_{\text{wc}}^{\text{Int}}$ be the worst-case path in the presence of interferences. Let $I$ be the maximum number of shared cache misses in the entire program caused by interferences (this number can determined using approximate WCIP). If $I_{\text{Int}}$ is the maximum number of shared cache misses
caused by interferences on the path $\pi_{\text{wc}}^{\text{Int}}$, then clearly, $I^{\text{Int}} \leq I$. Let $B^{\text{Int}}$ be the increase in the shared bus delay on the path $\pi_{\text{wc}}^{\text{Int}}$ which happens due to the shared cache misses caused by interferences.

Let there be $n_c$ cores, and let $s_l$ be the slot length of each core in the TDMA schedule. For simplicity, we assume that the length of each slot is the same, and each core is assigned exactly one slot in the schedule. Hence, the TDMA period will be $n_c s_l$. We will show that $B^{\text{Int}} \leq 2n_c s_l I^{\text{Int}}$. Since $I^{\text{Int}} \leq I$, $2n_c s_l I$ would be a safe upper bound on the maximum increase in shared bus delay.

Let $s_1, s_2, \ldots s_N$ be the sequence of shared cache accesses made during the execution of the worst case path $\pi_{\text{wc}}^{\text{Int}}$. Moreover, let $s_{i_1}, s_{i_2}, \ldots s_{i_l}$ be the accesses in this sequence, which were initially shared cache hits, but became misses due to interferences. Note that $l = I^{\text{Int}}$, the maximum number of misses on the worst-case path.

Upto $s_{i_1}$, there are no shared cache misses caused by interferences, and hence no extra bus delay will be caused. The access $s_{i_1}$ is the first access to experience a miss due to interferences, which will result in an access to the main memory and hence extra cache miss penalty $c_p$. Let $\beta_{i_1}$ be the actual time at which the access $s_{i_1}$ takes place, and $o_{i_1} = \beta_{i_1} \mod (n_c s_l)$ be the offset in the TDMA period.

In the worst-case, this offset can occur just before the slot assigned to the core finishes, in such a way that the original cache hit could be served within the slot, but the cache miss cannot be served within the same slot. Formally, if $[s_l (p - 1), s_l p]$ was the slot of the core issuing the request, and $\gamma_{i_1}$ was the original time required for the cache hit ($\gamma_{i_1} + c_p$ is the new time for the cache miss), then the worst-case happens when $s_l p - o_{i_1} \geq \gamma_{i_1}$, but $s_l p - o_{i_1} < \gamma_{i_1} + c_p$. In this case, the core must wait for the next slot assigned to it, resulting in a bus delay of at most $n_c s_l$, which would not have been encountered in the run without interferences. Note for all other offsets of the access $s_{i_1}$, no extra bus delay would happen due to the cache miss, and the maximum difference between the execution times would be just the cache miss penalty.

Because of the cache miss suffered by $s_{i_1}$, the time of the next shared bus access $s_{i_1 + 1}$ will also change. Here, we use the offset relocation lemma proposed in [38], which states if there are two executions of the same path, with one execution starting at offset $o$ and another starting at a different offset $o'$ in the TDMA period, then assuming identical behavior for every other micro-architectural component except the shared bus, the two executions will differ by at most $n_c s_l$ cycles. The reasoning is that the worst case scenario would be where the new offset would occur just after the assigned slot, while the old offset may be at the beginning of the slot, resulting in the maximum delay of one TDMA period. In our case, the offset of $s_{i_1 + 1}$ will change because of the miss to $s_{i_1}$, from the original offset when $s_{i_1}$ was a hit. However, this will cause a maximum increase of $n_c s_l$ in the execution time of the path starting from $s_{i_1 + 1}$.
Hence, the cache miss to \( s_{i_1} \) can cause a maximum of increase of \( 2n_c s_i \) due to shared bus delays. Each miss \( s_{i_j} \) will cause a similar increase, and hence the total increase in shared bus delay will be upper bounded by \( 2n_c s_i I^{int} \).

### 3.9 Experimental Evaluation

#### 3.9.1 Setup

We use the WCET analyzer Chronos [41] for our experiments. We experimented on 27 benchmarks from the Mälardalen WCET benchmark suite\(^1\). Since most of these benchmarks are fairly small, we also experimented on the benchmark *susan* from the MiBench suite\(^2\), and the *autopilot* module from the PapaBench suite [50]. We used *lp_solve* [4] to solve the generated ILPs, and our experiments were performed on a 4-core Intel i5 CPU with 4 GB memory.

Unless stated otherwise, we assume a 2-core architecture with a 2-level cache hierarchy. Since we are focusing on the impact of shared instruction caches on the WCET, we assume a perfect data cache. We also assume a fixed cache miss penalty for every shared cache miss caused due to interferences. We ignore the effect of the shared bus, to avoid relying on any specific arbitration policy, and instead assume a fixed bus latency for every shared cache access.

For comparing the WCETs obtained using ILP-based and approximate WCIP, we have also implemented Hardy et al.’s [35] approach, which considers the effect of all interferences on all shared cache hits. We note that this technique is also used for shared cache analysis in multi-core Chronos [23]).

The code size of the Mälardalen benchmarks ranges from 0.2 KB to 60 KB, with an average size of 23.5 KB. For these benchmarks, we assume a 1 KB 4-way L1 I-cache, and a 4 KB 8-way L2 I-cache, with block size of 32 bytes. However, both *susan* and *autopilot* are fairly large (130 KB and 110 KB code size respectively) and show almost zero L2 cache hits for the above cache architecture, even without interferences. Hence, we use a 16 KB 8-way L2 cache while experimenting on these benchmarks. We assume an L1 cache miss latency of 6 cycles, and L2 cache miss latency of 30 cycles. To compute WCET of a benchmark on a 2-core architecture, we assume that the benchmark runs on one core and the benchmark *nsichneu* runs on the other core. For all benchmarks except *jjdctint*, *nsichneu* is the worst-case adversary, i.e. the benchmark which causes the maximum shared cache interference.

---


3.9.2 Results

With the above assumptions, we calculated the WCET using the three techniques, and found that WCIP gave lower WCETs for 9 of the 27 Mälardalen benchmarks, as well as both susan and autopilot as compared to Hardy et. al.’s approach. ILP-based WCIP failed to provide the final WCET for susan. The WCETs were same for the rest of the benchmarks. Figure 4.7 shows the precision improvement of WCET (in %) obtained using the two WCIP approaches over Hardy et. al.’s approach. (The precision improvement is calculated as $\frac{WCET_H - WCET_O}{WCET_H}$, where $WCET_H$ is obtained using Hardy et. al.’s approach, and $WCET_O$ is obtained using WCIP).

We note that the Mälardalen benchmarks which did not show any precision improvement all had very few L2 cache hits even without interferences. For all benchmarks, Hardy et. al.’s approach was not able to guarantee a single shared cache hit after considering the effect of interferences. The precision improvement in WCET using the ILP-based WCIP and approximate WCIP is equal for almost all benchmarks. In some benchmarks, the precision improvement using approximate WCIP is slightly higher due to the way in which the ILP-based approach counts misses due to interferences for cache hits inside loops. Specifically, multiple hit paths could be on the worst case path, and hence the same miss could be counted multiple times. The average precision improvement over 10 benchmarks for ILP-based WCIP was 27.5 %, and for approximate WCIP (over 11 benchmarks), it was 26 %. Note that over the same 10 benchmarks as ILP-based WCIP, the precision improvement of approximate WCIP was also 27.5 %.

The complexity of the ILP increases with the number of shared cache hits in the program, because the ILP-based approach essentially searches among all distributions of interferences to shared cache hits to find the maximum WCET. On the other hand, the complexity of approximate WCIP is independent of the number of cache hits. This is demonstrated by the large benchmark susan, for which the ILP-based approach fails to provide the final WCET because the solver is unable to solve the ILP (with a timeout of 24 hours). On the other hand, approximate WCIP requires only 2.5 seconds to give the final WCET, with a precision improvement of 6 % over Hardy et. al.’s approach. The analysis times for the rest of the benchmarks were similar for both ILP-based and approximate WCIP, on average 0.82 seconds (with maximum of 4.5 s). The reason that ILP-based approach fails for susan is because the number of instructions causing L2 cache hits (without interferences) in susan were 325, while the maximum number of instructions causing L2 cache hits across all the other benchmarks was 15 (for autopilot).
Figure 3.8: Precision improvement of WCET obtained using (1) ILP-based WCIP and (2) Approximate WCIP over Hardy et. al’s approach
3.9.3 Hits on the WC path and the maximum OF

To understand why approximate WCIP performs so well, we measured the impact of the two assumptions made by approximate WCIP. The first assumption is that all the shared cache hits of the program are present on the worst-case path. We measured the number of shared cache hits on the worst-case path (obtained assuming no interferences), and the total number of shared cache hits in the program. We found that among the 11 benchmarks, an average of 95.6% of the total number of shared cache hits were present on the WC-path (with minimum of 70% and maximum of 100%). This shows that the first assumption will likely not have a major impact on the precision of approximate WCIP.

The average eviction distance (across all shared cache hits) across all benchmarks was 7.6, (ranging from 6.1 to 8). The second assumption made by approximate WCIP is to remove the overlapping effect, by multiplying the original budget of interferences with the maximum Overlapping Factor (OF). We found that the maximum OF across all cache sets was 1 for 9 out of the 11 benchmarks (it was 4 for ndes and 7 for jfdejint). This shows that removing the overlapping effect does not cause a huge increase in the number of interferences.

Another advantage of approximate WCIP is that we can express the increase in WCET due to interferences as a piecewise-linear function of the number of interferences. Figure 3.9 shows the increase in WCET (as compared to WCET obtained assuming no interferences) corresponding to different number of interferences (expressed as the ratio of the number of interferences to number of cache hits in the program). While there would be a separate graph for each cache set, here we take the total number of interferences across all cache sets on the x-axis. The figure shows that the increase in WCET is less than 50% for all benchmarks if the number of interferences do not exceed the number of cache hits. Moreover, for majority of the benchmarks, the increase in WCET is within 30% even when the number of interferences are twice the number of shared cache hits.

Changing the number of cores: We also experimented with a 4-core architecture, with the same cache configurations, except that the L2 cache is now shared among all the 4 cores. We assume different instances of the same benchmark nsichneu running on 3 cores. For this architecture, approximate WCIP gave lower WCETs for 9 benchmarks over Hardy et al.’s approach, with average precision improvement of 25.5% (benchmarks qurt and sqrt did not show any improvement). Higher number of cores results in higher number of interferences to be distributed, causing more cache misses, but the increase in WCET computed using approximate WCIP continues to remain much smaller as compared to Hardy et al.’s approach.
Figure 3.9: Graph showing the relation between WCET obtained using WCIP and the amount of shared cache interference

3.9.4 Changing the cache size

We also experimented with two other L2-cache configurations, half the original size and double the original size. Figure 3.10 shows the precision improvement in WCET obtained using ILP-based and approximate WCIP, over Hardy et al.'s approach, for both the cache configurations. On halving the L2 cache, 7 (out of the original 11) benchmarks showed precision improvement. Again, both the ILP-based and approximate WCIP gave similar results with the ILP-based approach again failing to provide the final WCET for susan. The average precision improvement over the 7 benchmarks was 33 %. Note that the average precision improvement for the same 7 benchmarks for the original L2 cache was 37.6 %. Hence, the precision improvement has decreased, which is as expected, since the smaller L2 cache will result in lower number of cache hits, thus decreasing the reliance of the WCET on L2 cache analysis.

On doubling the L2 cache, all benchmarks (except sqrt and autopilot) showed higher precision improvement for WCIP, and the average precision improvement for approximate WCIP over the 11 benchmarks was 35.7 %. However, for ILP-based WCIP, the ILP solver was not able to solve the generated ILPs for 3 more benchmarks (ndes, qurt and jfdctint), in addition to susan, within 24 hours. This has happened because of the higher number of L2 cache hits, and the subsequent increase in the complexity of the ILP. On the other hand, approximate WCIP shows a precision improvement of 47 % for susan, since the number of shared cache hits have almost doubled because of the larger shared cache. For all benchmarks, approximate WCIP took less than 5 seconds to find the WCET, with high precision improvement in the 3 benchmarks for which ILP-based WCIP fails. This illustrates the advantage of using approximate WCIP over ILP-based WCIP.
Figure 3.10: Precision improvement of the WCET for two different cache configurations (a) 2 KB L2 cache (b) 8 KB L2 cache

3.9.5 Comparison with Simulated WCET

We also compared the estimated WCETs obtained using the three techniques, with the WCET obtained using simulation. To obtain the simulated WCET, we used the modified version of Simplescalar framework, used for validation in Multi-core Chronos. The modified version supports simulation of shared cache and shared bus, among other architectural components. For our purposes, we only simulated the effect of the shared cache, and assume constant shared bus delay.

As has been noted by [23], it is difficult to simulate the exact interleaving for accesses which will result in the worst-case scenario for shared caches. Hence, the simulated WCET may highly under-estimate the actual WCET of the program. It is also difficult to obtain the exact worst-case input for some of the benchmarks, such as qurt and ndes, which involve branching based on complex mathematical calculations.

We calculated the WCET estimation ratio, defined as \( \frac{\text{Estimated WCET}}{\text{Simulated WCET}} \), for 11 benchmarks. We first found the simulated and estimated WCETs assuming a private L2 cache, to determine the impact of infeasible paths, private cache analysis, etc. on the overestimation of the estimated WCET. Then, we assumed the same L2 cache shared between two cores, with the benchmark
We found that for majority of the benchmarks, the overestimation ratio in the case where the shared L2 cache is analyzed using WCIP, is almost the same as the overestimation ratio for private L2 cache. The average overestimation ratio for private L2 cache was 1.82, while for shared L2 cache analysis using ILP and approximate WCIP, it was 2.1. On the other hand, Hardy et. al.’s analysis introduces large amounts of imprecision, and the average overestimation ratio was 3.4. This shows that shared cache analysis using WCIP does not introduce large amounts of imprecision in the estimated WCETs.

### 3.9.6 Cache partitioning

Cache partitioning is a hardware-based approach to simplify shared cache analysis in multi-core architectures. In cache partitioning, each core is assigned a private portion of the shared cache, which will not be accessed by any other core. This ensures that there will be no interferences to account for during shared cache analysis, and hence private cache analysis techniques can be directly applied for the shared cache. The disadvantage is that a core will not be able to use the entire shared cache, and hence it may suffer more shared cache misses (both capacity and conflict misses).
Here, we limit our attention to fixed partitioning, and compare the WCETs obtained by using partitioned shared cache, to the WCETs obtained using WCIP in an un-partitioned shared cache. There are two ways in which fixed partitioning can be implemented: (1) Vertical partitioning, where each core is assigned a subset of ways in all cache sets, and (2) Horizontal partitioning, where each core is assigned a subset of cache sets. For our 2-core architecture, we divided the cache equally between both the cores.

