Cache Analysis for Multi-level Data Caches

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of
Master of Engineering
in
Computer Science and Engineering

by

Kartik Nagar
Advisor: Prof. Y N Srikant

Computer Science and Automation
Indian Institute of Science
BANGALORE – 560 012

JUNE 2012

TO

my Parents

Acknowledgements

I would like to take this opportunity to thank Prof. Y.N. Srikant for providing me with the opportunity for pursuing this project as well as for his guidance during the course of the project. I would like to thank IISc for providing me with the resources to complete the project, as well as for creating a great research environment which continues to inspire me. I would like to thank all the professors of the Department of CSA for their wonderful teaching which forced me to be inquisitive and ask questions, and at the same time instilled a deep understanding of the fundamentals. I would like to thank all my friends for being there for me when needed and for helping me to adjust and feel at home in a new environment. Special mention also to Rathijit Sen who started this project few years ago, and built a solid foundation, which allowed to us to concentrate on the issues addressed by this project. Finally, a special thanks to my parents who made all the right decisions for me when I was in no position to make them and set me up on this path, and have since given me the freedom to see it through to its end. Without their support, I would not be in this position.

Abstract

Obtaining a precise and safe estimate of the Worst Case Execution Time(WCET) of executables is an important problem for real-time systems. Modelling the underlying micro-architecture, especially the caches, while determining the WCET estimates has an enormous impact on its precision. Large amounts of research work has concentrated on predicting the cache contents to precisely estimate the execution time of memory-accessing instructions. Must Analysis is one of the most widely used techniques to provide a safe estimate of cache contents, but is known to be highly imprecise. In our work, we propose to use May Analysis to assist the Must Analysis cache update and make it more precise for a class of programs. We prove the safety of our approach as well as provide examples where our Improved Must Analysis provides better precision. Further, we also detect a serious flaw in the original Persistence Analysis, and use Must and May Analysis to assist the Persistence Analysis cache update, to make it safe and more precise than the known solutions to the problem. Finally, we give a definitive multi-level cache analysis, which is a simple extension of single-level cache analysis framework. Our analysis takes into account the writeback effect, and we use the novel idea of partial and pseudo cache blocks to improve the precision of multi-level joins in Must analysis. Our experiments show that using partial blocks can result in an improvement in precision as high as 25.16%. We also provide a safe and precise way of performing May and Persistence Analysis for multi-level caches in the presence of write-backs.

Contents

List of Tables

List of Figures

Keywords

Worst Case Execution Time Estimation, Real-time systems, Multi-level Set-associative Data Caches, Static Program Analysis, Abstract Interpretation

Chapter 1

Introduction

Task Scheduling on hard and soft real systems generally requires an estimate of the WCET of the programs to be scheduled. The estimate must be *safe*, i.e. no run of the program should go beyond the WCET time, and as *precise* as possible, to optimize the scheduling and minimize the wastage of resources. For better precision of the WCET estimate, just a high-level analysis at the code level is not sufficient. Low-level analysis using the details of the architecture on which the program is to be run is equally important. Cache memories are one of the most important components in a system at the hardware level. As the processors used in modern real-time systems become faster and faster, the gap between the processor speed and memory speed continues to widen. As a result, almost all real time systems today use cache memories, which provide good access rates, although only for a limited subset of the main memory.

For the purpose of estimation, keeping track of this dynamically changing subset of memory in the cache is crucial, as the difference between access time for memory blocks in the cache to those not in the cache is generally of the order of tens of processor cycles. Moreover, most real-world programs generally spend a significant portion of their execution time in fetching data to/from memory. Hence, taking the pessimistic option of classifying every memory access as a cache miss will significantly blow the estimate. On the other hand, most of the cache replacement algorithms that control the contents of the cache are deterministic, and hence it is possible to safely estimate the cache contents.

In order to further decrease the wide gap between processor speed and main memory speed, modern processors today use multiple levels of cache memory. The higher cache levels provide respectable access rates and at the same time, accomodate more memory blocks, thus preventing the sudden jump from the L1 cache to the main memory. Moreover, the usefulness of multi-level caches increases with the advent of multi-cores, as cores can share a higher level cache to facilitate data sharing and at the same time, have private lower level caches. The higher level caches can also shield lower caches from cache coherency issues.

The Abstract Interpretation based approach for WCET estimation combines analysis at code level and architecture level by abstracting important details of both the code to be analyzed as well as the system on which the program is to run. The approach goes as follows: The first phase is Address Analysis, where we obtain a safe estimate of the set of cache blocks which will be accessed by each instruction in the program. The next step is abstract interpretation based cache analysis, which uses the accessed cache blocks computed by address analysis to determine the worst case execution time of each instruction in the program. In this step, details of the system architecture such as cache levels, associativity, block size, instruction latencies, etc. can be modelled. Using this, we calculate the worst case execution time of each basic block in the program. Then we build an Integer Linear Program(ILP), using the WCET of basic blocks, subject to structural constraints and loop bounds(thus incorporating the program structure), to determine the worst case execution path in the program.

The primary focus of our work is in the Cache Analysis phase. We use the previous work of Rathijit Sen for Address Analysis and ILP formulation[10] and do not make any changes to it. Safety is of paramount importance while estimating WCET of programs for hard(or even soft) real time systems, and Must Analysis is one of the few techniques for cache analysis which has been proven safe theoretically. Since Must Analysis provides guarantees for cache blocks present in the cache across all executions, precision is severely compromised. The issue of precision is more severe in Data cache Must Analysis, because Address Analysis for Data caches is imprecise and frequently gives a non-singleton set of cache blocks accessed by an instruction(especially for instructions inside loops). For such multi-reference accesses, the original Must Analysis does not bring any of the accessed cache blocks to the Must cache, but simply makes the existing cache blocks older. This will result in the Must cache becoming empty even for simple programs with accesses inside loops, where even a quick manual analysis can reveal that the Must cache should not be empty.

We use the May Analysis to tackle this issue. May Analysis determines all cache blocks that may enter the cache under all executions, and this information can be used by the Must Analysis to deduce that some cache blocks must remain in the cache, as there are just not enough younger cache blocks which may force eviction. Another cache analysis which has been widely used for WCET estimation and which does not suffer from the precision issue is Persistence Analysis. Persistence Analysis cannot classify cache accesses as always hit or always miss, but instead classifies accesses inside loops as first miss, which means that the first access to a block may or may not find it in the cache, but all the other accesses to it will definitely be satisfied by the cache. There is a safety issue with the original Persistence Analysis given by [12], and in our work, we pinpoint the reason behind the lack of safety in the Persistence Cache update. We propose to use both Must and May Analysis to rectify the safety issue. Others have identified similar issues with Persistence Analysis, in [4] and [7]. Our solution is more precise than the solutions proposed in both the papers, and we give examples where our approach is able to detect more persistent blocks.

We also propose a new framework for multi-level cache analysis. For single level caches, the Address Analysis provides the set of memory blocks to be accessed by each instruction as an input to the cache analysis phase. However, for multi-level caches, the set of accessed blocks at each level depends upon the contents of the lower level caches, and need to be determined by the cache analysis algorithm itself. In a non-inclusive cache hierarchy with writeback, new memory blocks which are brought in move from higher levels to lower levels, and at the same time, dirty memory blocks evicted by lower lowels move towards higher levels. Hence, there is a flow in both directions, which needs to be taken into account, while maintaining safe estimates and being as precise as possible.

We use three different analyses - Must Analysis, May Analysis and Persistence Analysis to determine the cache contents. As far as we know, this is the first effort for performing Persistence Analysis in Multi-level caches in the presence of writebacks. We extend the classical abstract lattice used for single level caches to the entire cache hierarchy. Following are some of the important features of our framework:

- A definitive version of Multi-level cache analysis which accurately simulates the writeback effect for a write-allocate, write-back, non-inclusive cache hierarchy.
- Takes into account all the optimizations for join in Multi-level Must analysis as pointed out by [11], through the novel use of partial and pseudo blocks in the cache hierarchy.
- Performs both May and Persistence analysis in addition to Must analysis, and provides a safe approach to perform these analyses for Multi-level caches.
- Uses the improved Must analysis and Persistence Analysis which uses the results of May analysis in its transfer function, providing better precision for higher level caches.

Chapter 2

Related Work

Abstract Interpretation is one of the more successful techniques employed for Cache Analysis. The pioneering work was done by [2] and [12], who proposed the abstract lattice for the cache analysis, as well as the transfer functions for Must Analysis, May analysis and Persistence Analysis[5]. Much of the earlier work in this area was limited to Instruction Caches, as address analysis for instruction caches yields precise results[9]. Earlier work on Cache Analysis for data caches concentrated more on finding techniques to make the address analysis more precise[13].

[4] points out the error in the original Persistence Analysis, and they argue that it arises because of mismatched cache update and join function. They propose to use May Analysis to count the total number of cache blocks that may be present in the cache, and deny any evictions from the Persistence cache if this count is less than the associativity. [7] augments the persistence analysis by keeping track of younger sets of all cache blocks that may enter the cache, and uses their cardinalities to perform safe cache update.

[6] proposed a natural extension to the single-level cache analysis of [2] for multi-level instruction caches. They concentrate on the filtering effect of lower level caches on the accesses to the higher level caches. [8] applies the same approach to multi-level Data caches. However, both approaches are limited to a write-no-allocate, write-through, non-inclusive cache hierarchy, and hence ignore the writeback effect. While their approach uses the Must, May and Persistence Analysis, they consider each cache level separately,

and run these analyses to fixpoint independently on each cache level, from lower to higher levels. Our Analysis treats the entire cache hierarchy as a single unit, and the three analyses run to fixpoint for the entire cache hierarchy. This approach has the advantage of improving the precision of the Must Analysis. As we will show later, knowing the state of the entire hierarchy at all times allows us to collaborate information across cache levels. Our analysis also targets a write-allocate, write-back non-inclusive cache hierarchy. [11] proposes a radically different approach to Multi-level Data cache Analysis, where they form pairs of cache levels and track the contents of these pairs which they call 'live caches' to incorporate the writeback effect. In our work, we extend the original, simple cache model used for single-level caches, and using known analysis techniques like Must, May and Persistence Analysis, we are able to solve the issues caused by writeback.

Chapter 3

Cache Analysis Terminology

Caches store fixed size chunks of memory in cache blocks(also called memory blocks or cache lines). All the data transfer to/from the cache takes place in the units of linesize(i.e. size of cache block). Given a memory reference x, $x/linesize$ gives the address of the cache block containing x. Given a cache block address, deciding where in the cache the block will be stored depends upon the cache associativity. In a fully-associative cache, a cache block can be placed anywhere, while in a direct-mapped cache, there is a fixed location for each cache block. Set-associative caches are organized in terms of cache sets, which are collections of cache blocks. In set-associative caches, the cache set containing a cache block is unique, but within the set the cache block can be placed at any location. A cache C_x with total size $capacity_x$, cache block size $linesize_x$ and associativity A_x has $blocks_x = capacity_x/linesize_x$ cache blocks which are distributed in $sets_x = blocks_x/A_x$ sets. A cache block with address $addr_x$ will be present in the set $addr_x\%sets_x$.

In an A-way set associative cache, if a set is full and another cache block needs to be brought into this set, then the cache replacement policy decides which of the A cache blocks needs to be evicted. LRU(Least Recently Used) is one such policy, which selects the cache block that has stayed the longest in the set without any references to it, to be evicted. Temporal locality dictates that such cache blocks have less chances of being referenced, and hence more optimal to being replaced. We will assume that the cache replacement policy is LRU. In each set, we will order the cache blocks by their last accesses, and the position at which the cache block resides will be its age. This has no relation, whatsoever, with the actual physical arrangement in the cache set.