For vertical partitioning experiments, we decreased the shared cache associativity from 8 to 4, while for horizontal partitioning experiments, we decreased the number of cache sets from 16 to 8. Figure 3.9.6 shows the percentage increase in the WCET obtained using approximate WCIP, horizontal cache partitioning, and assuming that all shared cache accesses as misses, as compared with WCET of the benchmarks running in isolation with an un-partitioned shared L2 cache.

The percentage increase in WCET is calculated as \( \frac{WCET_{\text{shared}} - WCET_{\text{orig}}}{WCET_{\text{orig}}} \), where \( WCET_{\text{orig}} \) is the WCET of the benchmark running on single-core architecture with the same (unpartitioned) cache hierarchy, while \( WCET_{\text{shared}} \) is obtained using either approximate WCIP or the two cache partitioning techniques or assuming that all shared cache accesses miss the cache. The point of comparing with \( WCET_{\text{orig}} \) is that the lower the percentage increase, the lower the imprecision.
introduced by shared cache analysis to the WCET.

First, note that using vertical cache partitioning does not result in any increase in the WCET for all benchmarks except \textit{jfdctint}. The reason is that most of these benchmarks do not access more than 4 cache blocks per cache set in the shared cache, and hence, their cache performance is not affected by decreasing the cache associativity to 4. \textit{jfdctint} accesses at least 6 cache blocks in every cache set, and the percentage increase in its WCET due to vertical partitioning was 28.4\%, which is greater than approximate WCIP (21.1\%).

Horizontal partitioning, on the other hand, is highly ineffective, as it introduces greater imprecision than both the other techniques for all benchmarks except \textit{ndes}. The average increase in the WCET across the 11 benchmarks for approximate WCIP is 11\%, while for horizontal partitioning, it is 49.3\%. Also, note the high increase in WCET for most benchmarks, when it is assumed that all shared cache accesses miss the cache. The average increase in WCET in this case is 80.1\%, which highlights the importance of shared cache analysis in accurate WCET estimation.

From the above results, it would seem that using normal cache analysis with vertical partitioning is more effective than using WCIP with an un-partitioned shared cache. However, note that with WCIP, the WCET also depends on amount of interference caused by programs running on other cores, and for the above experiment, we used the worst case adversary. By selecting programs which generate less shared cache interference, the WCET can be controlled using WCIP, but with cache partitioning, there will be no impact on the WCET. With vertical partitioning, we are effectively assuming that there is an adversary program which generates a constant amount of shared cache interference (in this case, 4 interferences for every shared cache access in the program under analysis). Moreover, in the instance where the number of cache blocks accessed per cache set goes beyond 4 (for \textit{jfdctint}), vertical partitioning introduces greater imprecision than WCIP.

### 3.9.7 Interaction with TDMA-based shared bus

As shown in Section 2.8, our technique can also be integrated with precise shared bus analysis techniques, which provide better bounds on the shared bus delay. To find the impact on precision of the WCET, we experimented with a shared bus architecture which uses a TDMA-based round robin arbitration policy. We assume the slot length of each core to be 50 cycles in the TDMA schedule. We use the global convergence analysis proposed in [38], to accurately bound the shared bus delay for each basic block. This analysis essentially finds all the offsets in the TDMA period that can occur at the start of a basic block, which are then used to bound the bus delay experienced by the shared cache accesses present in the basic block.
We used Hardy et. al.’s approach for shared cache analysis, in conjunction with the global convergence analysis for shared bus, to find $WCET_H$. We used the shared cache behavior in isolation, in conjunction with the global convergence analysis to find $WCET_I$. Then, we used approximate WCIP to find the maximum increase in WCET due to interferences, added this increase to $WCET_I$, along with $200(=2n_c s_l)$ cycles for each shared cache miss, to find $WCET_O$. As shown in Section 6, this is a safe over-approximation of the extra bus delay caused by the shared cache misses. We found that approximate WCIP continued to provide lower WCET estimates for all the 11 benchmarks, with average precision improvement of $WCET_O$ over $WCET_H$ being 21.4%.
Chapter 4

Private Cache Analysis

In this chapter, we present new approaches for precise analysis of the private cache, which take care of a number of precision issues with the state-of-the-art Abstract Interpretation based approach. Our work is targeted towards the analysis of the private L1 cache attached to each core (as shown in Figure 1.1 in Chapter 1). We first review all related work in the area of private cache analysis. Then, we demonstrate the precision issues with the state-of-the-art AI-based analysis using simple examples. To solve these precision issues, we formally introduce the concept of concrete and abstract cache miss paths, and also provide the AI-based approach to find abstract cache miss paths. We then use cache miss paths in conjunction with the structural information about the program, to find worst-case profiles of basic blocks. We also show how cache miss paths can be directly incorporated into the IPET ILP to find the exact cache behavior along the worst-case execution path.

4.1 Literature Survey

Abstract Interpretation (AI) based approaches ([11, 29, 36, 25]) are widely used for cache analysis, because they guarantee safety, give adequately precise results, and scale well for large programs. However, these approaches are not sufficient to precisely capture cache behavior for WCET estimation. State space explosion is a serious problem while analyzing cache behavior, due to the exponential number of cache states that can arise during the program execution, especially as the cache associativity increases. Hence, some form of abstraction is necessary to ensure scalability, resulting in a trade-off between analysis precision and efficiency. Instead of maintaining several actual cache states which may be possible at a program point and which could be exponential in number, AI-based approaches determine a single abstract cache state at every program point which safely encodes the bare minimal information that is necessary to
predict cache behavior. Such an approach is only sufficient to classify the behavior of individual cache accesses into a small number of classes. In the next section, we demonstrate various instances of cache behavior which can be safely used for WCET estimation, but which cannot be specified in terms of the hit-miss classifications determined using AI-based approaches.

Data Flow Analysis has also been used to perform cache analysis [49]. Conceptually, this approach is similar to AI-based cache analysis, as it defines a data flow analysis framework to find cache contents which may enter the cache across all executions, or cache contents which must be present in the cache across all executions. It also suffers from the same precision issues as AI-based analysis, as it uses the same classes for hit-miss classification of individual accesses.

There have been multiple efforts to use Model Checking for WCET analysis ([65, 26, 32, 47]). These approaches essentially explore the entire state space of all possible actual cache states and give precise cache analysis results, but they do not provide any bounds on the analysis time [62]. Most of these approaches have only been tested on small benchmarks, and some of them only consider the impact of processor pipeline, assuming absence of a cache.

There have also been efforts in combining cache analysis with path analysis to find the exact cache behavior along the worst case execution path (WCEP), most notably, the CSTG-based approach proposed by Lee et al. [59]. In this work, the authors first generate the cache state transition graph (CSTG), whose nodes are all possible cache states generated during execution, and edges show the transition between the cache states. Integer variables are introduced in the IPET ILP (used to find the WCEP) for all the edges in the CSTG, and these variables are then used to provide an upper bound on the number of hits experienced by accesses. Clearly, this approach can potentially introduce a very large number of variables and constraints in the ILP, and, in fact is considered non-practical even for small programs [62].

An important factor, ignored by AI-based analysis, which can have a potential impact on the precision of cache analysis is the presence of infeasible paths. These infeasible paths generally take the form of pairs of basic blocks whose execution is guarded by conditionals which can never be satisfied together. Information about infeasible paths can be obtained separately using abstract execution [31], SMT solvers [19], model checking [21], etc. and is part of the program flow analysis stage. This stage generally occurs before timing analysis, and is primarily used to determine the program CFG, loop bounds, etc. A number of works have integrated infeasible path information into the IPET formulation, ensuring that infeasible paths will be ignored while finding the worst-case path in the program ([28], [19]). However, these works have not considered the impact of infeasible paths on cache analysis.

There are few works ([21, 15]) which have used SAT-solvers to directly search for infeasible paths which can affect cache behavior. In [21], the authors instrument the code by introducing
variables to count the number of cache misses suffered by accesses, and then use SAT solvers to verify assertions on these variables. This approach requires code instrumentation, and is also known to have very high analysis time [15]. There is no way to reuse infeasible path information, which may already have been determined separately during the program flow analysis stage. Moreover, the approach will improve the precision of cache analysis only when there are actual infeasible paths (and SAT-solvers can identify them).

[15] modifies the AI-based approach for cache analysis, by annotating cache states with logic formulae, corresponding to partial paths along which the cache state would be realized. However, their work can only handle limited types of infeasible paths. In particular, they only target short infeasible paths, since their abstract lattice only keeps track of partial paths where the basic blocks must be close to each other in the program CFG (there cannot be more than one merge point in the partial path). Moreover, they also ignore the worst-case path information and consider only the impact of infeasible paths on cache analysis.

The precision loss of the AI-based approaches mostly arises due to the imprecise nature of the join of abstract cache states at merge points. This issue has been identified before, but the join of abstract cache states is also responsible for a substantial reduction in state space being searched and reduction in the analysis time. In [14], the authors make a trade-off between the precision of the join and the expressiveness of the abstract cache states, by strengthening the former but significantly weakening the latter. They formulate a separate AI-based analysis for each basic block, and instead of maintaining complete information about cache states, they only maintain information relative to the cache blocks accessed by the basic block under analysis. This allows them to avoid the loss in precision due to the join, and they find all the relative cache states possible at the start of the basic block under analysis, which are then used to find the maximum number of cache misses in the basic block.

However, there are several issues with their approach. Their approach is specifically targeted towards analysis of direct-mapped caches (whose cache associativity is 1), and does not extend well for set-associative caches with higher associativity. In particular, they assume that maximum of one cache block per cache set is accessed in every basic block. However, as the cache associativity increases, the number of cache blocks mapped to the same cache set increases, which increases the possibility of multiple cache blocks accessed in the same basic block and mapped to the same cache set. In fact, during our experiments with larger programs and higher associativity, we found that this assumption was violated for almost all basic blocks. Further, maintaining information about the cache states relative to the cache blocks accessed in the basic block under analysis is not enough for set-associative caches, since this could result in imprecise cache updates and over-estimate the number of cache misses. Finally, their approach
is neither capable of using information about infeasible paths or worst case execution path, nor capable of refining cache behavior prediction of accesses inside loops.

4.2 Examples

In this section, we illustrate the precision issues with AI-based cache analysis, using several examples. Must analysis is used to find cache blocks which are guaranteed to be in the cache across all execution instances, so that accesses to these cache blocks can be classified as Always-Hit. However, it can actually miss such scenarios for some programs. For example, consider Figure 4.1 which shows a portion of program CFG and the instruction cache accesses therein. $m_1, m_2$ indicate cache blocks, and also the access to those cache blocks. Assume that they map to the same cache set and the cache associativity is 2. The figure also shows the abstract Must cache states at various program points (in dotted lines, with more recently accessed cache blocks towards the left). As mentioned earlier, Must cache analysis computes, at a program point, all those cache blocks which are guaranteed to be present in the actual cache at that program point. It also maintains an upper bound on the ‘age’ of a cache block, which is simply the number of cache blocks more recently accessed. For a given access, if the accessed cache block is present in the Must cache, then it is classified as Always-Hit.

Consider the Must cache states at the end of basic blocks $v_1, v_2$, and the result of their join at the start of $v_3$. The access to $m_2$ in $v_2$ increases the age of $m_1$, but also brings $m_2$ in the cache. However, since join in Must analysis at a merge point selects only those cache blocks which are present in the must caches at the end of all predecessor basic blocks and also takes
Figure 4.2: Precision issue with Persistence Analysis

their maximum age, \( m_2 \) will not be present in the Must cache after join, but its effect on the age of \( m_1 \) will be retained. As a result, the access to \( m_2 \) in \( v_3 \) will evict \( m_1 \) out of the Must cache, due to which the access to \( m_1 \) in \( v_1 \) will not be classified as Always-Hit. However, it can be clearly seen that \( m_1 \) is guaranteed to be present in the cache at the start of \( v_1 \), and Must analysis is incorrectly adding the aging effect of accesses to the same cache block (\( m_2 \)) to another cache block (\( m_1 \)) multiple times.

Note that real programs frequently exhibit such behavior. For example, the last instruction of \( v_1 \) could be a conditional jump, with \( v_2 \) being the fall-through basic block and \( v_3 \) the target of the jump. In such a scenario, the instructions towards the end of \( v_2 \) and beginning of \( v_3 \) would be contiguous in the address space, and could map to the same cache block. While persistence analysis would be able to identify the access to \( m_1 \) as persistent, this classification will still result in one miss every time the loop is entered, which could be substantially high for deeply nested loops.

For instruction caches, Persistence analysis is often more effective than Must analysis, because it identifies those cache blocks which are never evicted (within a fixed scope), and cache accesses inside loops frequently have this property. Such accesses will cause at most one cache miss for every entry to the scope in which they have been classified as persistent. The safe version of Persistence analysis ([36, 25]) determines, for every cache block \( m \), the maximal set of younger cache blocks that may have been accessed since the last access to \( m \) (within a specific scope). If the cardinality of this set is less than the cache associativity, then \( m \) is declared as persistent. Hence, for the example of Figure 4.1, \( m_1 \) would be declared as persistent. However, there are other precision issues with Persistence analysis, and it may also miss cache accesses which are actually persistent.
Consider the program CFG shown in Figure 4.2. Assume that cache blocks $m_1, m_2, m_3$ map to the same cache set, and the cache associativity is 2. In this example, $m_1$ and $m_2$ are not persistent, but $m_3$ is persistent. However, the set of younger cache blocks of $m_3$ would contain both $m_1$ and $m_2$, and hence Persistence analysis would not be able to identify $m_3$ as persistent.

Further, classifications such as Always Hit and Persistent are not enough to capture precise cache behavior. These classifications apply to individual accesses, but often, a more precise prediction can be made about a group of accesses. For example, consider the program CFG shown in Figure 4.3. Assume that cache blocks $m_1, m_3$ map to cache set $s_1$, while $m_2, m_4$ map to cache set $s_2$ and the cache associativity is 1. We focus on the accesses to $m_3, m_4$ in basic block $v_3$. Note that neither of these accesses can be classified as Always Hit or Persistent. $m_1$ in $v_1$ will evict $m_3$, while $m_2$ in $v_2$ will evict $m_4$. However, both these evictions cannot happen simultaneously in the same iteration. In other words, at least one cache hit in $v_3$ is guaranteed to occur in every iteration (except possibly the first).