For a multi-level cache hierarchy, the cache inclusion policy is equally important as the cache replacement policy in determining the contents of different levels of the cache. In a strictly inclusive cache hierarchy, every cache level is a strict subset of higher cache levels. Note that lower cache levels are closer to the processor, and as we go up, we get closer to the main memory. To maintain the subset relation, whenever a cache block is brought into a cache level, it is also brought into all lower cache levels. While evicting a cache block from a cache level, it is also evicted from all lower cache levels(if it is present in the lower levels). If this eviction policy is relaxed and evictions across levels are allowed to be independent of each other, the cache hierarchy is called non-inclusive(or partially inclusive). Hence, non-inclusive cache hierarchies allow lower level caches to contain cache blocks which are not present in higher level caches.

In a non-inclusive cache hierarchy, on a memory access from the processor, cache levels are searched in increasing order starting from L_1 cache. A miss at cache level L_x brings the accessed cache block to level L_x(and also makes it the most recently accessed block at level x). A hit at level L_x makes the accessed block the most recently accessed at level x, and cache levels above level x are not touched.

Generally writes to cache blocks are treated differently from reads. If a cache hierarchy follows a writethrough, write-no-allocate policy then for a write miss, the cache block being written is not even brought into the cache, and the change is made directly to the main memory. For a write hit, the change is still propagated all the way to the main memory. Hence, every write request suffers main memory latency, irrespective of whether the requested cache block is present in the cache or not. In the writeback, write-allocate policy, writes are treated identically as reads, except that a write to a cache block marks it as *dirty*. That is, the cache block being written will become the most recently accessed cache block (for a write miss, it would be brought into the cache as well). On eviction, a dirty cache block is not simply thrown out, but is instead written to the next cache level, and this write to the next level is treated as a normal processor write. This means that this evicted cache block written to the next level is now the most recently accessed block at that level. On eviction from the highest cache level, the dirty cache block is finally written to the main memory.

To summarize, in a non-inclusive cache hierarchy with writeback, write-allocate policy, on a write request from the processor, cache levels are searched in increasing order starting from L_1 cache. A write miss at cache level L_x brings the accessed cache block to the level L_x in a dirty state(also making it the most recently accessed block). A write hit at level L_y makes the accessed block dirty as well as most recently accessed. When a cache block marked as dirty is evicted from level L_x, it is brought into level L_{x+1}, marked as dirty as well as most recently accessed. In our Analysis, we will assume a cache hierarchy with these properties.

Abstract interpretation[3] is a static program analysis technique which formalizes the data flow analyses used in Compilers. For WCET analysis, safety is one of the paramount requirements, and abstract interpretation provides a method for formally proving the safety of our analysis. For using abstract interpretation, one needs to specify the concrete lattice, which is generally the power set lattice of program property of interest. In our case, we are interested in the entire state of the cache hierarchy-i.e. the cache blocks inside each level, their ages and their dirty-states. Note that the main memory is finite, and at all times, the cache hierarchy contains only a subset of the main memory. Hence, the number of cache hierarchy states is also finite. We extend the concrete lattice specified in [2] to multiple levels. Below, we define the concrete states formally:

A cache hierarchy F is the set $F = \{F_1, \dots, F_n\}$, where F_x are cache levels. Each cache level F_x is the set $F_x = \{f_{1,x}, f_{2,x}, \dots, f_{sets_x,x}\}$, where $f_{i,x}$ are the cache sets in the xth cache level. Each cache set $f_{i,x}$ is the set $f_{i,x} = \{l_{i,x}^1, l_{i,x}^2, \dots, l_{i,x}^{A_x}\}$. Note that the line number also signifies the age of the memory block present in that line. Hence, $l_{i,x}^1$ contains the most recently used cache block in the ith set at xth level. Let M be the size of the main memory(in bytes), and let M_x be the main memory as 'viewed' from the xth cache level, i.e., $M_x = \{m_1, \dots, m_{M/linesize_x}\}$, where m_i are memory blocks of size

$linesize_x$.

Definition *Concrete Set State* is a function $s_{i,x} : f_{i,x} \rightarrow M_x \cup \{\perp\}$, where $\perp$ signifies the empty memory block. Let $S_{i,x}$ be the set of all such functions.

Definition *Concrete Cache State* is a function $c_x : F_x \rightarrow \bigcup_{i=1}^{sets_x} S_{i,x},\ such\ that\ c_x(f_{i,x}) \in S_{i,x}, \forall 1 \leq i \leq sets_x$. Let C_x be the set of all such functions.

Definition *Concrete Cache Hierarchy State* is a function $h : F \rightarrow \bigcup_{x=1}^{n} C_x$, such that $h(F_x) \in C_x$. Let H be the set of all concrete cache Hierarchy States.

Each element of the concrete lattice is a subset of H. The concrete transfer function for this lattice takes a concrete cache hierarchy state and a memory reference, and outputs another concrete cache hierarchy state. This transfer function depends on the inclusion policy of the cache hierarchy and the nature of writes to the cache(i.e. writeback or writethrough), and basically mimics the actual cache update. For the abstract lattice, we have the following definitions:

Definition *Abstract Set State* is a function $\hat{s}_{i,x} : f_{i,x} \rightarrow 2^{M_x \cup \{\perp\}}$. In an abstract set state, we allow a cache line to contain multiple cache blocks. Again, let $\hat{S}_x$ be the set of all such functions.

Definition *Abstract Cache State* is a function $\hat{c}_x : F_x \rightarrow \bigcup_{i=1}^{sets_x} \hat{S}_{i,x},\ such\ that\ \hat{c}_x(f_{i,x}) \in \hat{S}_{i,x}, \forall 1 \leq i \leq sets_x$. Let $\hat{C}_x$ be the set of all such functions.

Definition *Abstract Cache Hierarchy State* is a function $\hat{h} : F \rightarrow \bigcup_{x=1}^{n} \hat{C}_x$, such that $\hat{h}(F_x) \in \hat{C}_x$. Let $\hat{H}$ be the set of all concrete cache Hierarchy States.

The following three analyses use the above mentioned definitions.
The *Must Analysis* produces an abstract hierarchy state at each program point such that every concrete hierarchy state possible at the program point contains all the memory blocks present in the abstract state, and the age of a memory block in the abstract state is an upper bound on the age of the block in all concrete states. Intuitively, the Must

analysis provides those cache blocks that are guaranteed to be present in the cache, and thus accesses to such blocks can be classified as always hit.

The *May Analysis* produces an abstract hierarchy state at each program point such that no concrete hierarchy state possible at the program point can contain a memory block not present in the abstract state. Hence, the abstract hierarchy state is in some sense, a superset of all concrete hierarchy states possible at the program point under all executions. Also, the age of a memory block in the abstract state is a lower bound on the age of the memory block in all concrete states. Intuitively, the May analysis provides those cache blocks that may enter the cache along some execution path. Hence, if a cache block is not present in the abstract state produced by May analysis, access to such a block can be safely classified as always miss.

Persistence Analysis produces an abstract state similar to the May analysis, but with the property that the age of a memory block in the abstract state is an upper bound on the age of the memory block in all concrete states. To indicate that a memory block in cache level x can be evicted, in which case its age would be $A_x + 1$, a special eviction line is added to each set, which contains those memory blocks which may have been evicted. Persistence Analysis is used to identify those cache blocks that are persistent. A cache block is persistent, if once it is brought into the cache, it is not evicted. Cache blocks in the persistence cache which are not in the eviction line are persistent. Persistence Analysis is used to classify references in loops as first miss(i.e. miss on first iteration, hits on all other iterations).

Chapter 4

Improved Must Analysis

Must analysis is the most important of the three analyses from the perspective of obtaining safe estimates, because an access classified as hit from the Must cache is guaranteed to be hit in the actual cache for all executions. This imposes a stringent safety requirement on the analysis itself. To satisfy this safety requirement, the precision of the analysis is severely compromised. This precision issue is further exacerbated in Data cache Must Analysis, because of the imprecise results of Address Analysis.

For the following discussion, assume that we are dealing with a single level cache. Hence, f_i denotes that ith set in the cache, while l_i^a denotes the cache line in this set with age a. If the cache block accessed by an instruction is precisely known(i.e single-reference access), then the transfer function of Must Analysis is similar to the actual LRU update of a normal cache. Given the accessed cache block, the set f_i containing it can be determined. The abstract set state $\hat{s}_i(f_i)$ of the Must cache is modified by bringing the accessed cache block to the first position(i.e. at l_i^1). If the accessed block was already present in the set state at position h, then the younger cache blocks are shifted by one position. If the accessed block was not present, then all the cache blocks in the set state are shifted, evicting the oldest referenced cache blocks(i.e. those in l_i^A).

On the other hand, for multi-reference accesses, the transfer function is not as precise as the actual LRU update. For such an access, the address analysis gives a set of cache blocks $X = \{m_1, \ldots, m_l\}$, which can be accessed by the instruction. Since the exact cache block which is accessed is not known, the transfer function does not bring any of the accessed cache blocks to the must cache. At the same time, any of the cache blocks in X can be accessed, and hence they can all contribute to the aging of the cache blocks already present in the Must cache. To simulate this aging effect, for cache blocks in the Must cache at position h, we count the number of accessed cache blocks which have an age greater than h, or are not present at all in the must cache. Let $X_i = \{m_1, \ldots, m_{l_i}\}$ be the cache blocks in X which map to the set f_i. We define the function $shiftctr(X_i, h) = |\{m \in X_i | (\exists a, h < a \leq A,\ m \in \hat{s}_i(l_i^a)) \vee (\forall a, 1 \leq a \leq A,\ m \notin \hat{s}_i(l_i^a))\}|$. $shiftctr(X_i, h)$ gives the worst case increase in the age of cache blocks at position h due to the access X. Since the actual access can only increase the age of a cache block by 1, if $shiftctr(X_i, h) \geq 1$, we increase the age of cache blocks at position h by 1.

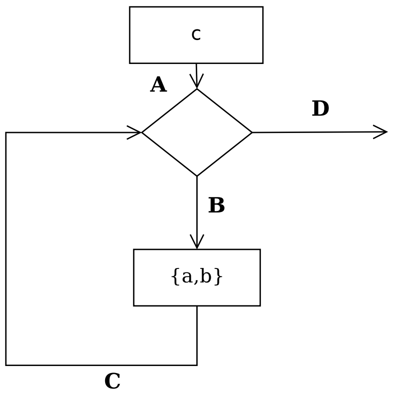

Figure 4.1: Imprecision of Original Must Analysis

While the above transfer function is safe, it suffers from lack of precision. Consider the loop represented by the CFG in the Figure 4.1. a, b, c are cache blocks which map to the same set, and the cache has an associativity of 4. Before entering the loop for the first time, the cache block c is already present in the Must cache with age 1 at program point A. Since the access inside the loop is multi-reference, cache blocks a and b will not be brought into the must cache after the access. Also, at program point B, $shiftctr(\{a, b\}, 1) = 2$, since both a and b are not present in the cache. Hence the cache block c will be shifted to position 2 in the must cache at program point C. Join in the Must Analysis is intersection of cache blocks in the corresponding set states, while taking the maximum age. Hence, in the next fixed point iteration, the join of the must caches at program points A and C will result in in the must cache with the cache block c in position 2 at program point B. Again $shiftctr(\{a, b\}, 3) = 2$, hence the cache block c will now have an age of 3 at program point c. Eventually, it will be evicted from the Must cache, resulting in an empty set state in the Must cache at program point C, and subsequently at program points B and D.

Since there are only three cache blocks involved in the loop, the cache block c will never be evicted during any actual execution. Hence, while Must Analysis give a safe estimate, it can be made more precise by including the cache block c in the Must cache at program point D. The key observation here is that at C, there can be maximum of two cache blocks which are younger than cache block c. This information can be captured using May Analysis.

Consider a cache block m at position h in the Must cache at some program point P. Now, consider the May cache at program point P. The May cache will also contain the cache block m, and it will be present with an age less than or equal to h. Now, consider all the cache blocks at positions less than or equal to h in the May cache at P. It can be argued that these cache blocks comprise the entire set of cache blocks which can be *younger* than the cache block m under any actual execution. No cache block that may be present in the cache at P will be missed by the May Analysis, and cache blocks with age greater than h in the May cache will never have an age less than h at program point P, since May cache also maintains the lower bound on ages. Hence, if a cache block was younger than m at P along some execution path, it will be captured by the May Analysis. Now if the instruction following program point P accesses a set of memory blocks X, then the memory blocks in X along with all the memory blocks at positions less than or equal to h in the May cache at P will be the maximum number of cache blocks that can be younger than m at the program point Q.

Let $MaxYoung(X_i, h) = |X_i \cup \bigsqcup_{a=1}^{h}(\hat{s}_i^{May}(l_i^a) - X_i)|$. Note that $\hat{s}_i^{May}$ indicates the ith cache set of the May cache. $MaxYoung(X_i, h)$ gives the maximum number of memory blocks which will be younger than a memory block present at position h in the

Must cache, after the access X_i. We now take the minimum of $h + shiftctr(X_i, h)$ and $MaxYoung(X_i, h)$, and if it is greater than h, we increase the age of cache blocks at position h in the Must cache by 1. In the example, after the cache block c moves to position 3 in the Must cache at program point B, for the next access, $Min(3 + shiftctr(\{a, b\}, 3), MaxYoung(\{a, b\}, 3)) = Min(5, 3) = 3$. Hence the cache block c will remain at position 3 in the Must cache after the access.

Let $NewPos(X_i, h)$ be $h+1$ if $Min(h + shiftctr(X_i, h), MaxYoung(X_i, h)) > h$, or h otherwise. Formally, the transfer function of the improved Must Analysis can be given as follows:

$$\hat{U}^{\hat{C}}_{Must}(\hat{c}, X) = \hat{c'}$$

$$where\ \forall i, 1 \leq i \leq sets, \hat{c'}(f_i) = \hat{U}^{\hat{S_i}}_{Must}(\hat{c}(f_i), X_i)$$

If $X = \{m\}$ is singleton, then

$$\hat{U}^{S_i}_{Must}(\hat{s}_i, X_i) = \begin{cases} \hat{s}_i, \\ \quad if\ X_i\ is\ empty \\ l_i^1 \mapsto \{m\} \\ l_i^a \mapsto \hat{s}_i(l_i^{a-1}), 2 \leq a \leq h-1 \\ l_i^h \mapsto (\hat{s}_i(l_i^h) - m) \cup s_i(l_i^{\hat{h}-1}) \\ l_i^b \mapsto \hat{s}_i(l_i^b), h+1 \leq b \leq A \\ \quad if \exists h, 1 \leq h \leq A, such\ that\ m \in \hat{s}_i(l_i^h) \\ l_i^1 \mapsto \{m\} \\ l_i^a \mapsto \hat{s}_i(l_i^{a-1}), 2 \leq a \leq A \\ \quad otherwise \end{cases}$$

If X is non-singleton, then

$$\hat{U}^{S_i}_{Must}(\hat{s}_i, X_i) = \begin{cases} \hat{s}_i, \\ \quad if\ X_i\ is\ empty \\ l_i^a \mapsto \bigsqcup_{\substack{b \\ b+NewPos(X_i,b)=a}} \hat{s}_i(l_i^b), 1 \leq a \leq A \\ \quad otherwise \end{cases}$$

Note that the notation $l \mapsto A$ indicates that the cache line l will now be mapped to set A in the new abstract set state. The important difference between the original Must analysis and the Improved Must analysis is that the original analysis uses only the shiftctr function, while we also use the MaxYoung function. However, note that the NewPos function will always be less than or equal to the shiftctr function. This means that if a cache block is not evicted from the Must cache by the original Must analysis(which would happen if $shiftctr < A + 1$), it will not be evicted by the improved Must analysis as well. Hence, the improved must analysis is at least as precise as the original Must analysis. The upper bound on ages computed by the improved Must analysis will always be less than or equal to those computed by the original Must Analysis.

Chapter 5

May Analysis

The Abstract lattice for May analysis is simply the set of all abstract cache hierarchy states, i.e. $\hat{H}$. The join is a level-by-level, set-by-set union of all the corresponding abstract set states, while taking the minimum age. Below, we present a slight reformulation of the original transfer function for May analysis. While it achieves the same effect as the original transfer function, it can be more efficiently computed, especially for multi-reference accesses. For a multi-reference accesses consisting of n accessed cache blocks, the original transfer function would perform n single-reference access updates, and then perform a join of the n updated cache hierarchies. Our transfer function inherently handles multi-reference accesses, and hence would require only one update and no joins.

For a single reference access to level x, the transfer function $\hat{U}^{\hat{C}_x}_{May}$ takes the input cache state and the accessed cache block and returns a new cache state. It simply applies the transfer function for abstract sets, $\hat{U}^{\hat{S}_x}_{May}$ to the set which contains the accessed cache block, and keeps the rest of the sets unchanged. Below, the notation $l \mapsto m$ indicates that after the update, the function will now map l to m.

$$\hat{U}^{\hat{S}_x}_{May}(\hat{s}_{i,x}, m) = \begin{cases} l^1_{i,x} \mapsto m, \\ l^a_{i,x} \mapsto \hat{s}_{i,x}(l^{a-1}_{i,x}),\ 2 \leq a \leq h \\ l^{h+1}_{i,x} \mapsto \hat{s}_{i,x}(l^{h+1}_{i,x}) \cup (\hat{s}_{i,x}(l^h_{i,x}) - m), \\ l^b_{i,x} \mapsto \hat{s}_{i,x}(l^b_{i,x}),\ h+2 \leq b \leq A_x; \\ \qquad if \exists h, 1 \leq h \leq A_x, m \in \hat{s}_{i,x}(l^h_{i,x}), \\ l^1_{i,x} \mapsto m, \\ l^a_{i,x} \mapsto \hat{s}_{i,x}(l^{a-1}_{i,x}),\ 2 \leq a \leq A_x; \\ \qquad otherwise \end{cases}$$

The transfer function for the abstract set brings accessed cache block to the first position in set. If the block m was already present in the set, then the ages of all cache blocks who were accessed recently relative to m would be increased by 1. This also includes those cache blocks who may have an age same as that of m.

This transfer function takes as input only one accessed cache block, but as stated earlier, address analysis for data caches is imprecise, and hence we may have to update the abstract cache with a set of accessed cache blocks, with the property that the actual access will to any element of this set. All the accessed cache blocks may not map to the same cache set. In such a scenario, since we do not know which cache set will actually be accessed, we cannot increase the age of any cache block in any of the cache sets, as May analysis needs to maintain the lower bound of ages. As a result, we simply bring all the accessed cache blocks to the first position in their respective cache sets.

In the scenario where all the accessed cache blocks map to the same set, we can consider the cache block among those accessed which has the minimum age, and safely increase the ages of all the cache blocks in this cache set having age less than the 'youngest' accessed cache block. Formally the transfer function is:

$$\hat{U}^{\hat{S}_x}_{May}(\hat{s}_{i,x}, \{m_1, \ldots, m_p\})$$

$$= \begin{cases} l_{i,x}^{1} \mapsto \{m_1, \dots, m_p\}, \\ l_{i,x}^{a} \mapsto \hat{s}_{i,x}(l_{i,x}^{a-1}),\ 2 \leq a \leq h \\ l_{i,x}^{h+1} \mapsto (\hat{s}_{i,x}(l_{i,x}^{h+1}) \cup \hat{s}_{i,x}(l_{i,x}^{h})) - \{m_1, \dots, m_p\}, \\ l_{i,x}^{b} \mapsto \hat{s}_{i,x}(l_{i,x}^{b}) - \{m_1, \dots, m_p\},\ h+2 \leq b \leq A_x; \\ \quad if \exists g, 1 \leq g \leq A_x, \exists j, 1 \leq j \leq p, \\ \quad m_j \in \hat{s}_{i,x}(l_{i,x}^{g})\ and\ h\ is\ the\ minimum\ of\ all\ such\ g \\ l_{i,x}^{1} \mapsto \{m_1, \dots, m_p\}, \\ l_{i,x}^{a} \mapsto \hat{s}_{i,x}(l_{i,x}^{a-1}) - \{m_1, \dots, m_p\},\ 2 \leq a \leq A_x; \\ \quad otherwise \end{cases}$$

Chapter 6

Improved Persistence Analysis

Persistence Analysis is used to determine the upper bound on the ages of all cache blocks that may enter the cache. Similar to May analysis, if there is any execution path along which a particular cache block enters the actual cache, then the Persistence cache must include this cache block. If the cache block has different ages along different paths, then Persistence analysis must determine the maximum age. To indicate that the cache block may also have been evicted, a special eviction line $l_i^{\top}$ is added to each set state.

Apart from this special eviction line, the abstract lattice for the original Persistence analysis is same as that of May analysis. The join in this lattice also does a cache set-by-set union, except that it takes the maximum age of a cache block, if it is present in both cache sets.

The transfer function for persistence analysis, as given in [12] is similar to that of Must analysis. For single-reference accesses, it brings the accessed cache block to the first position in the set state. If the cache block was already present in the cache, then the ages of all cache blocks younger than the accessed cache block will be increased. The rest of the cache blocks retain the same age, including the cache blocks in the special eviction line. Finally, if the cache block was not present in the cache, the age of all cache blocks(except those in the eviction line) are increased. The newly evicted cache blocks are simply added to the eviction line, which retains the cache blocks present in it before the update.

Since the join used in Persistence Analysis is cache set union, but the transfer function is similar to Must Analysis, the objective of Persistence Analysis–to maintain an upper bound on ages of all cache blocks–is not achieved. Consider a cache block m present in the persistence cache at a Program point P, with age $h \leq A$. Since the join used by Persistence Analysis is set union, m may not be present at program point P in the actual cache along some execution path. Then along such a path, an access to m at P will contribute to an increase in the age of all cache blocks present in the actual cache. However, since the persistence cache contains m, an access will only increase the age of those cache blocks which are younger than m. Hence the upper bound on ages computed by the Persistence Analysis for those cache blocks which have higher ages than m will not be correct.

This suggests that while doing cache update, we must only consider those cache blocks which are guaranteed to be present in the cache, and use these cache blocks to decide the new ages of cache blocks in the persistence cache. The Must analysis precisely computes the set of cache blocks that must be in the cache at a program point, and it runs independent of the persistence analysis. However, Must Analysis suffers from lack of precision since the join used is set intersection. Hence, just relying on Must Analysis will give safe but imprecise results. The upper bound on the age of cache block m at a program point must itself be upper bounded by the maximum number of younger cache blocks than m that enter the cache along all execution paths. May Analysis can be used to determine this upper bound, similar to the improved Must Analysis.

Using the contents of the abstract cache maintained by the Must analysis and the May analysis, we propose the following transfer function for Persistence Analysis:

We will present the transfer function for a general multiple-reference access to level x. The transfer function for the abstract cache state, $U_{Per}^{\hat{C}_x}$, applies the transfer function for abstract set state, $U_{Per}^{\hat{S}_x}$ to all sets, which takes as input the abstract set state and the accessed cache blocks mapping to the set. Let $\hat{c}_x^{Must}$ and $\hat{c}_x^{May}$ be Must and May caches respectively at the program point, and let $\hat{s}_{i,x}^{Must} = \hat{c}_x^{Must}(f_{i,x})$, $\hat{s}_{i,x}^{May} = \hat{c}_x^{May}(f_{i,x})$ $1 \leq i \leq sets_x$. Let $X_i = \{m_1, \ldots, m_{l_i}\}$ be the cache blocks in the incoming access, mapped to set i.

We use the function $shiftctr(X_i, h) = |\{m \in X_i | (\exists a, h < a \leq A_x,\ m \in \hat{s}^{Must}_{i,x}(l^a_{i,x})) \vee (\not\exists a, 1 \leq a \leq A_x,\ m \in \hat{s}^{Must}_{i,x}(l^a_{i,x}))\}|, 1 \leq h \leq A_x$. Note that shiftctr function uses the contents of the Must cache to determine the increase in age of cache blocks in the Persistence cache. For a cache block with age h in the Persistence cache, all the accessed cache blocks which are either not present in the Must cache, or which are older(i.e have age greater than h) can contribute to its aging. Shiftctr function exactly counts such cache blocks, and thus gives the worst case increase in the ages of blocks in position h due to the access X_i.