Accesses inside loops may not exhibit the worst-case behavior in every iteration, but only in a subset of the iteration space. Consider the program CFG in Figure 4.4. Assume that $m_1$ and $m_2$ map to the same cache set, and the cache associativity is 1. Again, neither of the accesses to $m_1$ and $m_2$ can be classified as Always-Hit or Persistent. However, for $m_1$ to miss the cache, basic block $v_2$ must have been executed in the previous iteration, and the same is true for $m_2$ and $v_1$. In other words, $m_1$ (or $m_2$) cannot cause a cache miss in every iteration of the loop, and the maximum number of misses caused by either is equal to half the total number of iterations. Every iteration can still result in one cache miss, but for that the execution must alternate between the two branches across iterations.
Figure 4.4: Example illustrating frequency of worst-case behavior for accesses in loops

Figure 4.5: Example illustrating impact of Worst case execution path on cache analysis
Finally, in all of the above examples, our aim has been to find the worst-case cache behavior that nonetheless holds across all execution instances. That is, the actual number of cache misses across all execution instances will always be less than number of misses for the worst-case behavior. However, knowledge about the worst case execution path can be used to find the exact cache behavior which only holds in the worst-case execution instance. This is the most precise prediction for cache behavior that can potentially be made, for WCET estimation, and there are instances where it will be strictly more precise than any analysis which does not use WCEP information (viz. any analysis that can handle all the precision issues described above). For example, consider the program CFG shown in Figure 4.5. Along with the cache accesses made by basic blocks $v_1, v_2, v_3$, we also know the maximum execution time (i.e. WCET) of the basic blocks $v_1$ and $v_2$ to be 100 and 1000 cycles respectively. Thus, the WCEP will pass through $v_2$. Assume that $m_1, m_1'$ map to cache set $s_1$, $m_2, m_2'$ map to cache set $s_2$ and $m_3, m_3'$ map to cache set $s_3$, and the cache associativity is 1. We focus on the basic block $v_3$. It is clear that even though the worst-case cache behavior of $v_3$ is 2 cache misses in every iteration (accesses to $m_1$ and $m_2$), execution along the WCEP will only cause 1 cache miss (to $m_3$) in every iteration (except possibly the first).

However, note that the estimation of the WCEP itself depends on the predicted cache behavior. In the above example, even though the WCET of $v_2$ is greater than $v_1$, execution of $v_1$ is also responsible for more cache misses, and if the latency of the extra cache misses caused by $v_1$ in combination with the WCET of $v_1$ is greater than the combined latency of the misses caused by $v_2$ and its WCET, then the WCEP would pass through $v_1$. Hence, determination of the WCEP cannot be carried out independently of the cache analysis, if one wants to use WCEP information to improve the prediction of cache analysis.

### 4.3 Cache miss paths

In order to detect the behaviors illustrated in the previous section for set-associative caches, we propose the concept of cache miss paths. The main insight is that since caches are expected to take advantage of temporal locality of accesses, if an access hits the cache, it is most likely that the same cache block has been accessed recently in the past. Conversely, it should be possible to predict that an access misses the cache by observing only a small portion of the most recent accesses. Hence, given an access, we analyze a small portion of the program which can be executed before the access in the backward direction with complete precision, keeping track of all accesses across individual paths. The hope is that only a small number of paths leading to an access need to be differentiated to determine whether it will hit or miss the cache.

In this section, we give a formal definition of concrete and abstract cache miss paths and
also show that abstract cache miss paths are guaranteed to cover all possible concrete miss paths. In the next sections (4.4 and 4.5) we will show how they can be used for precise cache analysis. For simplicity, in the rest of the chapter, we focus only on the first level instruction cache. Let $G = (V, E)$ be the control flow graph (CFG) of the program. $V$ is the set of basic blocks in the program, and the edges in $E \subseteq V \times V$ denote the control flow among them. Let $v_{\text{start}}$ be the unique start basic block. Let $\mathcal{B}$ be the set of all cache blocks accessed by the program, and $\mathcal{S}$ be the set of all cache sets. Let $\text{Set} : \mathcal{B} \rightarrow \mathcal{S}$ map every cache block to its cache set. Assume that the cache associativity is $k$ and the replacement policy is LRU.

Let the function $\text{Acc} : V \times \mathcal{S} \rightarrow \mathcal{P}(\mathcal{B})$ give the set of cache blocks mapped to the given cache set accessed by the given basic block in the program. We also define the functions $\text{Acc}_a : V \times \mathcal{B} \rightarrow \mathcal{P}(\mathcal{B})$ and $\text{Acc}_b : V \times \mathcal{B} \rightarrow \mathcal{P}(\mathcal{B})$. $\text{Acc}_a(v, m)$ and $\text{Acc}_b(v, m)$ give the set of cache blocks mapped to the cache set of $m$, and accessed by $v$ after the last access and before the first access to $m$ in $v$, respectively. If $m$ is not accessed in $v$, then they return $\text{Acc}(v, \text{Set}(m))$. Note that if a cache block $m$ is accessed multiple times in a basic block $v$, then we will only focus on the first access to $m$ in $v$, since the cache behavior of the rest of the accesses to $m$ in $v$ will remain the same in all execution instances and can be easily determined (by AI-based Must analysis).

A walk $\sigma$ in $G$ is a sequence of basic blocks $v_1 v_2 \ldots v_p$ such that $(v_i, v_{i+1}) \in E$, for all $i$, $1 \leq i \leq p - 1$ (note that repetition of basic blocks is allowed). We lift the $\text{Acc}$ function to $\sigma$ in a straightforward manner to give the total set of cache blocks mapped to a given cache set in the entire walk. We use the notation $|S|$ to mean the number of elements in the set $S$. We only concentrate on those accesses which are not classified as Always Hit by Must cache analysis.

**Definition 1.** A concrete cache miss path of an access to $m \in \mathcal{B}$, mapped to $s \in \mathcal{S}$, in basic block $v$ is defined as a walk $\sigma = v_1 v_2 \ldots v_p v$ in the CFG $G$ with the following properties:

1. $m \notin \bigcup_{i=2}^{p} \text{Acc}(v_i, s)$,

2. $(|\text{Acc}_a(v_1, m) \cup \bigcup_{i=2}^{p} \text{Acc}(v_i, s) \cup \text{Acc}_b(v, m)| \geq k) \lor (v_1 = v_{\text{start}} \land m \notin \text{Acc}(v_{\text{start}}, s))$, and

3. $|\bigcup_{i=2}^{p} \text{Acc}(v_i, s) \cup \text{Acc}_b(v, m)| < k$

A concrete cache miss path of access $r$ is a walk in the $G$ along which $r$ will suffer a cache miss, and also has the property that no suffix of the walk will be a concrete cache miss path. For LRU caches, an access suffers a cache miss only if either the accessed cache block has not been brought into the cache since the start of the program, or at least $k$ distinct cache blocks have
been accessed since the last access to the same cache block. Concrete cache miss paths thus provide a necessary and sufficient condition for cache misses. However, they can be arbitrarily large, and hence we abstract them in two ways: we allow “gaps” in the miss paths and include only those basic blocks which are absolutely necessary, and we allow only a bounded number of such basic blocks to occur in the miss paths. Let $T$ be the maximum allowable miss path length.

Given a concrete cache miss path $\sigma = v_1v_2 \ldots v_pv$ of access to $m$ in $v$, let $\alpha_{v,m}(\sigma) = \{v_i : Acc(v_i, Set(m)) \neq \phi\}$. We define $\alpha^T_{v,m}(\sigma) = \alpha_{v,m}(\sigma)$ if $|\alpha_{v,m}(\sigma)| \leq T$, otherwise if $j$ is the largest subscript in $\sigma$ (in other words, $v_jv_{j+1} \ldots v_pv$ is the smallest suffix of $\sigma$) such that $|\alpha_{v,m}(v_jv_{j+1} \ldots v_pv)| = T$, then $\alpha^T_{v,m}(\sigma) = \alpha_{v,m}(v_j \ldots v_pv)$. 

Definition 2. Given a concrete cache miss path $\sigma$ of access to $m$ in $v$, $\alpha^T_{v,m}(\sigma)$ is called an abstract cache miss path of access to $m$ in $v$.

In the abstract cache miss path of an access to $m$, we only maintain information about those basic blocks in a concrete cache miss path which actually access the cache set of $m$. In the example in Figure 4.3, $v1v3$ is a concrete cache miss path of the access to $m3$, and if $T = 2$, then $\alpha^T_{v3,m3}(v1v3) = \{v1, v3\}$ is its abstract cache miss path.

We use Abstract Interpretation to find the abstract cache miss paths of an access. For simplicity, we now provide a description of the approach to find all the abstract cache miss paths of a single access to $m$ in basic block $v$, mapped to cache set $s$. The method can be easily extended to find the miss paths of all accesses in all basic blocks simultaneously. The abstract lattice is $\mathcal{L} = (\mathbb{P}(\mathbb{P}(V)), \subseteq)$. Each element is a set of possible abstract cache miss paths. The analysis is carried out in the backward direction in the CFG. For every basic block $w$, the approach maintains an IN element $IN_w \in \mathcal{L}$ at the end of the basic block, and OUT element $OUT_w \in \mathcal{L}$ at the beginning of the basic block. Every basic block $w$ is also associated with a transfer function $f_w : \mathcal{L} \rightarrow \mathcal{L}$ such that $OUT_w = f_w(IN_w)$. Also, $IN_w = \bigcup_{(w,u) \in E} OUT_u$.

We define the function $DB_{v,m} : \mathbb{P}(V) \rightarrow \mathbb{N}$ as $DB_{v,m}(\pi) = |\bigcup_{w\in\pi} Acc_a(w, m) \cup Acc_b(v, m)|$, to count the number of distinct cache blocks accessed in the basic blocks of $\pi$ and mapped to the cache set of $m$. We now describe the transfer function $f_w$ separately for different cases. Note that $P \in \mathcal{L}$.

Case - 1 : $w \neq v$, $Acc(w, s) = \phi$

$$f_w(P) = P$$
Case - 2: $w \neq v$, $m \notin \text{Acc}(w, s)$ and $\text{Acc}(w, s) \neq \phi$

\[
 f_w(P) = \{ \pi \in P : DB_{v,m}(\pi \setminus \{v\}) \geq k \lor |\pi| = T \}
\]
\[
\cup \{ \pi \cup \{w\} : \pi \in P \land DB_{v,m}(\pi \setminus \{v\}) < k \land |\pi| < T \}
\]

Case - 3: $w \neq v$, $m \in \text{Acc}(w, s)$

\[
 f_w(P) = \{ \pi \in P : DB_{v,m}(\pi \setminus \{v\}) \geq k \lor |\pi| = T \}
\]
\[
\cup \{ \pi \cup \{w\} : \pi \in P \land DB_{v,m}(\pi \setminus \{v\}) < k \land |\pi| < T \}
\]
\[
\land DB_{v,m}((\pi \cup \{w\}) \setminus \{v\}) \geq k \}
\]

Case - 4: $w = v$

\[
 f_v(P) = \{ \{v\} \cup \{ \pi \in P : DB_{v,m}(\pi) \geq k \lor |\pi| = T \}
\]

Case - 5: $w = v_{\text{start}}$

\[
 f_{v_{\text{start}}}(P) = \{ \pi \in P : DB_{v,m}(\pi \setminus \{v\}) \geq k \lor |\pi| = T \}
\]
\[
\cup \{ \pi \cup \{v_{\text{start}}\} : \pi \in P \land DB_{v,m}(\pi \setminus \{v\}) < k \land |\pi| < T \}
\]

An incomplete miss path is a set of basic blocks $\pi$ such that $|\pi| < T$ and $DB_{v,m}(\pi \setminus \{v\}) < k$.

The transfer function does not change the incoming state if the basic block does not access the cache set $s$ (Case-1). If the basic block $w$ does not access $m$, but accesses some other cache block mapped to $s$, then $w$ is simply added to those miss paths which are incomplete, in addition to retaining completed miss paths (Case-2). If the basic block $w$ does access $m$, then only those miss paths which are either already complete, or are completed–due to accesses to other cache blocks mapped to $s$ after the access to $m$ in $w$–are retained, in the latter case after adding $w$ (Case-3). The same scenario occurs for the basic block $v$ itself, and we also add the incomplete path $\{v\}$ to begin the collection of miss paths (Case-4). Finally, if an incomplete miss path reaches the start basic block, then it is completed (Case-5). We start the analysis by assigning $IN_w$ for all $w$ to $\phi$.

Table 4.1 shows the $IN$ and $OUT$ values of basic blocks across different iterations of the fix-point loop, on applying the proposed miss path analysis in the example of Figure 4.1. Here, we perform the analysis with respect to the access to $m_1$ in basic block $v_1$, assuming that $T = \infty$. Note that the associativity $k = 2$. We start with all $IN$ values being empty for every basic block. In the first iteration, only Case 4 of the transfer function applies for basic block $v_1$, resulting in an incomplete miss path $\{v_1\}$ added to $OUT_{v_1}$. Note that since the analysis
is carried out in backward direction, the OUT values correspond to the beginning of a basic block, while the IN value correspond to the end of a basic block. In iteration 2, since $v_3$ is a predecessor of $v_1$, the miss path $\{v_1\}$ is propagated to $IN_{v_3}$.

In iteration 3, Case 2 of the transfer function applies to $v_3$. Since $DB_{v_1,m_1}(\{v_1\} \setminus \{v_1\}) = 0$, $v_3$ is added to the incomplete miss path $\{v_1\}$. In iteration 4, the miss path $\{v_1, v_3\}$ is propagated to the IN values of both $v_1$ and $v_2$. In iteration 5, Case 4 of the transfer function applies for basic block $v_1$. Since $DB_{v_1,m_1}(\{v_1, v_3\}) = 1$, this incomplete miss path will be removed from consideration, resulting in an unchanged OUT$_{v_1}$. In the same iteration, Case 2 of the transfer function applies to $v_2$, and since $DB_{v_1,m_1}(\{v_1, v_3\} \setminus \{v_1\}) = 1$, $v_2$ will be added to the incomplete miss path $\{v_1, v_2\}$. In iteration 6, the miss path $\{v_1, v_2, v_3\}$ is propagated from OUT$_{v_2}$ to IN$_{v_1}$. In iteration 7, Case 4 applies to $v_1$, but since $DB_{v_1,m_1}(\{v_1, v_2, v_3\}) = 1$, this incomplete miss path will also be removed from consideration, resulting in an unchanged OUT$_{v_1}$. Since all the IN and OUT values have remained unchanged, no more updates will be performed. Finally, since OUT$_{v_1} = \{v_1\}$, this is the only incomplete miss path that will be propagated outside the loop to IN$_{v_4}$. Case 3 of the transfer function applies for basic block $v_4$, and since $DB_{v_1,m_1}(\{v_1\} \setminus \{v_1\}) = DB_{v_1,m_1}(\{v_1, v_4\} \setminus \{v_1\}) = 0$, OUT$_{v_4}$ will be empty. Finally, we can conclude that the access to $m_1$ in $v_1$ has no abstract cache miss paths.