Let $MaxYoung(X_i, h) = |X_i \cup \bigsqcup_{a=1}^{h} \hat{s}^{May}_{i,x}(l^a_{i,x}) - X_i|$. $MaxYoung(X_i, h)$ gives the maximum number of memory blocks which will be younger than a memory block present at position h in the Persistence cache, after the access X_i.

The transfer function must now shift the cache blocks in position h to $h+1$ if $Min(h + shiftctr(X_i, h), MaxYoung(X_i, h))$ is greater than h. Otherwise the cache blocks will remain at position h (the new position is designated as $NewPos(X_i, h)$). If the access is single-reference(i.e. X_i is singleton), then the accessed cache block will be brought in the first position in the persistence cache and the rest of cache blocks will follow the above rule. However, if the incoming access is multi-reference, then all the accessed blocks cannot be brought into the first position. Let X_i' be the set of memory blocks in X_i where are not present in the set $\hat{s}_{i,x}$ and let $z = |X_i'|$. Then, the cache blocks in X_i' will be brought into position z in the persistence cache. Among the memory blocks that are referenced in X_i and also present in the persistence cache, we cannot decrease their relative ages and their new ages will be determined using the NewPos rule.

$$U^{\hat{S}_x}_{Per}(\hat{s}_{i,x}, X_i)$$

$$= \begin{cases} \begin{cases} l^a_{i,x} \mapsto \bot, \quad 1 \le a < z \\ l^z_{i,x} \mapsto X'_i \\ l^a_{i,x} \mapsto \bigsqcup_{b \le a \ \wedge \ NewPos_i(b)=a}^{b,} \hat{s}_{i,x}(l^b_{i,x}), \quad z < a \le A_x \end{cases} & if\ z \le A_x \\ \begin{cases} l^a_{i,x} \mapsto \bot, \quad 1 \le a \le A_x \\ l^\bot_{i,x} \mapsto X'_i \cup \bigsqcup_{a=1}^{A_x} \hat{s}_{i,x}(l^a_{i,x}) \end{cases} & otherwise \end{cases}$$

The problem with the original Persistence Analysis as well as approaches to overcome it have been proposed in [4] and [7]. Cullmann's approach[4] uses May analysis to count the total number of cache blocks that can be present in the cache at a program point, and then depending upon whether this number is greater than the cache associativity, it evicts the cache blocks with age A in the Persistence cache. This approach is imprecise because it always increases the age of all cache blocks in the Persistence cache irrespective of whether the accessed blocks are already present or not.

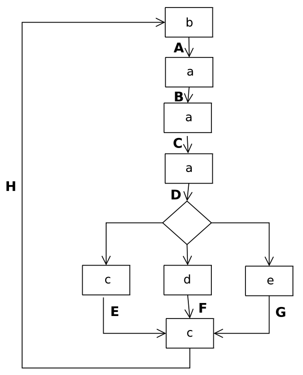

Figure 6.1: Imprecision of Cullmann's Analysis

For the program represented by the CFG shown in Figure 6.1, assume that cache blocks a, b, c, d, e all map to the same cache set, and the cache associativity is 4. At the program point B, the block a will be present in the persistence cache, and will also be the youngest. Hence, the next access to the same block should not age any other cache blocks in the Persistence cache. However, Cullmann's analysis continues to increase the age of all cache blocks in the Persistence cache at any access. Hence, at program point B, block b will have an age of 2, at C, it will have an age of 3, and finally at point D, block b will have an age of 4. After the accesses to c, d and e, the total number of cache blocks in the May cache would become 5, resulting in eviction of the cache block b at the next access to block c by Cullmann's analysis. Hence, cache block b is classified as non-persistent by Cullmann's Analysis. However, in the actual cache, the cache block b will never be evicted during any execution. This is because at the program point D, block b will have a maximum age of 2, and the next two accesses(the first access being either c, d, or e and the second access c) will at most increase its age by 2, leaving b with a maximum age of 4 at program point H. Since the Must caches at program points B, C and D will contain the cache block a at the first position, our analysis will not increase the age of any cache blocks in the persistence cache at those points. Hence, our analysis will be able to capture the correct upper bound on the cache block b at all program points, and declare b as persistent. Along with its imprecision, Cullmann's Analysis also seems incapable of handling multi-reference accesses.

The approach proposed in [7] augments the original persistence analysis by also calculating the younger set(i.e. the set of younger cache blocks) for every cache block present in the Persistence cache. While similar to our approach in spirit, there are two important advantages of our approach. First, we use the May Analysis to calculate the younger set of each cache block. This is an elegant and much more efficient way of calculating the younger set. Their approach separately maintains a younger set of each cache block that may enter the cache, along with the persistence cache, which results in a lot of duplication. As an example, when a cache block is accessed, it is added to the younger set of every cache block that is present in the Persistence cache, whereas in our approach, it would simple appear once in the May cache.

More importantly, their approach only considers the cardinality of the younger set while updating the age of cache blocks, while ignoring the contents of the Must cache. This affects the precision of Persistence Analysis, and cache blocks that are actually persistent can be missed by their analysis.

Consider Figure 6.2, depicting part of the CFG of a program. Let the associativity of the cache be 2, and assume that cache blocks a, b, c map to the same cache set. After

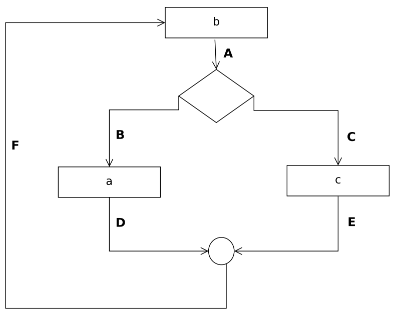

Figure 6.2: Imprecision of [7]'s Analysis

the join, at program point F, the younger set of cache block b would contain both a and c, and hence an age update solely based on younger set would conclude that b could have been evicted, when program point F is reached(The age of a cache block is the cardinality of the younger set + 1). However, in our analysis, at program points D and E, the cache block b would be at position 2. This is because at points B and C, cache block b would be at position 1 in the persistence cache and the must cache(as it has just been accessed), while the cache blocks a and c would not be present in the Must cache. Hence $shiftctr(\{a\}, 1) = shiftctr(\{c\}, 1) = 1$, which would mean that the cache block at position 1 in the Persistence cache (which would be the block b), would be moved to position 2. Join in our Persistence Analysis is just set union, while taking the maximum of ages. Since block b is at age 2 in both the persistence caches at D and E, it would remain at position 2 in the persistence cache at F. This analysis is more precise, because the cache block b would never be evicted during any actual execution, and hence is persistent. Note that $MaxYoung(\{a\}, 1) = MaxYoung(\{c\}, 1) = 2$ as well, since only the cache block b will be at position 1 in the May cache at program points B and C.

Chapter 7

Multi-level Cache Analysis

7.1 Motivation

The basic strategy for Multi-level cache analysis is to identify the cache blocks that are guaranteed to be present in the cache, for each cache level. Once we have this information, we can determine the worst case execution time of each memory-accessing instruction, which will just be the access latency of the first cache level L_x, which satisfies the condition that all memory blocks accessed by the instruction are present in levels L_1 through L_x. If no such level exists, then it will be the main memory access latency.

Must analysis at each cache level should be sufficient to accomplish this goal. However, as stated earlier, the contents of level L_x determines which blocks will be referenced at level L_{x+1}(except for the first level). The access pattern across levels also depends on the inclusion policy and writeback property. We assume a non-inclusive cache hierarchy with writeback, write-allocate. We will show that May and Persistence Analysis are required at all levels for such a cache hierarchy. The following interdependencies between different analyses can be observed:

Must Analysis at level $x + 1$ depends on May Analysis at level x, because of the non-inclusive property of the cache hierarchy. Consider a single-reference access to the L_x cache. If the L_x Must cache does not contain the accessed block, then mimicking the actual cache hierarchy update, the transfer function would bring the accessed cache block to the Must cache and then forward the access to the next level. The transfer function at the next level will bring the accessed block to L_{x+1} Must cache. Now, if the accessed block is present in the L_x May cache, then during some execution of the program, the accessed block will be present in the actual cache at level x. In such a case, the access itself will not go to the level $x + 1$, during this execution. Due to the non-inclusion property, the accessed block may or may not be present in the $x + 1$ level in the actual cache. This violates the safety of the Must Analysis at the $x + 1$ level, as we have already brought the accessed block in the L_{x+1} Must Cache.

To safely bring new cache blocks in the L_{x+1} Must Cache, we define $Access_{x+1}^{Must}$ to be the set of accesses that are guaranteed to reach level $x + 1$. $Access_{x+1}^{Must}$ can be determined from $Access_x^{Must}$ by removing the accesses which are satisfied by L_x May Cache. $Access_1^{Must}$ is initialized to the access given by Address Analysis, in case of a single-reference access, or empty in case of a multi-reference access. $Access_x^{Must}$ is defined separately for each program point, for all cache levels x. Note that even for a strictly inclusive cache hierarchy, May Analysis is required to calculate $Access_x^{Must}$ at each level. This is because Must Analysis also needs to maintain the upper bound on the age of cache blocks. Only the accessed cache blocks in $Access_x^{Must}$ can be brought to the first position in the L_x Must cache.

Must Analysis at level $x + 1$ depends on Persistence Analysis at level x, because of the write-allocate policy. The writeback, write-allocate policy brings the evicted dirty cache block from cache level x to $x+1$, and also considers this dirty block as the most recently accessed block in L_{x+1}. For the abstract caches, this means that dirty blocks evicted from L_x also increase the age of the cache blocks in L_{x+1}. Hence, we need to determine maximum number of dirty blocks which can be evicted from the xth level, at each program point, across all executions. $evicted_x^{May}$ contains all dirty blocks that may get evicted from level x. Again, this set is calculated separately for each program point, and for each level x. Also, none of the cache blocks in $evicted_x^{May}$ can actually be brought into L_{x+1} Must cache, as these cache blocks are not guaranteed to be evicted. Instead, we only increase the ages of the existing cache blocks in the L_{x+1} Must cache using the phantom access of $evicted_x^{May}$.

The need to determine $evicted_x^{May}$ has also been mentioned in [2], and they suggested the following approach for its determination : Given an access, the abstract Must Cache, and the abstract May cache, consider the set of all dirty blocks in the May cache before the access(i.e. before applying the transfer function), and remove the dirty blocks from this set which are also present in the Must cache after the access. The dirty blocks that are still present in the Must cache after the access are guaranteed to avoid eviction, and hence, they can be safely removed, while the rest of dirty blocks from the May cache will constitute a writeback to the new level.

This approach is pessimistic, because May cache maintains the lower bound on age of all cache blocks, and hence to assume that all dirty blocks in May cache will get evicted is imprecise. On the other hand, Persistence Cache maintains the upper bound on age of all cache blocks, and a cache block in the Persistence cache which is not in the eviction line is guaranteed to avoid eviction. Hence, the set of newly evicted dirty cache blocks in the L_x Persistence cache after the access is precisely the set $evicted_x^{May}$. Finally, note that if the write policy is write-no-allocate, then Must analysis at level $x + 1$ does not require $evicted_x^{May}$. This is because dirty blocks in $evicted_x^{May}$ are not actually brought into the L_{x+1} Must cache, but are only used in increasing the ages of cache blocks already present. In the write-no-allocate policy, evicted cache blocks do not increase the ages of cache blocks already present.

The information about which cache blocks are dirty is maintained separately by the three analyses at each level. Must cache maintains 'must dirty' information, which means if a cache block is dirty in the Must cache, it will be always be dirty in the actual cache across all executions. May and Persistence analysis maintain 'may dirty' information, which implies a dirty cache block in these caches may or may not be dirty in the actual cache. For safety, the persistence cache will be used to determine which cache blocks are dirty from the newly evicted cache blocks, and only the dirty cache blocks will be present in $evicted_x^{May}$.