We now prove that the AI-based approach determines abstract miss paths for all concrete miss paths of $m$, i.e. $OUT_{v_{\text{start}}} = \{\alpha^T_{v,m}(\sigma) : \sigma \text{ is a concrete miss path of } m\}$. A function $f : \mathcal{L} \rightarrow \mathcal{L}$ is called distributive, if given $L \subseteq \mathcal{L}$, $f(\bigcup_{P \in L} P) = \bigcup_{P \in L} f(P)$.

**Lemma 3.** The transfer function $f_w$ is distributive for all basic blocks $w$.

**Proof.** Since the transfer function $f_w(P)$ (for all cases) operates individually on every $\pi \in P$, $f_w(P) = \bigcup_{\pi \in P} f_w(\{\pi\})$. Given $L \subseteq \mathcal{L}$,
\[
\bigcup_{P \in L} f_w(P) = \bigcup_{P \in L} \bigcup_{\pi \in P} f_w(\{\pi\}) \\
= \bigcup_{\pi \in \bigcup_{P \in L} P} f_w(\{\pi\}) \\
= f_w\left(\bigcup_{P \in L} P\right)
\]

\[\square\]

It is known that in an AI framework, if the individual transfer functions are distributive, then the abstract fix-point value \(\text{OUT}_w\) is equal to the join over all paths (JOP) of all abstract values possible at the start of \(w\). Let \(w_{\text{end}}\) be the unique end basic block (i.e. \(\not\in w\), such that \((w_{\text{end}}, w) \in E\)). Given a walk \(\sigma = v_1v_2 \ldots v_p\), let \(f_\sigma = f_{v_1} \circ f_{v_2} \circ \ldots \circ f_{v_p}\) be the cumulative transfer function of \(\sigma\) (in reverse direction). For a basic block \(w\), let \(\Sigma_w\) be the set of all walks in \(G\) from \(w\) to \(w_{\text{end}}\).

**Lemma 4.** For all basic blocks \(w\), \(\text{OUT}_w = \bigcup_{\sigma \in \Sigma_w} f_w(f_\sigma(\phi))\).

*Proof.* \(\bigcup_{\sigma \in \Sigma_w} f_\sigma(\phi)\) is the (backward) JOP over all paths from \(w\) to \(w_{\text{end}}\), and since the transfer functions are distributive, this will be equal to \(\text{OUT}_w\) computed using fix-point-based (backward) analysis. \[\square\]

**Lemma 5.** Given a concrete cache miss path \(\sigma = v_1v_2 \ldots v_p v\) of access to \(m\) in \(v\), \(\alpha_{v,m}^T(\sigma) \in f_{v_1} \circ f_{v_2} \circ \ldots \circ f_{v_p}(\{v\})\).

*Proof.* Consider the case when \(|\text{Acc}_a(v_1, m) \cup \bigcup_{i=2}^p \text{Acc}(v_i, s) \cup \text{Acc}_a(v, m)| \geq k\). Also, suppose \(|\alpha_{v,m}^T(\sigma)| \leq T\). We will show that for all \(i, 1 \leq i \leq p\), \(\exists \pi \in f_{v_1} \circ f_{v_2} \circ \ldots \circ f_{v_p}(\{v\})\) such that \(\alpha_{v,m}^T(v_i \ldots v_p v) = \pi\). We show this using induction on \(p - i\). For \(p - i = 0\), i.e. for \(f_{v_p}\) only Cases 1 and 2 of the transfer function will apply. If \(\text{Acc}(v_p, s) = \phi\), then \(\alpha_{v,m}^T(v_pv) = \{v\}\), hence the statement trivially holds. If \(\text{Acc}(v_p, s) \neq \phi\), then \(\alpha_{v,m}^T(v_pv) = \{v, v\}\), but then Case 2 applies and \(v_p\) will be added to \(\pi = \{v\}\).

Now, assume the inductive hypothesis holds for some \(p - i\). We want to show the result for \(p - (i - 1)\). If \(i > 1\), then again only Cases 1 and 2 apply. If \(\text{Acc}(v_{i-1}, s) = \phi\), then \(\alpha_{v,m}^T(v_{i-1} \ldots v_pv) = \alpha_{v,m}^T(v_{i} \ldots v_pv)\). Also, \(f_{v_{i-1}} \circ f_{v_i} \circ \ldots \circ f_{v_p}(\{v\}) = f_{v_i} \circ \ldots \circ f_{v_p}(\{v\})\) (by Case 1). Hence, by the inductive hypothesis, \(\exists \pi \in f_{v_{i-1}} \circ f_{v_i} \circ \ldots \circ f_{v_p}(\{v\})\) such that \(\alpha_{v,m}^T(v_{i-1} \ldots v_pv) = \pi\).

If \(\text{Acc}(v_{i-1}, s) \neq \phi\), then \(\alpha_{v,m}^T(v_{i-1} \ldots v_pv) = \{v_{i-1}\} \cup \alpha_{v,m}^T(v_{i} \ldots v_pv)\). However, \(v_{i-1}\) will also be added \(\pi\) in \(f_{v_i} \circ \ldots \circ f_{v_p}(\{v\})\) (by Case 2).
Finally consider the case when \( i = 1 \). Now, only Cases 3 and 4 apply. If \( v_1 = v \), then 
\[
\alpha_{v,m}^T(v_1v_2 \ldots v_pv) = \alpha_{v,m}^T(v_2 \ldots v_pv).
\]
Hence, by inductive hypothesis, \( \exists \pi \in f_{v_2} \circ \ldots \circ f_{v_p}(\{v\}) \)
such that \( \alpha_{v,m}^T(v_2 \ldots v_pv) = \pi \). Case 3 applies and since \( DB_{v,m}(\alpha_{v,m}^T(\sigma)) \geq k, \pi \in f_{v_1}(\{\pi\}) \). If 
\( v_1 \neq v \), then since no suffix of the concrete cache miss path is also a concrete cache miss path, \( v_1 \) will be added to \( \pi \) by the transfer function \( f_{v_1} \) (Case 4). This completes the proof for the case when \( |\alpha(\sigma)| \leq T \) and \( v_1 \neq v_{\text{start}} \). The proof for the two remaining cases (i.e. \( v_1 = v_{\text{start}} \) and \( |\alpha(\sigma)| = T \)) will be similar.

**Theorem 2.** For every concrete cache miss path \( \sigma \) of access \( r \) in basic block \( v \), there exists an abstract cache miss path \( \pi \in \text{OUT}_{v_{\text{start}}} \) such that \( \pi = \alpha_{v,m}^T(\sigma) \).

**Proof.** Let \( \sigma = v_1 \ldots v_pv \). Let \( \sigma_e \) be a walk in \( G \) from \( v \) to \( w_{\text{end}} \) which does not pass through \( v \). Then \( f_{\sigma_e}(\phi) = \phi \). By Lemma 5, \( \alpha_{v,m}^T(\sigma) \in f_{v_1} \circ f_{v_2} \circ \ldots \circ f_{v_p}(f_{\sigma_e}(\phi)) = f_\sigma(\phi) \). If \( v_1 = v_{\text{start}} \), then \( \sigma_e \) is a walk from \( v_{\text{start}} \) to \( w_{\text{end}} \), and hence, by Lemma 4, \( f_{\sigma \sigma_e}(\phi) \in \text{OUT}_{v_{\text{start}}} \).

If \( v_1 \neq v_{\text{start}} \), then let \( \sigma_s \) be a walk from \( v_{\text{start}} \) to \( v_1 \). Now, either \( DB_{v,m}(\alpha_{v,m}^T(\sigma)) \geq k \) or \( |\alpha_{v,m}^T(\sigma)| = T \), and hence, for all \( w \) in \( \sigma_s \), \( f_w(\{\alpha_{v,m}^T(\sigma)\}) = \{\alpha_{v,m}^T(\sigma)\} \). Hence, \( \alpha_{v,m}^T(\sigma) \in f_{\sigma_\sigma \sigma_e}(\phi) \). Again by Lemma 4, this means that \( \alpha_{v,m}^T(\sigma) \in \text{OUT}_{v_{\text{start}}} \).

Since the length of an abstract cache miss path is at most \( T \), the maximum number of cache miss paths of an access is \( O(|V|^T) \). Hence, the maximum length of an ascending chain in the abstract lattice \( L \) would be \( O(|V|^T) \). The fix-point based approach has a complexity of \( O(|V|^2) \) times the maximum length of any ascending chain in the lattice, which yields a final complexity of \( O(|V|^{T+2}) \) of the above approach.

### 4.4 Algorithmic Approach

We now show how cache miss paths can be used to tackle the various precision issues discussed in Section 4.2.

#### 4.4.1 The precision issue with Must analysis

The following simple theorem shows that lack of abstract cache miss paths is a sufficient condition for Always-Hit accesses.

**Theorem 3.** If an access to \( m \) in \( v \) does not have any abstract cache miss paths, then it is guaranteed to cause a cache hit.

**Proof.** By Theorem 2, if \( m \) does not have any abstract cache miss paths, then it also does not have any concrete cache miss paths. This implies that it can never cause a cache miss. \( \square \)
In the example in Figure 4.1, the access to \( m_1 \) in \( v_1 \) does not have any cache miss paths, and hence we can conclude that the access is guaranteed to hit the cache.

### 4.4.2 The precision issue with Persistence analysis

Cache miss paths can be used to find persistent accesses within a static scope. We assume that the program CFG is reducible, which means that every loop has a unique entry and exit basic block. We say that a miss path is completely inside loop \( L \), if every basic block of the miss path is either inside \( L \) or an inner loop of \( L \). An access is said to be persistent in a loop \( L \), if it can cause at most one cache miss every time execution enters \( L \) from outside.

**Theorem 4.** If an access to \( m \) in \( v \) does not have any abstract cache miss paths which are completely inside an enclosing loop \( L \), then \( m \) is persistent in loop \( L \).

**Proof.** Every abstract miss path of \( m \) must contain a basic block which is outside \( L \). This implies that \( m \) cannot have a concrete cache miss path completely inside \( L \). Hence, \( m \) can cause at most one cache miss, for every entry to the loop \( L \) from outside the loop. \( \square \)

In general, if \( m \) is persistent in loop \( L \), \( L' \) is the parent loop of \( L \) (i.e. \( L \) is immediately nested inside \( L' \)), and \( B_{L'} \) is the maximum execution count of \( L' \), then the maximum number of cache misses caused by \( m \) would be \( B_{L'} \). We use the following strategy to perform scope-aware persistence analysis using miss paths: for an access to \( m \) inside \( v \), if \( L \) is the inner-most loop which completely contains an abstract cache miss path of \( v \), and if \( B_L \) is the maximum execution count of loop \( L \), then \( m \) can cause at most \( B_L \) cache misses. In the example of Figure 4.2, both \( m_1 \) and \( m_2 \) have miss paths inside the loop (\( \{v_2,v_3\} \) and \( \{v_1,v_3\} \) resp.), but \( m_3 \) does not have a miss path inside the loop, and hence is persistent.

### 4.4.3 Finding maximum number of cache misses in a basic block

Cache miss paths can be used to reason about the worst-case behavior of a group of cache accesses, where the individual accesses themselves might not always hit the cache or be persistent. Here, we focus only on those accesses which are classified as neither Always-Hit or Persistent and are present inside the same basic block. Our approach is based on finding cache miss paths of such accesses which can never be executed together.

**Definition 3.** Given cache accesses \( r_1 \) and \( r_2 \) in basic block \( v \), and their miss paths \( \pi_1 \) and \( \pi_2 \) respectively, we say that \( \pi_1 \) and \( \pi_2 \) do not conflict with each other if there exists a walk \( \sigma = v_1 \ldots v_p v \) in \( G \) such that \( \forall i, v \neq v_i \) and \( \pi_1 \cup \pi_2 \subseteq \{v_1, \ldots, v_p\} \). If such a walk does not exist, then we say that \( \pi_1 \) and \( \pi_2 \) conflict with each other.
If two miss paths conflict, then they cannot cause cache misses together. In the example in Figure 4.3, miss path \( \pi_1(= \{v1\}) \) of \( m3 \) and \( \pi_2(= \{v2\}) \) of \( m4 \) conflict with each other, and hence cannot cause cache misses together. The following results provide the necessary and sufficient conditions required to automatically find such miss paths.

**Lemma 6.** Given a set of basic blocks \( W = \{v_1, \ldots, v_n\} \) and basic block \( v \ (v \not\in W) \), if \( \forall v_i, v_j \in W \), there exists a walk in \( G \) either from \( v_i \) to \( v_j \) or \( v_j \) to \( v_i \) which does not pass through \( v \), then there exists a walk in \( G \) which contains all the basic blocks in \( W \) and also does not pass through \( v \).

**Proof.** We use induction on the size of the set \( W \). If the size is 1, then the statement is trivial. Suppose the result holds when the size is \( k \). Let \( W = \{v_1, \ldots, v_k, v_{k+1}\} \). By inductive hypothesis, assume that there exists a walk \( \sigma \) in \( G \) which contains all basic blocks from \( v_1 \) to \( v_k \) (in increasing order). We know that \( \forall i \), there exists a walk in \( G \) either from \( v_{k+1} \) to \( v_i \) or \( v_i \) to \( v_{k+1} \) which does not pass through \( v \). Let \( j \) be the maximum subscript such that there is a walk from \( v_j \) to \( v_{k+1} \). Now consider the sub-walk of \( \sigma \) from \( v_1 \) to \( v_j \), followed by the walk from \( v_j \) to \( v_{k+1} \), followed by the walk from \( v_{k+1} \) to \( v_{j+1} \), followed by the sub-walk of \( \sigma \) from \( v_{j+1} \) to \( v_k \). This is a walk in \( G \) which contains all basic blocks from \( W \) and does not pass through \( v \). This proves the result.

**Lemma 7.** Miss paths \( \pi_1 \) and \( \pi_2 \) of two accesses in \( v \) do not conflict \( \iff \) \( \forall w_1 \in \pi_1, \forall w_2 \in \pi_2 \), there exists a walk in \( G \) either from \( w_1 \) to \( w_2 \) or from \( w_2 \) to \( w_1 \) which does not pass through \( v \).

**Proof.** The forward direction is trivial, since we can take the required sub-walk from the walk \( \sigma \) which contains all basic blocks of \( \pi_1 \) and \( \pi_2 \). For the reverse direction, we simply take \( W = \pi_1 \cup \pi_2 \), and apply Lemma 6, which implies that there is walk in \( G \) which contains all the basic blocks of \( \pi_1 \) and \( \pi_2 \) and does not pass through \( v \). Note that by definition of miss paths, there always exists a walk between two basic blocks of the same miss path which does not pass through \( v \), and there is walk in \( G \) from every basic block in the miss path to \( v \). This shows that \( \pi_1 \) and \( \pi_2 \) do not conflict with each other.