May Analysis and Persistence Analysis at level $x + 1$ depends on Must

Analysis at level x, because of the filtering effect. This is easy to see, because for carrying out May Analysis at level $x+1$, we require $Access_{x+1}^{May}$, which consists of accesses that may reach level $x+1$ across all executions. The L_{x+1} May cache needs to contain all cache blocks that may enter the actual cache at level $x+1$, for which it needs $Access_{x+1}^{May}$. The accesses that are satisfied by the L_x Must cache will never reach level $x+1$. Hence, $Access_{x+1}^{May}$ can be determined from $Access_{x}^{May}$ by removing such accesses. Again, $Access_{1}^{May}$ is initialized to the set of accesses calculated by the Address Analysis. Note that since May Analysis also needs to maintain the lower bound on the ages of cache blocks, $Access_{x+1}^{May}$ cannot be used to increase ages of cache blocks already present in the L_{x+1} May cache. Instead $Accesses_{x+1}^{Must}$ will be used to increase the ages of such cache blocks. Determining $Accesses_{x+1}^{Must}$ requires May analysis at level x, which shows that May Analysis at level $x+1$ also depends on May Analysis at level x. In any inclusive cache hierarchy, cache levels are always searched from bottom to top, and the search stops at the first level which satisfies the access. This results in the so-called filtering effect, whereby accesses not satisfied by the May cache at a level are guaranteed to reach the subsequent levels, while accesses satisfied by the Must cache at a level are guaranteed to never reach the subsequent levels.

Persistence Analysis at level $x+1$ also requires $Access_{x+1}^{May}$, since Persistence Analysis should also capture all cache blocks that may enter the actual cache during some execution. Also, since Persistence Analysis also needs to maintain the upper bound on ages of cache blocks, $Access_{x+1}^{May}$ itself will be used to increase the ages of cache blocks already present in the L_{x+1} persistence cache.

May Analysis at level $x+1$ depends on Persistence Analysis at level x, because of the writeback policy. The evicted dirty cache blocks from level x are written to level $x+1$ because of the writeback policy. Hence, May Analysis at level $x+1$ requires $evicted_{x}^{May}$, which requires Persistence Analysis at level x. Unlike Must Analysis, these dirty cache blocks will actually be brought into the L_{x+1} May cache.

7.2 Improved Join for Multi-level Must Analysis

For single level caches, the abstract lattice for Must analysis is the set of all abstract cache states. The join of two abstract cache states is simply an abstract set-by-set intersection, while taking the maximum of the ages of a cache block in the two abstract set states. For multi level caches, the abstract lattice for Must Analysis would be the set of all abstract cache hierarchy states. An extension of the join of two abstract cache hierarchies to a level-by-level, set-by-set intersection, however, is imprecise.

We assume that $linesize_1 \leq linesize_2 \leq \cdots \leq linesize_n$. We also assume that $linesize_x$, $1 \leq x \leq n$ is a multiple of $linesize_1$. Then, we can consider any normal cache block in level x as a set of $r_x = linesize_x / linesize_1$ cache blocks from M_1 (Remember that M_1 is the main memory as 'viewed' from the L1 cache level). Let us call memory blocks in M_1 as basic cache blocks. When we say that cache block m_x at level x is a subset of m_y at level y, what we mean is that all the basic cache blocks in m_x are present in m_y.

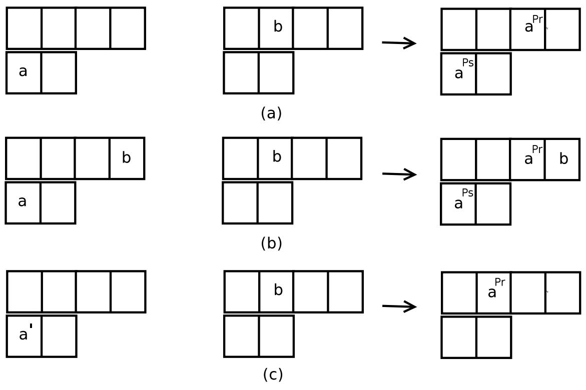

Figure 7.1: Partial Block Generation

Consider two abstract cache hierarchy states $\hat{h}$ and $\hat{h}'$. Let $\hat{h}(F_x) = \hat{c}_x$ and $\hat{h}'(F_x) = \hat{c}'_x$. Let $Range(\hat{c}_x) = \bigsqcup_{i=1}^{sets_x} Range(\hat{c}_x(f_{i,x}))$, and $Range(\hat{s}_{i,x}) = \bigsqcup_{a=1}^{A_x} \hat{s}_{i,x}(l_{i,x}^a)$. Consider a cache block $m_x \in Range(\hat{c}_x)$, $m_x \notin Range(\hat{c}'_x)$, and suppose $m_x \subset m_y$, and $m_y \in Range(\hat{c}'_y)$, $m_y \notin Range(\hat{c}_y)$, with $x < y$. This situation is depicted in Figure 7.1a(with $m_x = a$ and $m_y = b$). Separate joins at levels x and y would result in cache blocks m_x

and m_y to be absent from both cache levels F_x and F_y. However, any access to the cache block m_x would not go beyond the F_y level. The Must analysis can be made more precise by including the 'partial' block m_x^{pr} at the yth level. The whole cache block m_y cannot be included at the yth level, because the linesizes at the two levels may be different, and in general, nothing can be said about the accesses to m_y which are not in m_x.

Note that partial blocks can be useful even when the actual blocks themselves survive the join process. In the above example, if $m_y \in Range(\hat{c}_y)$, then m_y would be present in the join(as depicted in Figure 7.1b). However, join in Must analysis also takes the maximum of the two ages. Hence, if the age of m_y is lower in $\hat{c}_y^{'}$, then the partial block m_x^{pr} would have a lower age than the actual block(and its superset) m_y in the join. This would mean the partial block would survive longer than the actual block for the subsequent accesses, making the Must analysis more precise.

The partial block m_x^{pr} will be present in the cache at level y, but it is linked to the block m_x at level x, and to the block m_y at level y. If either of these blocks get evicted from the cache during subsequent accesses, then the partial block m_x^{pr} should also get evicted. To achieve this, we place the partial block at the minimum eviction distance of the two actual blocks, with an age of $A_y - Min(A_x - h_x, A_y - h_y)$. A_x and A_y are the associativities, while h_x and h_y are the ages of the cache block m_x and m_y respectively.

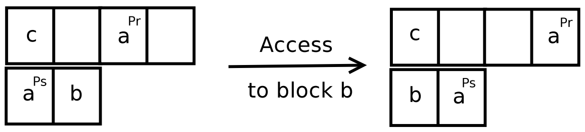

Figure 7.2: Indirect Age update through Pseudo block

If a subsequent access reaches level x, but does not reach level y, even then, it may result in an increase in the age of the actual block m_x at level x, which should be propagated to the partial block at level y(to maintain the condition that if the actual block gets evicted, the partial block should also get evicted). However, the actual block m_x may not have survived the earlier join process in the abstract cache. To solve this issue, whenever a partial block is added, we also add a corresponding 'pseudo' block at the lower level. In the above example, the pseudo block m_x^{ps} would be added at level x, with the same age as the actual block m_x. The pseudo blocks cannot be used to satisfy any accesses, but they experience age increase just like any normal cache block. These age increases are propagated to the corresponding partial blocks to which the pseudo blocks are linked. For example, in the abstract hierarchy state shown in Figure 7.2, the access to cache block b will be satisfied at L1, and hence will not reach L2(thus not updating the age of block c in L2). However, the increase in the age of pseudo block a^{Ps} will be forwarded to the partial block a^{Pr}, resulting in an indirect age update.

If the block m_x at level x is dirty, then it is not necessary to keep track of its eviction from level x in the process of maintaining the age of the partial block. This is because on eviction, it would be written to the next level, and hence keeping the partial block at level y would still be safe. Hence, in such a scenario, the pseudo block m_x^{ps} would not be included in the join at level x, and the partial block m_x^{pr} will be included in the join at level y with the same age as the actual block m_y. Since the pseudo block is not present at the lower level, only accesses which reach level y will contribute to the aging of the partial block. This scenario is shown in Figure 7.1c, where the dirty block a results in the creation of the lone partial block a^{Pr} with the same age as the actual block b in L2.

In general, the following strategy for joins is proposed: For join at the level y, we first do a normal set-by-set intersection of $\hat{c}_y$ and $\hat{c}_y^{'}$. After that, for each such cache block m_y in $\hat{c}_y$, if there exists $m_x \in Range(\hat{c}_x^{'})$, $1 \leq x < y$ with $m_x \subset m_y$, we include the partial block m_x^{pr} in the join at level y, with the minimum eviction distance of both levels. We also include the pseudo block m_x^{ps} at level x with the same age as the actual block m_x. If the block m_x is dirty, then the pseudo block will not be included. We repeat the procedure for all cache blocks in $\hat{c}_y^{'}$. For partial(and pseudo) blocks which are already present, we do a level-by-level, set-by-set intersection, just like normal cache blocks. Finally, if the partial(or pseudo) block is present in $\hat{c}_y$, while the corresponding actual block is present in $\hat{c}_y^{'}$, then the partial(or pseudo) block will be present in the join as well, with the maximum of the two ages.

To recap:

- Partial blocks are treated just like normal cache blocks, except that they may not have the same size as a normal cache block on the same level. They may or may not have a corresponding pseudo block at a lower level. If they have a pseudo block, then any increase in age to the pseudo block will be propagated to the partial block as well. A partial cache block in level x will contain less than(or equal to) r_x basic cache blocks. To determine whether an access can be satisfied by the partial cache block, only the basic cache blocks present in it are considered.

- Pseudo blocks are different from normal cache blocks in the fact that they cannot be used to satisfy any accesses. However, just like normal cache blocks, they will experience age increases and eventually get evicted. There exists a partial block for every pseudo block, but there may exist partial blocks who do not have any corresponding pseudo blocks.

Allowing such joins across levels will change the meaning of Must analysis in the multi-level cache context. While for single level caches, the must analysis gives the upper bound on the age of cache blocks, for multi-level caches, the new must analysis gives an upper bound for normal cache blocks, and for partial cache blocks, it gives the upper cache level bound, implying that in the concrete cache hierarchy states corresponding to the abstract state, these blocks will either be at the same level or at a lower level(determined by the size of the partial block), with an eviction distance less than or equal to the eviction distance of the partial block.