Hence, to find whether two miss paths conflict with each other, we need to determine whether there is a walk between every pair of basic blocks from the miss paths which does not pass through \( v \). A simple way to do this is to formulate a Data-flow Analysis (DFA) \([10]\). For basic block \( v \), the DFA \( D_v \) determines, for all other basic blocks \( w \) in the program, the set of basic blocks \( IN_w \), such that there exists a walk in \( G \) from every basic block in \( IN_w \) to \( w \) which does not pass through \( v \).
Table 4.2: Data flow analysis $D_v$ to determine conflict information for basic block $v$

For $D_v$, the data-flow domain is $D = V$, the set of all basic blocks in the program. The GEN and KILL sets for all basic blocks are given in Table 4.2. The DFA fix-point algorithm calculates the $IN_w$ and $OUT_w$ sets for all basic blocks $w$, which obey the following equations:

$$IN_w = \bigcup_{u: (u, w) \in E} OUT_u$$

$$OUT_w = GEN_w \cup (IN_w \setminus KILL_w)$$

At the end of the analysis, $IN_w$ will contain all basic blocks who have a walk in $G$ to $w$, which does not pass through $v$. The correctness of the DFA is easy to see, because if there is a walk from $v_1$ to $v_2$ which does not pass through $v$, then $v_1$ will flow to the set $IN_{v_2}$ from $OUT_{v_1}$ through the edges of this walk in $G$.

**Lemma 8.** Given miss paths $\pi_1$ and $\pi_2$ of two accesses in $v$, $\pi_1$ and $\pi_2$ do not conflict $\iff$ $\forall w_1 \forall w_2 \in \pi_1 \cup \pi_2, (w_1 \in IN_{w_2} \lor w_2 \in IN_{w_1})$.

**Proof.** By Lemma 7 and the correctness of the DFA $D_v$. □

**Lemma 9.** Given miss paths $\pi_1$, $\ldots$, $\pi_n$, of accesses in $v$, there exists a walk in $G$ which contains all the miss paths and contains $v$ at the end if and only if there is no pairwise conflict in the set $\{\pi_1, \ldots, \pi_n\}$.

**Proof.** The forward direction is trivial, because if there exists a walk which contains every basic block of all miss paths, then it will contain a walk between every pair of basic blocks which does not pass through $v$, and hence none of the miss paths will conflict with each other. For the reverse direction, we take $W = \bigcup_{i=1}^n \pi_i$. Since there is no pairwise conflict between the miss paths, by Lemma 7, there exists a walk between $v_i$ and $v_j$ which does not pass through $v$, $\forall v_i, v_j \in W$. By Lemma 6, this means that there exists a walk in $G$ which contains all the basic blocks of $W$ and does not pass through $v$. □

To find the maximum number of miss paths which do not pairwise conflict with each other, we create the Miss Path Conflict Graph (MPCG). For basic block $v$, let $R_v = \{r_1, r_2, \ldots, r_n\}$ be the set of accesses in $v$ which are neither persistent nor always hit the cache. In the MPCG $G_M$, each vertex represents a miss path of an access in $R_v$. An edge is added between miss path $\pi_i$ of access $r_i$ and miss path $\pi_j$ of access $r_j$ if $r_i \neq r_j$ and $\pi_i$ and $\pi_j$ do not conflict with each other.
Theorem 5. Given the MPCG $G_M$ of basic block $v$, the size of the maximum clique in $G_M$ is an upper bound on the maximum number of cache misses that can occur in $v$.

Proof. Suppose $\{r_1, \ldots, r_m\}$ is a set of accesses in $v$ that can become misses together. Then, there exist concrete cache miss path $\sigma_i$ for each $r_i$ such that a walk in $G$ contains all the concrete miss paths $\sigma_i$. By Theorem 2, for every $\sigma_i$, there exists an abstract cache miss path $\pi_i = \alpha_{v,r_i}^T(\sigma_i)$. This implies that there exists a walk in $G$ which contains all the basic blocks of $\pi_i$ (for all $i$, $1 \leq i \leq m$). By Lemma 9, this means that there is no pairwise conflict in the set $\{\pi_1, \ldots, \pi_m\}$, and hence these abstract cache miss paths will form a clique in the MPCG $G_M$ of $v$.

Algorithm 4: Algorithm to find the maximum number of cache misses in every basic block of the program

1: for all basic blocks $v$ do
2: $R_v \leftarrow$ Set of NC accesses in $v$
3: Perform data flow analysis $D_v$
4: $V_M \leftarrow \bigcup_{r \in R_v}$ Miss paths of $r$
5: for all miss path $\pi_1$ of $r_1$, $\pi_2$ of $r_2$, such that $r_1, r_2 \in R_v$, $r_1 \neq r_2$ do
6: if $\forall w_1, \forall w_2 \in \pi_1 \cup \pi_2$, $(w_1 \in IN_{w_2} \lor w_2 \in IN_{w_1})$ then
7: Add edge $(\pi_1, \pi_2)$ to $E_M$
8: end if
9: end for
10: $\text{Misses}_v \leftarrow$ Size of Maximum clique in $(V_M, E_M)$
11: end for

Algorithm 4 summarizes the various steps discussed so far. We concentrate on the NC (non-classified) accesses, perform the data flow analysis for finding conflict information, and then create the MPCG. Finally, we find the size of the maximum clique in the MPCG, which gives us the maximum number of misses.

An important property of the algorithm is that the calculation for each basic block is carried out independently, and hence it can be easily parallelized. The data-flow analysis will reach a fix-point after a constant number of traversals of the entire CFG $G$, and hence has a complexity of $O(|E|)$. Note that we enforce a maximum limit ($N$) on the number of cache miss paths of a single access, so that if the number of miss paths of an access exceeds $N$, it is simply considered a cache miss and not included in $R_v$. Since both the number of miss paths and the length of a miss path are now constants, the for loop from lines 5-9 will take $O(\|R_v\|^2)$ time. Finding the maximum clique in a graph is an NP-Hard problem, with the complexity being exponential in the number of vertices in the graph, which in our case will have a maximum value of $N|R_v|$.  

95
Hence, if \( m \) is the maximum number of NC accesses inside a single basic block (across all basic blocks), then the final complexity of the algorithm is \( O(|V|(|E| + 2^m)) \).

The size of the maximum clique in the MPCG can still over-estimate the maximum number of cache misses, because of the two abstractions, i.e. allowing gaps in the miss path and bounding its maximum size. Due to these abstractions, execution of an abstract miss path may not necessarily result in a cache miss. However, in special cases, which in fact occur frequently for instruction caches, the above approach will actually give the most precise result.

**Lemma 10.** Given access \( r \) in basic block \( v \) which accesses \( m \), if \( v \) is the only basic block in the program which accesses \( m \), and if for all concrete cache miss paths \( \sigma \) of \( r \), \(|\alpha_{v,m}(\sigma)| \leq T\), then execution of any abstract cache miss path of \( r \) is guaranteed to cause a cache miss.

**Proof.** Consider an abstract cache miss path \( \pi \) of \( r \), and a walk \( \sigma \) in \( G \) which contains all the basic blocks of \( \pi \) and ends at \( v \) (and also does not contain any other instances of \( v \)). Since \( v \) is the only basic block which accesses \( m \), this walk will not contain any other accesses to \( m \). Moreover, since none of the concrete cache miss paths exceed the maximum length \( T \), either \( \pi \) contains \( v_{\text{start}} \), or it accesses at least \( k \) distinct cache blocks mapped to cache set of \( m \). In either case, the execution of walk \( \sigma \) will result in a cache miss for \( r \). \( \square \)

The implication of the above result (along with Lemma 9) is that if \( T \) is selected to be sufficiently large, and \( \text{Misses}_v \) is the size of the maximum clique in the MPCG of \( v \), then there exists a walk in \( G \) which contains all the miss paths in the clique, and hence will cause \( \text{Misses}_v \) number of cache misses in \( v \). In instruction caches, because of small cache block sizes and comparatively larger basic block sizes, there are many cases where a cache block is accessed solely inside a single basic block, and the above lemma guarantees the most precise analysis for the accesses to such cache blocks.

### 4.4.4 Complexity

In the algorithm presented in the previous sub-section, we are relying on an NP-Hard problem (finding maximum cliques) to find the maximum number of misses in a basic block. This raises the issue of whether there exists an efficient method to perform the same. In the section, we prove that the problem of finding the maximum number of cache misses experienced by a basic block for a general cache configuration is NP-Hard. Hence, an efficient algorithm which works for all cache configurations may not exist. We will show a reduction from the MAX-INDEPENDENT-SET problem in any arbitrary graph to the problem of finding the maximum number of cache misses for a basic block. Given a undirected graph \( G_A = (V_A, E_A) \), where \( V_A \) is the set of vertices and \( E_A \subseteq V_A \times V_A \) is the set of edges. Let \( k \) be the maximum degree
among all vertices in $G_A$ (the degree of a vertex is the number of edges incident on the vertex). An independent set $S$ is a subset of $V_A$ such that there does not exist an edge between any two vertices of $S$. The MAX-INDEPENDENT-SET problem is the problem of finding the maximum sized independent set in a graph, and it is known to be NP-Hard.

Given $G_A$, we construct a special program $P_A$ (shown in Figure 4.6) in the following manner : $P_A$ contains a basic block $b$ which contains $|V_A|$ cache accesses $r_1, \ldots, r_{|V_A|}$. We assume that the cache associativity is equal to the maximum degree in the graph (i.e. $k$), and the number of cache sets is equal to $|V_A|$. Every cache access in $b$ will map to a different cache set. There are $|E_A|$ conditionals (if-then-else statements) before $b$, each corresponding to an edge of $G_A$. Each hexagonal structure before $b$ in Figure 4.6 represents an if-then-else statement.

We select the cache blocks accessed by $P_A$ in such a manner that access $r_i$ (corresponding to vertex $v_i$) in $b$ has exactly one cache miss path. This can be achieved as follows : given an edge $e = (v_i, v_j)$, the miss path of $r_i$ will pass through the if-branch of the if-then-else statement corresponding to $e$, while the miss path of $r_j$ will pass through the else branch. In other words, the if-branch will access a different cache block mapped to the same cache set as the access $r_i$, while the else-branch will access the same cache set of $r_j$ (these will be the only cache accesses). For a cache of associativity $k$, every miss path must access at least $k$ distinct cache blocks mapped to the same cache set, but this can be achieved by having another basic block $b'$ before all the if-then-else statements, such that $b'$ will access the same cache block accessed by $r_i$ followed by $k - \text{degree}(v_i)$ different cache blocks to the same cache set, for all $r_i$ (where $\text{degree}(v_i)$ is the degree of $v_i$ in $G_A$).

The result of the above construction is that the access $r_i$ in $b$ will experience a cache miss only if the appropriate branches are taken in all the if-then-else statements corresponding to the edges incident on $v_i$. In particular, if $(v_i, v_j) \in E_A$, then both $r_i$ and $r_j$ in $b$ cannot become cache misses together, because their miss paths will pass through conflicting branches in the same if-then-else statement. Hence, if a set of accesses in $b$ can become cache misses together, then the vertices corresponding to those accesses must form an independent set. Conversely, given an independent set in $G_A$, the accesses corresponding to the vertices in the set can become cache misses together, because there will exist a path in $P_A$ which contains their miss paths, since all the miss paths will pass through different if-then-else statements.
Hence, if there is a technique which can find the maximum number of cache misses suffered by $b$, then this will also give the size of the maximum independent set in $G_A$. This shows that finding the maximum number of cache misses for a basic block is an NP-Hard problem, and a general polynomial-time technique which works for all cache configurations may not exist.

4.4.5 Finding worst-case profiles of basic blocks

For basic block inside loops, the maximum number of misses calculated using Algorithm 4 may not be possible for every iteration. In the example of Figure 4.4, the maximum number of cache misses in both $v1$ and $v2$ is 1, but they cannot cause 1 cache miss in every iteration. Hence, instead of just finding the maximum number of cache misses possible in a basic block, we would instead like to find all possible worst-case profiles of a basic block, in the form $<\text{Max misses, iters}>$, which says that $\text{Max misses}$ number of cache misses can happen in the basic block, only if it is executed once for every $\text{iters}$ number of iterations. In the example of Figure 4.4, the worst-case profiles of both $v1$ and $v2$ are $<1,2>$.

In order to find worst-case profiles, we use the already constructed MPCG, and continue searching for maximum cliques in the MPCG until we get a clique whose miss paths can all occur within a single iteration. In order to determine whether the miss paths in a clique can occur in a single iteration of the enclosing loop, we use the unique entry basic block $v_h$ of the loop. Essentially, if there exists a walk which does not pass through $v_h$ between every pair of basic blocks in the miss paths, then all the miss paths can occur together in a single iteration.

In the following, we only consider those miss paths which are completely present in the innermost loop containing the basic block (although the approach can be applied at all nesting levels). As a result, the following results do not apply to the cache behavior of a basic block when it is executed for the first time (for every entry to the loop containing the basic block from outside), but instead are applicable to the cache behavior for the rest of the iterations.

Consider basic block $v$, and let $L$ be the innermost loop containing $v$. Let $(V_M, E_M)$ be the MPCG of $v$, and let $V^L_M \subseteq V_M$ be those miss paths whose basic blocks are all present either inside loop $L$ or in one of its inner loops. Let $E^L_M \subseteq E_M$ be the edges incident on $V^L_M$. Let $v_h$ be the unique entry basic block of loop $L$. Due to the assumption that the CFG $G$ is reducible, every path from outside the loop $L$ must enter $L$ through $v_h$, and all the back edges of $L$ must also be incident on $v_h$. In other words, every iteration of $L$ must begin with $v_h$.

We use the notation $v_1 \leadsto_w v_2$ to mean that there exists a walk in $G$ from $v_1$ to $v_2$ which does not pass through $w$. Consider a set of misspaths $\{\pi_1, \ldots, \pi_k\} \subseteq V^L_M$ of accesses in $v$. By Lemma 9, we know that if $\forall v_1, v_2 \in \bigcup_{i=1}^{k} \pi_i$, either $v_1 \leadsto_v v_2$ or $v_2 \leadsto_v v_1$, then all the miss paths $\pi_i$ can occur together on a walk ending in $v$, and hence can cause $k$ misses in $v$. The following
two lemmas give the necessary and sufficient condition for a set of miss paths to occur in the same iteration.

**Lemma 11.** Miss paths $\pi_1$ of access $r_1$, $\pi_2$ of $r_2$, ..., $\pi_k$ of $r_k$ in $v$ can cause $k$ misses in $v$ in consecutive iterations of $L$ $\iff$ there exists a walk from $v$ to $v$ which contains exactly one instance of $v_h$ and contains all the miss paths.