7.3 Abstract Lattice for Multi level cache analysis

We consider the most difficult case of non-inclusive, writeback, write-allocate cache hierarchy. As noted earlier, for such a cache hierarchy, we need all the three analyses for safety and precision. Our abstract lattice is hence the cross-product lattice $\hat{H} \times \hat{H} \times \hat{H}$. We carry out our improved Must analysis, May Analysis, and Improved Persistence analysis on three different abstract cache hierarchy states. The algorithm (shown on the next page) describes how the Must, May and Persistence caches are updated across all levels due to a single access X.

```
Input: $\hat{h}_{in}^{Must}, \hat{h}_{in}^{May}, \hat{h}_{in}^{Per}, X$
Output: $\hat{h}_{out}^{Must}, \hat{h}_{out}^{May}, \hat{h}_{out}^{Per}$
$Access_1^{May} = X;$
if $|X| == 1$ then
    $Access_1^{Must} = X;$
else
    $Access_1^{Must} = \{\};$
end
x = 1;
while $x \leq n$ and $Access_x^{May} \neq \phi$ do
    $\hat{h}_{out}^{Per}(F_x) = \hat{U}_{Per}^{\hat{C}_x}(\hat{h}_{in}^{Per}(F_x), Access_x^{May});$
    $\hat{h}_{out}^{Must}(F_x) = \hat{U}_{Must}^{\hat{C}_x}(\hat{h}_{in}^{Must}(F_x), Access_x^{May});$
    $\hat{h}_{out}^{May}(F_x) = \hat{U}_{May}^{\hat{C}_x}(\hat{h}_{in}^{May}(F_x), Access_x^{Must});$
    /* Order of update is important, as Persistence Analysis uses Must and May cache before update
       */
    Bring all cache blocks accessed in $Access_x^{May}$ to first position in their respective set states in $\hat{h}_{out}^{May}(F_x);$
    $Access_{x+1}^{Must} = Access_x^{Must} - \{m \in Access_x^{Must} | Access\ m\ is\ satisfied\ by\ \hat{h}_{in}^{May}(F_x)\};$
    $Access_{x+1}^{May} = Access_x^{May}$ - $\{m \in Access_x^{May} | Access\ m\ is\ satisfied\ by\ \hat{h}_{in}^{Must}(F_x)\};$
    if $Access_{x+1}^{Must} == \phi$ then
        /* Add empty access if no access is guaranteed */
        $Access_{x+1}^{May} = Access_{x+1}^{May} \cup \{\perp\};$
    end
    if $x \neq n$ then
        $evicted_x^{May}$ = Set of newly evicted dirty cache blocks in $\hat{h}_{out}^{Per}(F_x);$
        $\hat{h}_{in}^{Per}(F_{x+1}) = \hat{U}_{Per}^{\hat{C}_{x+1}}(\hat{h}_{in}^{Per}(F_{x+1}), evicted_x^{May});$
        $\hat{h}_{in}^{Must}(F_{x+1}) = \hat{U}_{Must}^{\hat{C}_{x+1}}(\hat{h}_{in}^{Must}(F_{x+1}), evicted_x^{May});$
        Bring all cache blocks in $evicted_x^{May}$ to first position in their respective set states in $\hat{h}_{in}^{May}(F_{x+1});$
    end
    $x = x + 1;$
end
if $x < n$ then
    while $x \leq n$ do
        $\hat{h}_{out}^{Must}(F_x) = \hat{h}_{in}^{Must}(F_x);$
        $\hat{h}_{out}^{May}(F_x) = \hat{h}_{in}^{May}(F_x);$
        $\hat{h}_{out}^{Per}(F_x) = \hat{h}_{in}^{Per}(F_x);$
        $x = x + 1$
    end
end
```

Algorithm 1: Multilevel Cache Analysis Update

For simplicity, we have not shown the details of the update to pseudo and partial blocks in the algorithm. In the algorithm, $\hat{U}_Y^{\hat{C}_x}$ is the transfer function for the cache level x for $Y \in \{Must, May, Persistence\}$ Analysis. While Must and Persistence caches can be directly updated using $Access^{May}$, the accessed cache blocks in $Access^{May}$ are simply brought to the first position in the May cache, while the update on the ages of the existing cache blocks in May cache takes place using $Access^{Must}$. Another thing to note is that if $Access_x^{Must}$ is empty, then this implies that access is not guaranteed at level x. The reason behind adding the empty access to $Access_x^{May}$ is to make sure this set will not be singleton. This ensures that the Must Analysis transfer function will not bring any new cache blocks to the Must cache at level x.

Chapter 8

Experimental Evaluation

The evaluation was carried out in two steps. In the first step, we only implemented the improved Must analysis for a single-level cache, to determine its improvement on precision as compared to the original Must analysis. In the second step, we implemented all the three analyses for a multi-level cache hierarchy, first without the partial block optimization, and then with it.

As part of the first step, we implemented the improved Must Analysis on top of the prototype for WCET estimation used by [10]. This prototype is built for estimating WCET of programs for the ARM7TDMI processor, which is a 32-bit RISC processor used in a number of real-time devices such as audio equipments, printers, etc. We did not make any changes in the Address Analysis and the ILP parts of the prototype. The prototype uses the original Must Analysis to estimate the cache contents and classify each memory access as always hit/indeterminate. We replaced this part with the improved Must analysis. The configuration of the cache memory used is : 16k, 4-way L1 cache with 32 bytes block size. We also assume the following latencies : Read/Write hit latency = 1 cycle, Read/Write Miss latency = 100 cycles. Apart from the memory accessing instructions, every other instruction has a latency of 1 cycle.

```
1   sum = 0;
2   for (i = 0; i < 40; i++)
3   {
4     rowsum = 0;
5     for (j = 0; j < 40; j++)
6       rowsum = rowsum + array[i][j];
7     sum = sum + rowsum;
8   }
```

Consider the above program used for summing all the elements of a matrix. The access to *array* on line 6 is a multi-reference access, and hence, the original Must analysis will empty the Must cache, removing the cache block containing the variable *sum*. However, because arrays are sequentially stored in the memory, they would actually span multiple consecutive cache blocks. Hence, a maximum of 1 or 2 cache blocks are actually accessed per cache set by the array access on line 6. Since the cache associativity is 4, there are not enough younger cache blocks mapping to the cache set containing the variable *sum* for it to be evicted. This will be captured by the improved Must analysis, and all the accesses to *sum* on line 7 will be classified as hit. The WCET estimated using the original Must analysis for the above program is 174557 cycles, while the WCET estimated using improved Must analysis is 170498 cycles. The difference(of 2.3%) corresponds to the multiple accesses of variable *sum* on line 7, across loop iterations.

The above program could be used in the scenario where the individual sum of each row of the matrix is also important. In general, programs with nested loops where the inner loops access either arrays or pointers, and the outer loops access variables which are not accessed by the inner loops will benefit from improved Must analysis. Generally array sizes in real world programs will be large enough to span all the cache sets, but not so large so as to fill the entire cache. Caches with high associativity and high block sizes will further ensure that the above conditions are met. We also estimated WCET for programs in the WCET benchmarks used in [10]. However, these benchmarks programs either have only single loops, or nested loops where all variables are accessed in the inner loops. Hence, the WCET estimates obtained using improved Must analysis were exactly the same as those obtained using the original Must analysis. Programs which analyze a collection of data structures such as lists, arrays, etc. and aggregate information from all the members of the collections will have nested loops. The inner loops would analyze individual members, while the outer loop would aggregate information. For such programs (which roughly have the same structure as the program shown above), our analysis would be able to provide better estimates.

As part of the second step, we implemented our algorithm for Multi-level cache analysis, along with the improved join for Must analysis, on top of the prototype built in step 1. As in the first step no changes were made in the Address Analysis and the ILP parts of the prototype. The prototype used single-level Must analysis to predict cache contents and classify memory accesses as Always hit or Indeterminate. We have upgraded this part by performing Must, May and Persistence analysis, for a 2-level non-inclusive, writeback, write-allocate cache.

The cache configuration is the following : 16k, 4-way L1 with block size 32 bytes, 64k 8-way L2 with block size 64 bytes. We ran our prototype on the WCET benchmarks obtained from [1]. The detailed results for different benchmarks using different variants of our Multi-level cache analysis are shown in Table 8.1. Following are some of the important observations of our experiments:

- We have showed that performing Persistence analysis, in addition to Must and May analysis, is necessary for obtaining a safe WCET estimate for Multi-level caches. However, it also results in marked improvement in the precision of the WCET estimate as well. We first implemented our Multi-level cache analysis using only Must analysis for classification of accesses as Always Hit or Indeterminate, and then used Persistence analysis to classify accesses inside loops as First Miss(FM). 12 of the 15 benchmarks that we analyzed showed improvement in their WCET estimates, when Persistence Analysis is used for classification, with the maximum improvement as high as 49.72% for the *bs* benchmark. The average improvement over all benchmarks is 21.06%.

- Partial blocks are included in higher level caches to improve the precision of join in Multi-level Must analysis. We next added the partial block optimization to our prototype. Out of 15, 3 benchmarks (*cnt*, *duff*, *matmult*) showed an improvement of 25.16 %, 14.63 %, and 0.7% respectively, while the rest of the benchmarks did not show any improvement in their WCET estimates. Considering the substantial improvement shown by two benchmarks above and none by the rest, it seems that programs with certain access patterns can benefit hugely from the Improved Join in Multi-level must analysis. However, it is interesting to note that in 13 of the 15 benchmarks, partial blocks were generated during the analysis, but either the accesses to L2 were not satisfied by these partial blocks, or the actual blocks themselves were present (alongwith the partial blocks). Also, in a number of benchmarks, the partial blocks were able to satisfy a subset of the references in a multi-reference access, which does not result in any change in the final estimated WCET(as the access will still be classified as miss).

- Improving the Address Analysis is one way of addressing the issue of large access sets for multi-reference accesses. However, even with the same address analysis, by partial physical and virtual unrolling of the loops in a program, the size of access sets can be decreased, along with the number of multi-reference accesses[10]. We employed the technique of loop unrolling with an unroll fraction of 0.5 and then ran Multi-level cache analysis without partial block optimization as well as with partial block optimization on the unrolled program. We found that 3 more benchmarks benefited from the partial block optimization, with the maximum improvement in precision being 25.96%(for *edn_latsynth*). Thus, along with the three previous benchmarks, a total of 6 out of 15(40%) of benchmarks now showed higher precision in their WCET estimates due to the use of partial blocks, with the average precision improvement being 14.52% over these 6 benchmarks.

Table 8.1: Comparison of WCET cycles estimated using different cache analyses

Benchmark	Cache Analysis w/o FM Classification	Cache Analysis with FM Classification		Cache Analysis with FM Classification, Partial blocks	
		Normal	50% unroll	Normal	50% unroll
cnt	42537	27147	27147	20316	20316
edn_fir	86960	84588	96678	84588	91458
edn_fir_no_red_ld	528265	470755	294346	470755	294346
edn_iir	48578	31028	19877	31028	19877
jfdcint	21260	18560	4577	18560	4307
edn_latsynth	47957	47957	10397	47957	7697
edn_mac	48533	27113	17213	27113	17213
bsort100	331466	318891	239175	318891	239175
duff	1020	615	675	525	675
fibcall	139	139	139	139	139
lcdnum	1058	789	789	789	789
qurt	5227	5227	5227	5227	5227
sqrt	985	688	592	688	592
matmult	16960	12829	12829	12739	12739
bs	1629	819	653	819	653

Chapter 9

Proof of Safety of Improved Must Analysis

The improved Must Analysis differs from the original Must analysis in two places : the abstract lattice and the transfer function. For the following discussion, assume that we are dealing with a single-level cache. While the abstract lattice used by the original Must analysis was the set of all abstract cache states(represented by $\hat{C}$), the lattice for the improved must analysis is the cross product lattice $\hat{C} \times \hat{C}$. The improved Must Analysis produces a pair of abstract cache states at each program point, one corresponding to the Must cache and other corresponding to the May cache. The May cache will be used in the transfer function for the Must Analysis as shown in Section 4. We assume the original May analysis transfer function.

The standard method of proving safety of an abstract interpretation based analysis is the following : First specify the concretization function(γ) which converts an element of the abstract lattice to an element of the concrete lattice, specify the abstraction function (α), which does the opposite thing, and then show that a Galois Connection exists between the two functions. The second step is to show the correctness of the abstract transfer function by proving it as an abstraction of the concrete transfer function.

9.1 Proof of Galois connection

Proving the first part in our case is straightforward, as both the concretization and abstraction functions are simple extensions of the corresponding functions for the original Must analysis. Given a Must cache and a May cache, γ converts it to a set of concrete cache states (which is an element of the concrete lattice 2^C). All the cache blocks in the Must cache must be present in all the concrete cache states given by γ, while any cache block in any concrete cache state must come from the May cache. The age of a cache block in the concrete cache will be upper bounded by its age in the Must cache and lower bounded by its age in the May cache. Formally, the concretization function can be written as:

$$\gamma^{\hat{C}}(\hat{c}^{Must}, \hat{c}^{May}) = \{c \in C | \forall i, 1 \leq i \leq sets,\ c(f_i) \in \gamma^{\hat{S}_i}(\hat{c}^{Must}(f_i), \hat{c}^{May}(f_i))\}$$

$$\gamma^{\hat{S}_i}(\hat{s}_i^{Must}, \hat{s}_i^{May}) = \{s \in S_i | (\forall a, 1 \leq a \leq A,\ \forall m \in \hat{s}_i^{Must}(l_i^a), \exists b, 1 \leq b \leq a, s(l_i^b) = m)$$
$$\wedge\ (\forall d, 1 \leq d \leq A, s(l_i^d) = m, m \in M \cup \{\perp\} \wedge\ \exists e, 1 \leq e \leq d, m \in \hat{s}_i^{May}(l_i^e))\}$$

α takes as input a set of concrete cache states and outputs the corresponding Must and May cache, and is defined as follows : An abstract set state in Must cache will be just the intersection of the corresponding concrete set states in the concrete caches, with the age in the abstract set being the maximum of the ages in the concrete set. Similarly, an abstract set state in the May cache will be union of the corresponding concrete set states, with the age in the abstract set being the minimum of the ages in the concrete set. To prove that γ and α form a Galois connection, we need to prove the following two statements:

$$\gamma(\alpha(S)) \supseteq S,\ \forall S \in 2^C \tag{9.1}$$

$$\alpha(\gamma((\hat{c}^{Must}, \hat{c}^{May}))) = (\hat{c}^{Must}, \hat{c}^{May}),\ \forall (\hat{c}^{Must}, \hat{c}^{May}) \in \hat{C} \times \hat{C} \tag{9.2}$$

For (9.1), consider any $c \in S$, and now consider any cache block m present in c. This cache block will be present in the May cache generated by $\alpha(S)$, and its age in the May cache will less than or equal to its age in c. Hence, m respects the lower bound imposed by the May cache. If it is not present in all cache states of S, it will not be present in the Must cache and hence there will be no upper bound limit (except the associativity). If it is present in the Must cache, then its age will be greater than or equal to its age in c, in which case m respects the upper bound imposed by the Must cache. Hence, γ will produce a concrete cache state where m has the same age as in c, and this is true for all cache blocks m in c. Hence $c \in \gamma(\alpha)(S)$. The proof of (9.2) can be similarly sketched.

9.2 Proof of Abstraction

Before proving the correctness of the transfer function, we will prove the following Lemma:

Lemma 9.2.1 *For a cache block with age h in the Must Cache, Number of cache blocks with age less than or equal to h in the May cache $\geq h$. This means that there are atleast h number of younger cache blocks in the May cache, which in turn implies that $MaxYoung(X, h) \geq h$, for any access X.*

Proof Consider the cache block m at position h in the Must cache at some program point P. Now, consider the last program point Q where the cache block m was the youngest in the Must cache(i.e. was at position 1). New cache blocks are brought(or ages are decreased) in the must cache only for single-reference accesses. Hence, the instruction just before the program point Q must have accessed the cache block m, and this access must have been single-reference. Single-reference accesses to the May cache also bring in the accessed block to position 1, and increase the age of all the cache blocks already present in the May cache by 1. Hence, at Q, cache block m will be the only cache block at position 1 in the May cache. Hence, the statement of the lemma is true at this point. Between program points Q and P, the cache block m ended up in position h in the Must cache, and there are no single-reference accesses to the block m between Q and P. Now, when the transfer function for Must Analysis increases the age of m

by a, at least a cache blocks will be added by the transfer function of May analysis to position 1 in the May cache. Hence cache updates by the transfer function preserve the statement of the lemma. The join for May analysis is cache set union while taking the minimum of the ages, while the join for Must analysis is cache set intersection, while taking the maximum of the ages. Hence, if cache block m is in position h_1 in Must cache $\hat{c}_1^{Must}$ and position h_2 in Must cache $\hat{c}_2^{Must}$, then there are atleast h_1 younger cache blocks in the May cache $\hat{c}_1^{May}$ and at least h_2 younger cache blocks in the May cache $\hat{c}_2^{May}$. After the join in the May cache all these $h_1 + h_2$ will be in positions less than $Max(h_1, h_2)$, which is the new position of cache block m in the Must cache after the join. Since $h_1 + h_2 \geq Max(h_1, h_2)$, the validity of the lemma is preserved. ∎

To prove the correctness of the abstract transfer function, we need to prove the following :

$$\gamma(\hat{U}((\hat{c}^{Must}, \hat{c}^{May}), X)) \supseteq U(\gamma(\hat{c}^{Must}, \hat{c}^{May}), X)$$

The set of concrete cache states, produced by applying γ on the updated abstract cache state obtained by applying the abstract transfer function($\hat{U}$) due to an access X, is the L.H.S of the above equation. In words, the equation means the following : if we instead apply γ on the un-updated abstract cache state, and then apply the concrete transfer function(U) individually on each of the concrete cache states, then the set of updated concrete cache states so produced must be a subset of the L.H.S. We have already stated that the concrete transfer function is nothing but a simple LRU update. Since $\gamma(\hat{c}^{Must}, \hat{c}^{May})$ and X are both sets,

$$U(\gamma(\hat{c}^{Must}, \hat{c}^{May}), X) = \bigsqcup_{c \in \gamma(\hat{c}^{Must}, \hat{c}^{May})} \bigsqcup_{m \in X} \{U(c, m)\}$$

where $U(c, m)$ updates the concrete cache state c due to the access to the block m. If X is singleton, then the abstract transfer function for Improved Must Analysis is the same as the original Must analysis, for which the abstraction proof is already known. Hence, we only deal with multi-reference accesses. Consider a concrete cache state c, $c \in \gamma(\hat{c}^{Must}, \hat{c}^{May})$, and an access m, $m \in X$. We will show that the updated c after the access will be in the concretization set of the updated Must and May cache. To show this, we have to prove that all cache blocks in the updated Must cache will be present in the updated c, and all cache blocks in the updated c will be present in the updated May cache. Also, cache blocks in the updated c should respect the upper bounds and lower bounds set by the updated Must and May cache, respectively. Since we use the original May analysis, we know that the abstract transfer function for May analysis is an abstraction of the concrete transfer function. Hence, there is no need to prove the results involving the May cache update.

Now, for a multi-reference access X, the Must analysis transfer function does not bring any new cache blocks into the must cache. All the cache blocks in the must cache before the update are already present in c, hence, we have to show that if such a cache block gets evicted from c by the concrete transfer function, then it will also be evicted from the must cache by the abstract transfer function. This proves that all the cache blocks in the updated Must cache will also be present in the updated c. If a cache block m^e gets evicted from c by the concrete transfer function, it must have the maximum age, i.e. it must be in l_i^A. If this cache block is also present in the Must cache, then it must also have the maximum age in the Must cache, since age in the concrete cache is upper bounded by age in Must cache. Also, eviction of m^e from c implies that the accessed block m is not present in c, which would mean that m is not be present in the must cache as well. Since $m \in X$, by definition, $shiftctr(X, A) \geq 1$, hence $A + shiftctr(X, A) > A$.

Now, by the lemma proved earlier, we know that there are at least A cache blocks in the May cache. If the accessed block m is not in May cache, then $MaxYoung(X, A) > A$, counting the accessed cache block m along with the minimum A number cache blocks in the May cache. Hence $NewPos(X, h) > A$ and so m^e will be evicted from the Must cache as well. In fact, if any of the accessed blocks in X, and not necessarily m are not in the May cache, even then $MaxYoung(X, A) > A$. Let us consider the case where all the accessed blocks in X are in the May cache. Now the cache block m^e is in position A in the concrete cache c before the update. The $A - 1$ cache blocks younger than m^e in c must come from the May cache. Hence, these A cache blocks(including m^e) are all present in the May cache. Also, the accessed cache block m is in the May cache, but it is not present in the concrete cache c. Hence, there are atleast $A + 1$ cache blocks in the May cache. Thus, $MaxYoung(X, A) > A$, and hence, in all cases m^e will be evicted from the Must cache as well by our new transfer function.

The last thing to prove is that the cache blocks in the updated concrete cache c respect the upper bounds set by the updated Must cache after the access. We know that before the update, the cache blocks in c do satisfy the upper bounds set by the Must cache. After the update by the concrete transfer function, the ages of all or some cache blocks in exactly one cache set of c will be increased by 1. This cache set will be the set to which the access m is mapped.

First let us take the case where the accessed cache block m is not present in c. Then the ages of all cache blocks will be increased by 1. In this case, the accessed block m will not be present in the Must cache as well, hence the shiftctr function for all positions $h, 1 \leq h \leq A$ will be atleast 1. Now, for position h, we know that $MaxYoung(X, h) \geq h$. Let m_h be the cache block in position h in the concrete cache c. If the accessed cache block m is not present in the May cache as well, then adding the accessed block to the younger set would mean that $MaxYoung(X, h) > h$. Even If the accessed cache block m is present in the May cache, consider the cache blocks from position 1 to h in the concrete cache c. These h cache blocks must be in positions less than or equal to h in the May cache and hence will contribute to the count of $MaxYoung(X, h)$. And the accessed block m is not any of these h cache blocks, so it will also contribute to MaxYoung. Hence $MaxYoung(X, h) \geq h + 1$. Hence, $NewPos(X, h) \geq h + 1$. Hence the upper bounds set by the Must cache are still maintained after the update.

Now, let us take the case where the accessed cache block m is present in c at position h. In this case, the cache blocks at positions less than h will see an increase of age by 1 due to the concrete transfer function. Now, the accessed block m will either not be present in the Must cache at all, or if present it will be present at positions greater than or equal to h. In either case, $shiftctr(X, a) \geq 1, \forall a, 1 \leq a \leq h - 1$. Also, by a similar argument as used earlier, $MaxYoung(X, a) \geq a + 1$, $\forall a, 1 \leq a \leq h - 1$. Hence, the transfer function for the improved must analysis will also increase the ages of cache blocks at positions less than h by atleast 1, thus maintaining the upper bounds.

Chapter 10

Proof of Safety of Multi-level Cache Analysis

In a general abstract interpretation framework, proving correctness of the abstract lattice and abstract transfer functions requires two things: First, to specify the concretization function γ and the abstraction function α, and to prove that they form a Galois Connection. Second, to show that the abstract transfer function is an abstraction of the concrete transfer function.

10.1 Galois Connection

In the Multi-level Cache Analysis framework, the concrete lattice is the power set of the set of all concrete cache hierarchy states(H). Our abstract lattice is the cross-product lattice $\hat{H} \times \hat{H} \times \hat{H}$. The concrete cache analysis would give a set of concrete cache hierarchy states at each program point, where each cache hierarchy state would describe the possible cache state during some execution of the program. The concretization function for our Abstract lattice(without the partial block optimization) is given as follows:

$$\gamma^{\hat{H}}(\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per}) = \{h \in H | \forall x, 1 \leq x \leq n,$$

$$h(F_x) \in \gamma^{\hat{C}_x}(\hat{h}^{Must}(F_x), \hat{h}^{May}(F_x), \hat{h}^{Per}(F_x))\}$$

$$\gamma^{\hat{C}_x}(\hat{c}_x^{Must}, \hat{c}_x^{May}, \hat{c}_x^{Per}) = \{c \in C_x | \forall i, 1 \leq i \leq sets_x,$$

$$c(f_{i,x}) \in \gamma^{\hat{S}_x}(\hat{c}_x^{Must}(f_{i,x}), \hat{c}_x^{May}(f_{i,x}), \hat{c}_x^{Per}(f_{i,x}))\}$$

$$\gamma^{\hat{S}_x}(\hat{s}_{i,x}^{Must}, \hat{s}_{i,x}^{May}, \hat{s}_{i,x}^{Per}) = \{s \in S_{i,x} | (\forall a, 1 \leq a \leq A_x,$$

$$\forall m \in \hat{s}_{i,x}^{Must}(l_{i,x}^a), \exists b, 1 \leq b \leq a, s(l_{i,x}^b) = m)$$

$$\wedge (\forall d, 1 \leq d \leq A_x, s(l_{i,x}^d) = m \wedge$$

$$\exists e, 1 \leq e \leq d, m \in \hat{s}_{i,x}^{May}(l_{i,x}^e) \wedge \exists f, d \leq f \leq A_x + 1,\ m \in \hat{s}_{i,x}^{Per}(l_{i,x}^f))\}$$

The concretization function treats each set of each level of the cache hierarchy separately. For a set, the concrete set states corresponding to the given abstract set states in Must cache, May cache and Persistence cache must be in accordance to the following rules : All the cache blocks present in the abstract set of the must cache should be present in the concrete set state, with their age upper bounded by the age of the cache block in the must cache. Any cache block present in the concrete set state must also be present in both the abstract set states of the May cache and Persistence cache, and the age of the cache block in the concrete set must be lower bounded by its age in the May cache, and upper bounded by its age in the Persistence cache. The equations given above mathematically state the above rules.