**Proof.** If $v$ is executed in a iteration, it will bring all the cache blocks accessed by $r_1, \ldots, r_k$ to the cache. Hence, for these accesses to miss the cache in the next iteration, the miss paths should all occur before $v$ is executed in the next iteration, which will require a walk from $v$ to $v$ containing all the miss paths and passing through $v_h$ once. On the other hand, if such a walk exists, then an execution along this walk can result in $k$ misses in $v$ in consecutive iterations. □

**Lemma 12.** Given miss paths $\pi_1$ of access $r_1$, ..., $\pi_k$ of access $r_k$ in $v$, there exists a walk from $v$ to $v$ containing all the miss paths and exactly one instance of $v_h$ $\iff$ $\forall v_1, v_2 \in \cup_{i=1}^k \pi_i$, $v_1 \sim_v v_2 \lor v_2 \sim_v v_1$ and $\forall v_1, v_2 \in \cup_{i=1}^k \pi_i \cup \{v\}$, $v_1 \sim_{v_h} v_2 \lor v_1 \sim_{v_h} v_2$.

**Proof.** Let $W = \cup_{i=1}^k \pi_i$. Let $\sigma$ be the walk from $v$ to $v$ containing all miss paths and one instance of $v_h$. The first part of the forward direction is trivial, since all the basic blocks in $W$ will be present in the walk, and since $v$ only occurs at the endpoints of the walk, there must be a walk between every pair of basic blocks in $W$ which does not pass through $v$. We partition $W$ into two set $W_\rightarrow$ and $W_\leftarrow$, such that $W_\rightarrow$ contains all basic blocks of $W$ which occur on $\sigma$ before $v_h$, and $W_\leftarrow$ contains all basic blocks of $W$ which occur on $\sigma$ after $v_h$. Then $v \sim_{v_h} v'$ for all $v' \in W_\rightarrow$ and $v' \sim_{v_h} v$ for all $v' \in W_\leftarrow$. Also, for all $v_1, v_2$ in $W_\rightarrow$, either $v_1 \sim_{v_h} v_2$ or $v_2 \sim_{v_h} v_1$. Similarly, for all $v_1, v_2$ in $W_\leftarrow$, either $v_1 \sim_{v_h} v_2$ or $v_2 \sim_{v_h} v_1$. Finally, for all $v_1 \in W_\rightarrow, v_2 \in W_\rightarrow, v_1 \sim_{v_h} v \sim_{v_h} v_2 \Rightarrow v_1 \sim_{v_h} v_2$. This proves the forward direction.

For the reverse direction, we redefine $W_\rightarrow$ and $W_\leftarrow$ as follows: $W_\rightarrow = \{w \in W \mid v \sim_{v_h} w\}$ and $W_\leftarrow = \{w \in W \mid w \sim_{v_h} v\}$. Now, $\forall v_1, v_2 \in W_\rightarrow, v_1 \sim_{v_h} v_2 \lor v_2 \sim_{v_h} v_1$. Assume that $v_1 \sim_{v_h} v_2$. This walk will also not pass through $v$, because otherwise $v_1 \sim_{v_h} v$, and this would imply a walk between two instances of $v$ which does not contain $v_h$, which is a contradiction because $v_h$ is the entry block of the innermost loop containing $v$. Now, since $\forall v_1, v_2 \in W_\rightarrow, v_1 \sim_{v_h} v_2 \lor v_2 \sim_{v_h} v_1$, by Lemma 1, there exists a walk $\sigma_\rightarrow$ which contains all basic blocks in $W_\rightarrow$ and does not pass through $v_h$. This walk will also not pass through $v$. Similarly, there exists a walk $\sigma_\leftarrow$ which contains all basic blocks in $W_\leftarrow$ and does not pass through $v_h$ and $v$. Now, the walk from $v$ to the first basic block in $\sigma_\rightarrow$, followed by $\sigma_\rightarrow$, followed by the walk from the last basic block in $\sigma_\leftarrow$ to $v_h$, followed by the walk from $v_h$ to the first basic block in $\sigma_\leftarrow$, followed by the walk $\sigma_\leftarrow$ is the required walk between two instances of $v$ which does contains all basic blocks in $W$ and does not contain $v_h$. □
The implication is that if every walk between two basic blocks in a loop must pass through the entry basic block \( v_h \) (for example, between \( v_1 \) and \( v_2 \) in Figure 4.4), then such a walk must skip the basic block under analysis \( v \) for at least one iteration. Hence miss paths containing such basic blocks cannot cause cache misses in consecutive iterations. The next lemma gives the minimum number of iterations required to execute such miss paths.

**Lemma 13.** Given basic blocks \( w_1, \ldots, w_k \) in loop \( L \) (or one of its inner loops), every walk containing these basic blocks contains at least \( k - 1 \) instances of \( v_h \) \( \Leftrightarrow \forall w_i, w_j, \ 1 \leq i < j \leq k \), neither \( w_i \sim_{v_h} w_j \) nor \( w_j \sim_{v_h} w_i \).

**Proof.** We prove the forward direction by contradiction. Suppose every walk containing \( w_1, \ldots, w_k \) contains at least \( k - 1 \) instances of \( v_h \). Assume, for the sake of contradiction, that \( \exists w_i, w_j \) such that \( w_i \sim_{v_h} w_j \). Now consider all basic blocks apart from \( w_j \). Clearly, there exists a walk which contains all these \( k - 1 \) basic blocks which contains \( k - 2 \) instances of \( v_h \) and ends at \( w_i \) (this is because there exists a walk between every \( w_i \) and \( w_m \) which passes through \( v_h \)). Now, appending the walk between \( w_i \) and \( w_j \) which does not contain \( v_h \) gives a walk containing \( k - 2 \) instances of \( v_h \) and all the \( k \) basic blocks, which is a contradiction.

The reverse direction can also be proved using contradiction. Suppose \( \forall w_i, w_j, 1 \leq i < j \leq k \), neither \( w_i \sim_{v_h} w_j \) nor \( w_j \sim_{v_h} w_i \). Assume, for contradiction, that there exists a walk which contains all the basic blocks \( w_1, \ldots, w_k \) and \( k - 2 \) instances of \( v_h \). The instances of \( v_h \) partition this walk into \( k - 1 \) segments which do not contain \( v_h \). Since all \( k \) basic blocks \( w_1, \ldots, w_k \) are present in these segments, by pigeon-hole principle, there must exist at least one segment which contains two basic blocks \( w_i, w_j \). However, this would mean a walk between these basic blocks which does not contain \( v_h \), which contradicts our assumption. \( \square \)

**Theorem 6.** Given miss paths \( \pi_1 \) of access \( r_1, \ldots, \pi_k \) of access \( r_k \) in \( v \), where \( \pi_1, \ldots, \pi_k \in V^L_M \), if there exists \( W_C \subseteq \bigcup_{i=1}^k \pi_i \cup \{ v \} \) such that \( \forall w, w' \in W_C \) neither \( w \sim_{v_h} w' \) nor \( w' \sim_{v_h} w \) then a walk from \( v \) to \( v \) containing all the basic blocks in \( W_C \), with \( v \) only coming at the endpoints, requires at least \( |W_C| \) instances of \( v_h \).

**Proof.** Let \( n = |W_C| \). First, consider the case where \( v \not\in W_C \). Since \( v \not\in W_C \), we know that \( \forall w \in W_C \), either \( v \sim_{v_h} w \) or \( w \sim_{v_h} v \). However, if \( \exists w, w' \in W_C \) such that \( v \sim_{v_h} w \) and \( w' \sim_{v_h} v \), then this would imply \( w' \sim_{v_h} w \) which is a contradiction. Hence, either there is a walk from \( v \) to all \( w \) in \( W_C \), or there is a walk from all \( w \) in \( W_C \) to \( v \), which does not contain \( v_h \). Suppose all walks are only from \( v \) to all basic blocks in \( W_C \). Now, by Lemma 13, a walk containing all \( n \) basic blocks in \( W_C \) requires at least \( n - 1 \) instances of \( v_h \). If \( w_s \) is the start basic block, and \( w_e \) is the end basic block of this walk, then a walk from \( w_e \) to \( v \) will require...
another instance of \( v_h \). Hence, a walk from \( v \) to \( v \) containing all \( n \) basic blocks will require \( n \) instances of \( v_h \). The case when there is a walk from all \( w \) in \( W_C \) to \( v \) without passing through \( v_h \) can be proved in a similar manner.

If \( v \in W_C \), then by Lemma 13, a walk containing all basic blocks in \( W_C \) will require \( n - 1 \) instances of \( v_h \). If such a walk starts with \( v \) and ends with some basic block \( w_e \in W_C \), then since there is no walk from \( w_e \) to \( v \) which does not pass through \( v_h \), for such a walk to end \( v \) will require one more instance of \( v_h \). Similarly, if such a walk ends with \( v \) but starts with some basic block \( w_s \in W_C \), then a walk from \( v \) to \( w_s \) will require another instance of \( v_h \).

\[\square\]

**Algorithm 5:** Algorithm to find all worst-case profiles of a basic block \( v \) inside a loop with entry block \( v_h \)

1: \( \text{Max\_misses} \leftarrow \text{Size of maximum clique in MPCG} \ (V_M^L, E_M^L) \) of \( v \)
2: Perform Data flow conflict analysis \( D_{v_h} \)
3: repeat
4: \( \text{while clique of size Max\_misses does not exist in the MPCG do} \)
5: \( \text{Max\_misses} \leftarrow \text{Max\_misses - 1} \)
6: \( \text{end while} \)
7: \( S \leftarrow \text{Clique of size Max\_misses in the MPCG} \)
8: Add \( < \text{Max\_misses}, \text{iters}(S) > \) to the worst-case profiles of \( v \)
9: until \( \text{iters}(S) = 1 \)
10:
11: \( \text{function iters} (S) \)
12: \( V_B \leftarrow \text{Set of all basic blocks in miss paths of } S \) and \( v \)
13: for all \( w_1, w_2 \in V_B \) do
14: \( \text{if } w_1 \not\in IN_{w_2} \land w_2 \not\in IN_{w_1} \) then
15: \( \text{Add} (w_1, w_2) \) to \( E_B \)
16: \( \text{end if} \)
17: end for
18: return \( \text{Size of maximum clique in } (V_B, E_B) \)

We now apply these results to find the worst-case profiles, as depicted in Algorithm 5. We start with the MPCG \( (V_M^L, E_M^L) \) and find the clique of maximum size (line 1). We perform the data flow analysis \( D_{v_h} \) to find conflict information, so that \( IN_w \) (for all \( w \)) will consist of those basic blocks which have a walk in \( G \) to \( w \) which does not pass through \( v_h \) (line 2). This ensures that if for \( w_1 \) and \( w_2 \), neither of the two are present in the \( IN \) set of the other, then neither \( w_1 \leadsto v_h w_2 \) nor \( w_2 \leadsto v_h w_1 \).

We keep finding maximum cliques (of possibly decreasing sizes) in the MPCG until we find a clique which requires only 1 iteration (lines 3-9). We store the size and the number of iterations.
required by all cliques discovered in this process (line 8). Note that we only keep one WC profile for a given number of iterations *iters*, the one with the maximum number of misses. The function *Iters*(\(S\)) finds the number of iterations required by clique \(S\) (lines 11-18). It does so by creating the Basic Block Conflict Graph (BBCG), whose vertices \(V_B\) represent basic blocks in the miss paths in \(S\) and \(v\) (line 12). We add an edge between two basic blocks if there does not exist a walk between them which does not pass through \(v_h\) (lines 13-17). A clique \(S_B\) in the BBCG means (by Theorem 6) that a walk which starts and ends at \(v\) and passes through these basic blocks in \(G\) will require at least \(|S_B|\) iterations. Hence, the maximum clique in the BBCG would correspond to the minimum number of iterations required by miss paths in clique \(S\) (line 18). If the size of the maximum clique in the BBCG is 1, then this means that there is a walk between every pair of basic blocks in the miss paths which does not pass through \(v_h\), and by Lemmas 11 and 12, this guarantees that the miss paths can cause misses in \(v\) in consecutive iterations.

The complexity of Algorithm 5 remains exponential in the number of cache accesses inside a single basic block, since the maximum number of cliques in the MPCG is also exponential in its size, and in the worst case, the function *Iters* can be called for each clique. The complexity of the *Iters* function is exponential in the size of the BBCG, but since the maximum length of a miss path as well as the number of miss paths are constants, the size of the BBCG will be linear in the number of accesses within a single basic block.

We now present a modified version of the IPET ILP which allows us to use the different worst-case profiles of a basic block while finding the worst-case path. For basic block \(v\), let \(c_v\) be the maximum possible execution count. For basic blocks inside loops, the maximum execution count can be obtained by multiplying the loop bounds of all its enclosing loops. For every worst-case profile \(p = \langle \text{Max}\_\text{misses}_p, \text{iters}_p \rangle\) of basic block \(v\), let \(e_{v,p}\) be the WCET of \(v\) assuming \(\text{Max}\_\text{misses}_p\) number of cache misses in \(v\), and let \(y_{v,p}\) be the integer variable storing the execution count of this worst-case profile on the worst-case path. Finally, let \(y_v\) be the integer variable storing the total execution count of \(v\) on the worst case path. The modified ILP formulation is presented below :
Maximize
\[ \sum_{v \in V} \sum_{p \text{ profile of } v} e_{v,p} y_{v,p} \]  
Subject to
\[ \forall v \in V, \quad \sum_{(w,v) \in E} z_{w,v} = y_v = \sum_{(v,u) \in E} z_{v,u} \]  
\[ \forall v \in V, \quad \sum_{\text{profile } p \text{ of } v} \text{iter}_p y_{v,p} \leq c_v \]  

In the objective function to be maximized, we add execution times of all profiles of basic blocks. Note that basic block outside loops will only have one worst-case profile, with the maximum number of cache misses obtained using Algorithm 4. Equation 4.2 ensures proper control flow across basic blocks, through variables \( z_{u,v} \), which store the number of times execution passes through the edge \((u, v) \in E\). The sum of execution counts for different profiles of a basic block will be equal to the total execution count the basic block on the WC path (Equation 4.3). Every occurrence of the worst case profile \( p \) will consume \( \text{iter}_p \) number of iterations, and the maximum number of iterations is upper bounded by \( c_v \). This establishes an upper bound on \( y_{v,p} \), which is the maximum number of times \( v \) can cause \( \text{Max misses}_p \) misses (Equation 4.4). While the WC path obtained using the original IPET formulation follows a single path in all iterations of a loop, the above ILP formulation allows the possibility of following different paths across iterations, and uses the appropriate WCET for basic blocks in such cases. Finally, we note that in Algorithm 5, it is not necessary to continue searching for cliques until we find a clique which requires only 1 iteration. We can stop the loop at any point and simply assume that the last clique found requires 1 iteration. This allows a trade-off between analysis precision and efficiency.

4.5 ILP-based Approach

While the algorithms in the previous section can tackle the precision issues illustrated in Figures 4.1-4.4, in order to solve the precision issue of Figure 4.5, we need information about the WCEP. However, as explained earlier, the WCEP is closely tied with the predicted cache behavior, and hence one cannot directly use the WCEP obtained using the IPET formulation. In this section,
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_v$</td>
<td>Integer variable storing the execution count of basic block $v$</td>
</tr>
<tr>
<td>$x_{v,m}$</td>
<td>Integer variable storing the total number of cache misses suffered by access to cache block $m \in Acc_f(v)$</td>
</tr>
<tr>
<td>$x_{v,m}^\pi$</td>
<td>Integer variable storing the number of cache misses of by access to $m \in Acc_f(v)$ along cache-miss path $\pi$</td>
</tr>
<tr>
<td>$z_{v,w}$</td>
<td>Integer variable storing the execution count of edge between basic blocks $v$ and $w$</td>
</tr>
<tr>
<td>$e_v$</td>
<td>Execution time of basic block $v$ assuming NC-instructions as cache hits</td>
</tr>
<tr>
<td>$CMP$</td>
<td>Cache miss penalty</td>
</tr>
</tbody>
</table>

Table 4.3: Notation used in ILP-based private cache analysis

We show how to integrate cache-miss paths into the IPET formulation [58], so that an access suffers a cache miss only if the WCEP contains a miss path of the access. As an added advantage, this approach automatically solves the various issues illustrated in Figures 4.1-4.4, so that the algorithms of the previous sections are not needed.

We build our ILP formulation on top of the IPET ILP, and introduce new integer variables for every access not classified as Always-Hit by Must analysis, as well as for each cache-miss path of these accesses. The number of cache misses suffered along a cache-miss path will be constrained by the execution counts of the basic blocks in the miss path.

Table 4.5 contains all the notations used in the ILP. Since we only focus on Not Classified (NC) accesses (i.e. accesses not classified as Always-Hit by Must analysis), only the first access to cache block $m$ in basic block $v$ needs to be considered. Let $Acc_f : V \rightarrow \mathcal{B}$, $Acc_f(v)$ gives the set of cache blocks accessed in $v$, such that the first access to these cache blocks in $v$ is non-classified. For each basic block $v$, let $MP_v : Acc_f(v) \rightarrow \mathcal{P}(\mathcal{P}(V))$ give the set of abstract cache miss paths of NC-accesses in $v$. These abstract miss paths are obtained using the AI-based approach of Section 4.3. Hence, for $m \in Acc_f(v)$, $MP_v(m) = OUT_{v,\text{start}}$, where $OUT_{v,\text{start}}$ is determined using the analysis for $m$ in $v$. Finally, for $m \in Acc_f(v)$, let $BB_{v,m} = \bigcup_{\pi \in MP_v(m)} \pi$ be the set of basic blocks in cache miss paths of $m$. Note that $e_v$ is the WCET of basic block $v$, assuming that all NC accesses hit the cache.

For each basic block $v$, $y_v$ stores the execution count of $v$ on the worst-case execution path. For an edge between basic blocks $v$ and $w$ in the CFG, the variable $z_{v,w}$ stores the number of times execution passes from $v$ to $w$ on the WCEP. The objective is to find the WCEP, i.e. the execution counts of basic blocks which maximizes the execution time of the program. The execution counts are constrained by the program structure, which basically places the restriction
that the number of times execution enters a basic block (through an incoming edge in the CFG) must be the same as the number of times execution leaves the basic block (through an outgoing edge), and this will also be the execution count of the basic block. Hence, the sum of the \( z \) variables for all incoming edges to a basic block will be the same as the sum of \( z \) variables for all outgoing edges. Following is our proposed ILP:

\[
\text{Maximize } \sum_{v \in V} (e_v y_v + \sum_{m \in \text{Acc}_f(v)} \text{CMP } x_{v,m}) \tag{4.5}
\]

subject to

\[
\forall v \in V, \quad y_v = \sum_{w \in \text{pred}(v)} z_{w,v} = \sum_{w' \in \text{succ}(v)} z_{v,w'} \tag{4.6}
\]

\[
\forall v \in V, \forall m \in \text{Acc}_f(v), \quad \begin{cases} 
  x_{v,m} \leq \sum_{\pi \in \text{MP}_v(m)} x_{v,m} \\
  \forall w \in \text{BB}_v(m), \sum_{\pi \in \text{MP}_v(m): \pi \cap w = \emptyset} x_{v,m} \leq y_w \tag{4.7}
\end{cases}
\]

The product \( e_v y_v \) is the contribution of \( v \) to the execution time of the program, assuming that all NC-instructions are cache hits. The variable \( x_{v,m} \) accounts for the cache misses suffered by access to \( m \) in \( v \). Each cache miss causes an additional execution time of \( \text{CMP} \). Hence, the objective function is the sum of the total execution times of all basic blocks on the WCEP (Equation 4.5). Equation 4.6 encodes the flow constraint for each basic block.

For each miss path \( \pi \) of the access to \( m \) in \( v \), the variable \( x_{v,m}^{\pi} \) counts the number of misses suffered by the access along \( \pi \). If an access has multiple cache miss paths, then any of those miss paths could be present on the WCEP. Moreover, for an access inside a loop, multiple cache-miss paths may actually be present on the WCEP (for example, different miss paths could be activated in different iterations). Hence, the total number of misses suffered by an access to \( m \) in \( v \) (\( x_{v,m} \)) is bounded by the sum of its \( x_{v,m}^{\pi} \) variables (Equation 4.7). A miss path is present on the WCEP if the execution counts of all basic blocks on the miss path are non-zero. Further, the maximum number of times that a miss path is executed would be equal to the minimum among the execution count of the basic blocks on the miss path. Finally, if a basic block is present on multiple miss paths of the same access, then a single execution of the basic block can activate at most one miss path during actual execution, and hence cause at most one miss. The set of constraints represented by Equation 4.8 encode all the above
requirements.

In addition to the above constraints, loop constraints, which will bound the execution count of loop headers and infeasible path constraints can also be added. An infeasible path generally takes the form of a set of basic blocks, which will never be executed together (due to their execution being guarded by conditionals whose conjunction is not satisfiable). The constraints will place an upper bound on the sum of the execution counts \( y_v \) of such basic blocks. By appending them to the above ILP, we not only guarantee that the worst case path will not contain the infeasible path, but also that no cache miss path will contain such basic blocks, and thus the cache misses caused due to infeasible paths will be ignored.

4.6 Experimental Evaluation

4.6.1 Setup

We have implemented the proposed techniques on top of the Chronos WCET analyzer \([41]\). Chronos takes as input the binary of the program whose WCET is to be determined (compiled for SimpleScalar ISA), along with annotations describing loop bounds and infeasible path information. First, it reconstructs the control flow graph (CFG) of the program and also in-lines all function calls. It then performs cache analysis and pipeline analysis to obtain upper bounds on the execution time of basic blocks, which are then encoded using the IPET ILP whose solution would be the WCET of the program. In this work, we focus exclusively on instruction cache analysis, and assume a perfect data cache. Further, we assume a single level hierarchy, although the proposed techniques can also be applied for analysis of the L1 cache in a multi-level cache hierarchy.

For cache analysis, Chronos uses Abstract Interpretation based Must and May cache analysis \([29]\) and the safe version of persistence analysis \([36]\) to provide a hit-miss classification to every memory access. For pipeline analysis, Chronos builds a separate execution graph \([40]\) for every basic block, modeling all the pipeline stages of every instruction in the basic block, their execution time estimates, and their interdependencies. Note that this requires a safe hit-miss classification for every cache access. These execution graphs are then used to determine the WCET estimate of every basic block.

We implement the proposed techniques in the following manner. We work on the virtually function-inlined CFG of the program reconstructed by Chronos, and perform AI-based Must, May and Persistence analysis to obtain safe hit-miss classifications for every memory access. We then focus on the accesses which remain Not Classified (NC) by AI-based Must and May analysis, and find the cache miss paths of such accesses using the proposed AI-based technique.
We then separately implement the ILP-based and the algorithmic approaches. The ILP-based approach (Section 4.5) encodes the cache miss paths of NC-accesses into the IPET ILP, and uses the cache miss penalty for an access only if the worst-case execution path contains a cache miss path of the access. However, the ILP-based approach requires as input the maximum execution time of every basic block, assuming all NC-accesses hit the cache (the constant $e_v$ for basic block $v$). It also assumes that every cache miss suffered by an NC access will add a constant cache miss penalty ($CMP$) to the final execution time. This does not necessarily hold true for architectures with pipelines, since accesses to main memory will frequently occur in parallel with ALU stages of other instructions, so that some portion of the cache miss penalty may not be visible in the final execution time. Hence, assuming a constant cache miss penalty for every cache miss may over-estimate the WCET by ignoring the effect of instruction parallelism.

To make matters worse, some architectures may exhibit strong impact timing anomalies \cite{61}, which happen when a local increase in the execution time of an instruction results in a greater global increase in the total execution time of the program. In our case, the increase in the execution time of the program due to an instruction cache miss may become greater than the main memory latency. \cite{61} lists the two main factors which may cause strong impact timing anomalies: (1) an out-of-order functional unit which allows instructions to be dispatched in an order different from the program order or (2) multiple, in-order, non-uniform functional units. An example of the latter is an architecture with separate functional units for floating point and integer computations. For architectures with strong impact timing anomalies, the ILP-based approach must assume in the worst-case that every cache miss can potentially cause a strong impact timing anomaly, and the cache miss penalty ($CMP$) must take into the account the maximum possible increase in execution time due to a strong impact timing anomaly. This would further hamper the precision of the ILP-based approach.

In our experiments, we assume an in-order, 5-stage pipeline with a single ALU unit capable of both integer and floating point computations. This precludes the presence of strong impact timing anomalies during execution, and hence we assume the cache miss penalty ($CMP$) in the ILP-based approach to be the main memory latency. We first perform pipeline analysis assuming that all NC-accesses hit the cache, and obtain the execution time estimate of every basic block (the constant $e_v$ in the ILP formulation). We use these execution time estimates of basic blocks, as well as the cache miss paths of the NC-accesses to generate the ILP (as described in Section 4.5). Since the maximum increase in the execution time of the program is guaranteed to be less than or equal to the cache miss penalty for every cache miss, the solution
of the ILP is guaranteed to be greater than the actual WCET of the program.

We now describe our implementation of the algorithmic approach (Section 4.4). We first find those accesses which do not have any cache miss paths, and change their classification to AH (Section 4.4.1). We then use the method described in Section 4.4.2 to find persistent accesses inside loops, which may have been missed due to the precision issue in AI-based persistence analysis. Finally we use Algorithms 4 and 5 to find the worst-case profiles of all basic blocks, which gives the maximum number of cache misses in a basic block, and if the basic block is inside a loop, then the minimum number of iterations required to cause them.

For every worst-case profile $p$ of basic block $v$, we perform pipeline analysis in the following manner: if the maximum number of misses possible in basic block $v$ is $k$ ($= \text{Max}\text{-}\text{misses}_p$), and the number of NC-accesses in $n$, then we create $\binom{n}{n-k}$ versions of $v$, corresponding to every choice of $n-k$ accesses. For a given version corresponding to a particular choice of $n-k$ accesses, we change the hit-miss classification of these $n-k$ accesses from NC to AH, then perform the pipeline analysis of $v$ to obtain an execution time estimate. We do this for all versions, and then take the maximum of the execution time estimates, which will be used for the constant $e_{v,p}$ in the modified IPET ILP formulation described towards the end of Section 4.4.5. Note that in every version corresponding to profile $p$ of basic block $v$, the maximum number of misses would be $n - (n-k) = k$, which establishes the safety of our approach. By performing a separate pipeline analysis for every version, we can also statically find instances when the cache miss latency of accesses which do miss the cache could be hidden by instructions executing in other stages, thus reducing the overall WCET estimate.

We used lp\_solve to solve the generated ILPs, and the clique library for finding maximal cliques in graphs. Our experiments were conducted on a 4-core Intel i5 CPU with 4 GB memory. We experimented on benchmarks chosen from Mälardalen, DEBIE, MiBench and StreamIt suites (obtained from the TACLEBench collection). For each benchmark, we use a different L1 instruction cache, with cache size approximately equal to 10% of the code size. The L1 cache hit latency is 1 cycle, while the main memory latency is 30 cycles. While determining the cache miss paths of accesses, the maximum miss path length ($T$) is restricted to 16, and the maximum number of miss paths per access ($N$) is restricted to 100. If an access has too many miss paths, then it is highly likely that one of the miss paths will be on every program path. A large number of miss paths also reduces the likelihood of finding conflicting accesses which may never occur together. We experimented with different bounds and found that a larger bound on the number of miss paths does not have any impact on the precision of the final WCET.

---

1\text{http://users.aalto.fi/pat/cliquer.html}
2\text{http://www.tacle.eu/index.php/activities/taclebench}
### Benchmark Code Size

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>1.1 KB</td>
</tr>
<tr>
<td>binarysearch</td>
<td>0.3 KB</td>
</tr>
<tr>
<td>expint</td>
<td>0.5 KB</td>
</tr>
<tr>
<td>compress</td>
<td>4 KB</td>
</tr>
<tr>
<td>crc</td>
<td>2.2 KB</td>
</tr>
<tr>
<td>ndes</td>
<td>5 KB</td>
</tr>
<tr>
<td>qurt</td>
<td>2.5 KB</td>
</tr>
<tr>
<td>ud</td>
<td>2 KB</td>
</tr>
<tr>
<td>countnegative</td>
<td>0.9 KB</td>
</tr>
<tr>
<td>lms</td>
<td>3.6 KB</td>
</tr>
<tr>
<td>qsort-exam</td>
<td>1.2 KB</td>
</tr>
<tr>
<td>select</td>
<td>1 KB</td>
</tr>
<tr>
<td>sqrt</td>
<td>0.5 KB</td>
</tr>
</tbody>
</table>

Table 4.4: Mälardalen Benchmarks and their code sizes

#### 4.6.2 Mälardalen Benchmarks

We compare the WCETs obtained using the proposed ILP-based and algorithmic approaches, with the WCET obtained using AI-based approach. Figure 4.7 shows the precision improvement (in %) of the WCET computed using ILP-based and algorithmic approaches, as compared to the WCET computed using AI-based approach, for benchmarks from the Mälardalen suite. The precision improvement is computed as $\frac{\text{WCET}_{AI} - \text{WCET}_x}{\text{WCET}_{AI}} \times 100\%$, where $x \in \{\text{Algorithmic, ILP}\}$. Note that we experimented on all benchmarks from the Mälardalen suite, and the figure shows the results only for those benchmarks for which the proposed approaches showed non-zero precision improvement.

The average precision improvement is 11.3 % for the ILP-based approach, and 10.2 % for the algorithmic approach. The results demonstrate that even though the algorithmic approach does not use the global worst-case execution path information, it still matches the precision improvement of the ILP-based approach for most benchmarks. This validates our hypothesis that the precision issues tackled by the algorithmic approach have a more significant impact than knowledge about the worst-case execution path, and they account for most of the precision improvement of the ILP-based approach. From the computational point of view, these precision
issues do not require information about the worst-case execution path, and can be determined separately for each basic block. This also allows a better integration with pipeline analysis.

In fact, out of the 4 benchmarks (qurt, ud, lms, sqrt) where the algorithmic approach provides higher precision improvement than the ILP-based approach, 3 of them perform floating point operations, which have a much higher latency as compared to other instructions. This also allows such high-latency instructions to hide the main memory latency of cache misses occurring in parallel. However, the ILP-based approach cannot take advantage of the available instruction parallelism, since it must add the entire main memory latency for every cache miss to the final WCET.

We also note that all the above benchmarks have multiple paths (i.e. if-then-else or switch-case statements), and in fact, these are the only multi-path benchmarks in Mälardalen suite, while all the other benchmarks for which the proposed approaches do not show any precision improvement contain only a single path. The precision issues which are targeted by the algorithmic approach require multiple paths in the program, and if there is only a single path in the program, then it will trivially contain all miss paths. Hence, the results demonstrate that majority of multi-path programs are highly likely to benefit from the proposed approaches.

Table 4.5 shows the exact break-down of the precision improvement shown by the algorithmic approach, in terms of the various precision issues. For each benchmark, the second column contains the number of cache accesses which should be classified as Always-Hit, but are missed by must analysis, determined using the approach of Section 4.4.1 (Theorem 3). The third column contains the number of cache accesses which should be classified as Persistent, but are
Table 4.5: Break-down of precision improvement

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No. of accesses/BBs which benefit from approach of Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.4.1</td>
</tr>
<tr>
<td>adpcm</td>
<td>0</td>
</tr>
<tr>
<td>binarysearch</td>
<td>0</td>
</tr>
<tr>
<td>expint</td>
<td>1</td>
</tr>
<tr>
<td>compress</td>
<td>5</td>
</tr>
<tr>
<td>crc</td>
<td>0</td>
</tr>
<tr>
<td>ndes</td>
<td>2</td>
</tr>
<tr>
<td>qurt</td>
<td>6</td>
</tr>
<tr>
<td>ud</td>
<td>4</td>
</tr>
<tr>
<td>countnegative</td>
<td>0</td>
</tr>
<tr>
<td>lms</td>
<td>2</td>
</tr>
<tr>
<td>qsort-exam</td>
<td>0</td>
</tr>
<tr>
<td>select</td>
<td>0</td>
</tr>
<tr>
<td>sqrt</td>
<td>1</td>
</tr>
</tbody>
</table>
missed by persistence analysis, determined using the approach of Section 4.4.2 (Theorem 4). The fourth column contains the number of basic blocks for which the worst-case number of misses were less than the number of NC accesses, determined using Algorithm 4. Finally the fifth column contains the number of basic blocks which have worst-case profiles requiring more than one iteration of their enclosing loop, determined using Algorithm 5.

The results show that all four approaches are successful to different degrees in different benchmarks. Across the benchmarks, the approaches of Section 4.4.1 and 4.4.5, which find AH-accesses missed by must analysis and worst-case profiles of basic blocks inside loops are slightly more successful. The precision issue in the must analysis requires only if-statements (even without corresponding else segments) to manifest, which are more frequently found in the above benchmarks. A large number of basic blocks also manifest worst-case profiles which require more than one iterations of their enclosing loops. Again, this precision issue only requires loops which have multiple paths from the entry to the exit of the loop. On the other hand, the approaches of Section 4.4.2 and 4.4.3 are slightly less successful. Most of the persistent accesses are identified by AI-based persistence analysis, however, there are benchmarks where our approach is able to identify higher number of persistent accesses. Similarly, our approach is also able to identify basic blocks where the maximum number of misses is strictly less than the number of NC-accesses.

Note that the number of accesses/BBs which benefit from the proposed approaches itself does not have a direct bearing on the amount of precision improvement in the WCET, since the execution counts of those accesses/BBs on the WCEP would also matter. The ILP-based approach only considers those accesses as misses whose miss paths are present on the worst-case path. This allows it to not only automatically subsume the precision improvement of all the four techniques used by the algorithmic approach, but also identify precision issues caused due to the worst-case path (as shown in the example of Figure 4.5). However, since it defers finding the final hit-miss classification to the ILP stage, it is also difficult to integrate the ILP-based approach with pipeline analysis, due to which the ILP-based approach cannot take advantage of instruction parallelism. Finally, we note that since these benchmarks are fairly small (code sizes in Table 4.4), the analysis time for all benchmarks was in the range of few seconds.

4.6.3 Benchmarks with Floating Point Operations

The results with the Mälardalen Benchmarks show that while the ILP-based approach provides better precision improvement than the algorithmic approach for majority of the benchmarks, it also suffers from lack of integration with pipeline analysis in benchmarks with floating point operations. To further test this behavior, we experimented on larger benchmarks which contain
Figure 4.8: Graph showing ratio of WCET obtained using ILP/Algorithmic approach as compared with AI-based approach for benchmarks with floating point operations. The selected benchmarks are basicmath and susan from MiBench suite, audiobeam and fmref from StreamIt suite. The code size of these benchmarks are 116 KB, 48 KB, 47 KB and 48 KB respectively. For this experiment, we assumed a 4-way 4 KB L1 cache with cache block size of 32 bytes.

Figure 4.8 shows the WCET ratio \(\frac{WCET}{WCETA_i}\) for the four selected benchmarks. Note that a lower WCET ratio corresponds to larger precision improvement. The WCET ratio is greater than 1 for the ILP-based approach, for all the 4 benchmarks, which means that the WCET determined by the ILP-based approach is greater than the WCET determined by the AI-based approach. This has happened because any precision improvement due to better prediction of cache behavior has been overshadowed by the loss in precision due to lack of integration with pipeline analysis. In particular, in these benchmarks it is more likely that cache miss penalties are hidden by floating point operations occurring in parallel. Both the AI-based and algorithmic approach would be able to identify such scenarios statically during the pipeline analysis stage. The algorithmic approach continues to determine lower WCET estimates, with the average precision improvement over the 4 benchmarks being 2.4 %.

### 4.6.4 DEBIE Benchmarks

We also experimented with a real-world benchmark, DEBIE-1, to show the scalability of the proposed approaches. The DEBIE-1 software is used to control the DEBIE-1 instrument, which is placed on a satellite to observe micro-meteoroids and small space debris by detecting
<table>
<thead>
<tr>
<th>DEBIE-1 Task</th>
<th>Root function</th>
<th>Code Size</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task - 1</td>
<td>TC_InterruptService</td>
<td>78 KB</td>
<td>4 KB</td>
</tr>
<tr>
<td>Task - 2</td>
<td>HandleTelecommand</td>
<td>420 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>Task - 3</td>
<td>HandleHealthMonitoring</td>
<td>321 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>Task - 4</td>
<td>TM_InterruptService</td>
<td>152 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>Task - 5</td>
<td>HandleHitTrigger</td>
<td>92 KB</td>
<td>4 KB</td>
</tr>
<tr>
<td>Task - 6</td>
<td>HandleAcquisition</td>
<td>466 KB</td>
<td>16 KB</td>
</tr>
</tbody>
</table>

Table 4.6: DEBIE-1 Tasks

Figure 4.9: Graph showing precision improvement in WCET of DEBIE-1 Tasks

Out of the 6 tasks, the proposed approaches show zero or negligible precision improvement for 3 tasks. The precision improvement of the other 3 tasks is shown in Figure 4.9. The average precision improvement of the ILP-based approach is 1.6 %, while for the algorithmic approach, it is 1.9 %. The algorithmic approach provides higher precision improvement than the ILP-based approach for 2 out of the 3 tasks. Note that the DEBIE-1 tasks contain floating point operations, which would explain slightly better performance of the algorithmic approach. The average precision improvement of both the approaches, however, has decreased when compared with their performance on the smaller Mälardalen benchmarks. Since the DEBIE-1 tasks are significantly larger, improving the cache prediction of some cache accesses does not have the

impacts on sensors. The DEBIE-1 software itself contains six tasks, and in this experiment, we separately compute the WCET of the six tasks using AI-based, ILP and Algorithmic approach. Table 4.6 shows the root function of these tasks, along with the code size and the L1 cache size that we used while performing the experiments.
### Table 4.7: Break-down of precision improvement

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No. of accesses/BBs which benefit from approach of Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.4.1</td>
</tr>
<tr>
<td>basicmath</td>
<td>3</td>
</tr>
<tr>
<td>susan</td>
<td>7</td>
</tr>
<tr>
<td>fmref</td>
<td>63</td>
</tr>
<tr>
<td>audiobeam</td>
<td>9</td>
</tr>
<tr>
<td>DEBIE-1 T1</td>
<td>15</td>
</tr>
<tr>
<td>DEBIE-1 T4</td>
<td>10</td>
</tr>
<tr>
<td>DEBIE-1 T5</td>
<td>244</td>
</tr>
</tbody>
</table>

same impact on the final WCET. On the other hand, the final WCET values of the DEBIE-1 benchmarks are also much larger as compared with the Mälardalen benchmarks, so that the decrease in WCET, in terms of the number of processor cycles, by the proposed approaches in the DEBIE-1 tasks is actually comparable to the total WCET values of majority of the Mälardalen benchmarks.

Table 4.7 shows the break-down of the precision improvement in terms of the various techniques used by the algorithmic approach (similar to table 4.5) for the DEBIE-1 Tasks and the benchmarks of Section 4.6.3. Again, all four techniques are successful to varying degrees across benchmarks. Notably, all benchmarks contain accesses which should be classified as AH but are missed by Must analysis. Majority of the benchmarks also have basic blocks whose worst-case profiles require multiple iterations of their enclosing loop.

Table 4.8 contains details about the time taken (in seconds) by various approaches. The second column contains the time taken by the AI-based approach, which includes the time taken for Must, May and Persistence analysis and solving the IPET ILP. The third column contains the time taken to find cache miss paths of accesses (the AI-based approach of Section 4.3). The fourth column contains the time taken to solve the ILP of Section 4.5. The last column contains the time taken by the algorithmic approach, which includes the time taken by Algorithms 4 and 5, and to solve the ILP of Section 4.4.5. Since both the ILP-based and algorithmic approach need cache miss paths, the total time taken by the ILP-based approach would be the sum of columns 3 and 4, while the total time taken by the algorithmic approach would be the sum of columns 3 and 5.
The time taken to find miss paths dominates the analysis time of both the ILP-based and the algorithmic approach. One of the reasons could be that we perform a separate AI analysis for every basic block to find the cache miss paths, and finding the fix-point in the AI-based analysis requires a constant number of traversals of the entire CFG. However, the total analysis time is still reasonably small for both the ILP-based and algorithmic approaches even for larger benchmarks.
Chapter 5

Conclusion and Future Work

In this thesis, we have proposed precise approaches for statically analyzing the worst-case behavior of private and shared caches towards the objective of obtaining tight estimates on the worst case execution time of programs. Since resource allocation in real-time and embedded systems is carried out on the basis of estimated WCET, precise estimates are crucial to leverage any advantage of modern architectural features such as multi-cores, caches, etc. However, these features introduce variable behavior in the execution time of individual program instructions, and hence they make the WCET estimation problem considerably harder.

In this work, we target the on-chip hardware cache, and try to find safe lower bounds on the number of cache hits. For shared caches in a multi-core architecture, guaranteeing that a particular access always hits the cache is not possible, due to the unpredictable interleaving of accesses from different cores. Hence, we shift our focus to finding minimum number of cache hits among groups of cache accesses. Our approach, called Worst Case Interference Placement (WCIP), tries to find the worst-case arrival of interfering accesses, which can cause maximum number of cache misses. We formulate and investigate this problem in detail, and propose an Integer Linear Programming based approach and an approximate algorithmic approach, and in the process also prove that the WCIP problem is NP-Hard. A crucial property of WCIP is that the increase in WCET due to shared cache interference can be shown to be a linear function of the amount of cache interference. Experimentally, WCIP gives significantly precise estimates of WCET as compared to earlier approaches to shared cache analysis, and also guarantees low increase in the WCET for small amount of cache interference.

We then apply the same high-level idea of targeting worst-case behavior of groups of cache accesses to private cache analysis. In programs with large number of execution paths, an individual access may hit or miss the cache along different paths. Previous approaches would simply consider the worst-case behavior of an individual access across all paths, but we discover various
scenarios where such an approach is imprecise, for example when the worst-case behavior of two accesses can never occur together, or it can never occur for all iterations of an enclosing loop, or it can never occur in the worst-case execution instance of the program. To automatically detect such cases in a scalable manner, we propose a two-step approach, where we first use Abstract Interpretation to obtain a summary of the path-sensitive behavior of individual accesses, and then use an algorithmic approach to find worst-case profiles of basic blocks. The algorithmic approach can also be replaced by an Integer Linear Programming based approach which finds the actual cache behavior along the worst case execution path. Experimentally, our approach provides more precise estimates of WCET over a wide range of benchmarks as compared to state-of-the-art AI-based approaches.

There are several directions of future work:

• While our approach to shared cache analysis provides substantial precision improvement over previous approaches, there are several avenues for further precision improvement. For example, we abstract the interference caused by co-running programs in the form of number of interfering shared cache accesses and number of distinct cache blocks. The sequence and ordering information about the interfering accesses is lost in this abstraction. Using information about possible sequences of interfering accesses (for example, sequence of cache sets accessed) in conjunction with the sequencing information of the program under analysis could further improve the precision of the analysis.

• The number of shared cache misses (and hence the increase in the WCET) of a program depends on the number of interfering accesses, which in turn depends on co-running programs. The scheduling policy decides the task assignment on cores and ordering of tasks on the same core. Normally, these policies take a constant WCET of tasks as input, and then schedule the tasks to ensure that the task deadlines are met. However, in a multi-core architecture, these scheduling decisions can affect the WCET of a task. Our approach provides a compact, piecewise linear relationship between the WCET of a task and the amount of interference it suffers. An important extension to the current real-time multi-core scheduling approaches is to leverage this information and improve the schedulability of task sets and utilization by using the relation of the WCET of individual tasks with co-running tasks on other cores.

• Our approach to find the maximum number of misses among a group of cache accesses is so far limited to cache accesses inside the same basic block. However, it is possible that two accesses in two different basic blocks may never cause misses together. A non-trivial
extension to our approach could be to efficiently find such accesses and use the resulting information to improve the precision of the WCET.

- The idea of finding the worst-case performance of a group of cache accesses can be applied for other analyses: analysis of the data cache, analysis of multi-level caches, Cache Related Pre-emption Delay (CRPD) analysis, etc. In all these cases, the current approaches try to find the worst-case behavior of each access individually, but the worst-case behavior of two accesses may not be possible simultaneously.
Bibliography


[56] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In CASES, 2001. 3


[62] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In VMCAI, 2004. 79