The abstraction function takes a set of concrete cache hierarchies(which is a member of the concrete lattice) and gives three abstract cache hierarchies-$\hat{h}^{Must}$, $\hat{h}^{May}$, $\hat{h}^{Per}$. The following natural definition of the abstraction function follows from the Concretization function and the need to maintain the Galois Connection : Let S be the set of Concrete Cache hierarchies. For defining $\hat{h}^{Must}$, for each cache level, and for each set, we take the intersection of that concrete set across all cache hierarchies in S, with the age of a cache block being the maximum of its age in all concrete sets. For defining $\hat{h}^{May}$, for each cache level and for each set, we take the union across all cache hierarchies, and the age of a cache block in $\hat{h}^{May}$ will be the minimum of its age across all cache hierarchies in S. For defining $\hat{h}^{Per}$, we again take the union across all cache hierarchies, but the age of a cache block will be the maximum across all cache hierarchies. This, however, will mean that the eviction line will remain empty in $\hat{h}^{Per}$ for all sets at all cache levels. This problem arises only because there are no eviction lines in the concrete caches, and can be easily solved by keeping an eviction way in all sets across all levels in the concrete caches. This will not affect the cache functions, because the eviction way will not be considered for cache lookup. Hence, we ignore the eviction way in concrete cache.

For the above definitions of γ and α, it is clear that $\gamma^{\hat{H}}(\alpha^{\hat{H}}(S)) \supseteq S$, for any set S of concrete cache hierarchies. Consider $h \in S$, we argue that for every cache block m in h, there exists a cache hierarchy in $\gamma^{\hat{H}}(\alpha^{\hat{H}}(S))$, which contains m at the same level, with the same age . Let m be at level x in h with age a. Then, m will be present in the May cache generated by $\alpha^{\hat{H}}(S)$, at level x with an age less than or equal to a. Hence, m is present in the May cache and its age respects the lower bound set by the May cache. Similarly, m will be present in the Persistence cache with an age greater than or equal to a. These facts ensure that there will be a concrete cache hierarchy generated by $\gamma^{\hat{H}}(\alpha^{\hat{H}}(S))$ with m at level x with age a. Note that while proving this, we have not assumed anything about the other cache blocks present in the concrete cache state. This, combined with the fact that the above result is true for all cache blocks in h prove that the cache hierachy state h will be present in $\gamma(\alpha(S))$.

The two sets may not be equal, because other than the elements of S, other cache hierarchies may also be included by γ, when applied on $\alpha(S)$ since it includes cache blocks at all ages between the minimum and maximum age. Also, after adding the eviction way in concrete caches, it is also clear that $\alpha(\gamma(\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per})) = (\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per})$.

10.2 Abstraction Proof for Single Ref Access

The next step is to consider the abstract transfer function and prove that it is an 'abstraction' of the concrete transfer function. The concrete transfer function is nothing but the actual LRU update of the cache hierarchy. In a write-back, write-allocate, partially inclusive cache hierarchy, the concrete transfer function can be represented as follows:

$$\begin{aligned} U^H(h,m) &= h', where \\ h'(F_1) &= f^{C_1}(h(F_1),m) \\ h'(F_2) &= f^{C_2}(f^{C_2}(h(F_2),m_1^e),m) \\ &\vdots \\ h'(F_r) &= f^{C_r}(f^{C_r}(h(F_r),m_{r-1}^e),m) \\ h'(F_{r+1}) &= h(F_{r+1}) \\ &\vdots \\ h'(F_n) &= h(F_n) \end{aligned}$$

where r is the smallest cache level in which the memory block containing the reference m is present. In the above equations, h indicates a concrete cache hierarchy state, while f^{C_i} is the concrete transfer function of the ith level, which simply performs an LRU update on the cache contents of the ith level. m_i^e is the dirty cache block evicted from the ith level, which will induce an LRU update in the (i+1)th level in a writeback, write-allocate cache hierarchy. (Note that if there is no dirty cache block evicted from the ith level, $m_i^e = \perp$).

We have already stated our abstract transfer function for cache update. We will prove that

$$\gamma(\hat{U}^H(\hat{h}^{Must},\hat{h}^{May},\hat{h}^{Per},X)) \supseteq U^H(\gamma(\hat{h}^{Must},\hat{h}^{May},\hat{h}^{Per}),X)$$

First, let us assume that X is a single reference access(i.e. $X = \{m\}$), and let $S = \gamma(\hat{h}^{Must},\hat{h}^{May},\hat{h}^{Per})$. Let r be the lowest cache level in the Must cache hierarchy $\hat{h}^{Must}$ which can satisfy this reference($1 \leq r \leq n$). Hence, according to our abstract transfer function, $Access_{r+1}^{May}$ is guaranteed to be empty, because $Access_r^{May}$ can atmost be $\{m\}$, and it will be removed since Must cache at level r satisfies this request. Hence, $\hat{U}^H$ will not update the cache levels beyond level r. Also, the memory block satisfying the reference will be also be present in the Persistence Cache and the May cache at level r, hence, $evicted_r^{May}$ will also be empty. Thus, the abstract update function will behave as identity for the cache levels in all the three hierarchies beyond level r. Now, for all concrete cache hierarchies in S, the memory block satisfying m must be present at cache level r, since it is present in the Must cache at level r. Hence, the concrete update function will not change the contents of cache level beyond r for all concrete cache hierarchies in S. This, it is clear that both U^H and $\hat{U}^H$ behave as identity functions for cache levels beyond r.

Let r' be the lowest cache level such that the access can be satisfied by the May cache at level r'. None of the May caches at level less than r' contain the memory block satisfying the access. Consider cache level $i \leq r'$. Since the May cache at level $i-1$ does not satisfy the access, $Access_i^{Must} = \{m\}$. Hence, the cache block containing the referenced memory location will be brought into the first position in $\hat{h}^{Must}(F_i), \hat{h}^{May}(F_i)$ and $\hat{h}^{Per}(F_i)$. Note that only the set to which the memory block is mapped will be affected, while the rest of set will remain the same. Now, in any concrete cache hierarchy in S, the access is guaranteed to reach level i, since the cache block satisfying the access will not be present at levels less than i. The concrete transfer function will also bring the cache block at the first position in all concrete cache hierarchies in S, for all levels $i \leq r'$. Finally, all the cache blocks that were already present in the Must, May and Persistence caches at levels less than r' (in the set to which the memory block is mapped) will see an increase in their ages by 1. The same update is also carried out by the concrete transfer function for all $h \in S$.

Also, consider the cache block m_i^e evicted from the cache level $i < r'$ in a concrete cache hierarchy $h \in S$ after the update. m_i^e must have had the maximum age in the set for it to be evicted(i.e. A_i the associativity of level i). Since the age of m_i^e is upper bounded by its age in the persistence cache, this means that its age in the persistence cache at level i must also have been A_i. Hence, after the abstract update, $m_i^e \in evicted_i^{May}$, and hence, it will also be brought in the May and Persistence caches at level $i+1$. Note that $evicted_i^{May}$ may also contain other cache blocks, and hence many more cache blocks may be brought to level $i + 1$. Thus, we have argued that the updates made by the concrete and abstract transfer functions at levels $i \leq r'$ are such that the concrete cache hierarchies produced after the concrete update are also produced by the abstract transfer function.

Now, consider the cache level $r' < i \leq r$. Since the May cache at level r' satisfied the access, but none of the Must caches at levels less than r will satisfy the access, hence $Access_i^{May} = \{\perp, m\}$. Hence, the abstract transfer function will bring the cache block satisfying m at the first position only in the May and the Persistence caches, but not in the Must cache. This is in accordance with the update of the concrete transfer function, because there will be concrete cache hierarchies in S which contain the memory block satisfying m at level r', which means that the access will not go beyond level r'.

The cache block satisfying the reference m will be brought to the first position in May cache at all levels between r' and r by the abstract transfer function. Since the empty access($\perp$) is present in $Access_i^{May}$ for all levels between r' and r, the block satisfying m will not be brought to the first position in the persistence cache if the block was already present. This is because there exist h in S such that the access does not reach level i, in which case the contents of cache at level i will not be changed by the concrete transfer function.

For other cache blocks between these levels, note that the ages of cache blocks in the May cache will not be updated by the abstract transfer function(since $\perp$ is present in the access), and hence the lower bound of ages will remain the same. On the other hand, the concrete transfer function will only increase the age of cache blocks in the concrete cache levels. Hence, the lower bounds decided by the May cache in the updated abstract cache hierarchy will be obeyed by the conrete cache hierarhies generated by the concrete transfer function. Since none of the Must caches between levels r and r' contain the accessed memory block, the abstract transfer function will increase the age of all cache blocks in the persistence cache by 1(according to our improved transfer function for persistence analysis). This is in accordance with the increase of age in cache blocks in the concrete cache hierarchies by the concrete transfer function.

Finally, the argument for the evicted cache blocks remains the same as already proved for cache levels less than r'. Note that the evicted cache blocks will also contribute to an increase in the ages of the cache blocks in the Must and Persistence cache between levels r and r'. This increase may be bigger compared to an increase in age due to eviction by the concrete transfer function, since in the concrete cache hierarchies, atmost one cache block will be evicted from a lower to a higer level, while the abstract transfer function considers all newly evicted cache blocks in the persistence cache as evicted. However, this will only increase the upper bound of the cache blocks and may generate concrete cache hierarchies in $\gamma(\hat{U}^H)$ which will not be present in $U^H(\gamma)$, still satisfying the superset relation.

Thus, we have proved that for all cache levels, the abstraction relation is satisfied.

10.3 Abstraction Proof for Multi Ref Access

Let $X = \{m_1, m_2, \ldots, m_k\}$ be the multiple reference access to the abstract cache hierarchy $(\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per})$. Note that there cannot be a multiple reference access to a concrete cache hierarchy. The single reference access to any member of S $(= \gamma(\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per}))$ can be any of the k access in the set X. This essentially means that we apply the concrete transfer function $U^H(h, m)$ for all $h \in S$ and for all $m \in X$ and the set of all the resulting concrete cache hierarchies is $U^H(\gamma(\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per}), X)$.

Proving the abstraction result is equivalent to proving the following : For all $h \in S$, the cache blocks in all levels of $U^H(h, X)$ are present in the May cache and Persistence cache of the corresponding level of $\hat{U}^H(\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per})$, respecting the upper bound set by the Persistence cache and lower bound set by the May Cache, and all the cache blocks at all levels in Must cache are present in the corresponding levels in h. This implies the $U^H(h, X) \in \gamma(\hat{U}^H(\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per}, X))$.

Consider the access $m \in X$ to the concrete cache hierarchy $h \in S$. Let r be the lowest cache level containing the accessed memory block in h. Hence, the accessed memory block must also be present in the abstract May cache at level r. Also, the memory block will not be present in the abstract Must caches at all level $i < r$, because otherwise h itself would contain the accessed memory block at a lower level, and the request would never reach level r. Hence, according to our abstract transfer function, the access $m \in Access_r^{May}$. Therefore, the accessed memory block will be brought in the first position in all levels of May cache upto level r. This is in accordance with the update by the concrete transfer function to h, which will also bring m to first position in all concrete cache levels of h upto level r.

For a multi-reference Access, $Access_i^{Must} = \phi$, for all levels i. Since the abstract transfer function increases the ages of cache blocks in the May Cache only using $Access_i^{Must}$, it is clear that the lower bounds of cache blocks as given by the May cache after the update, will be the same as the lower bounds before the update. Moreover, no cache blocks will be evicted from the May cache at any level. Since the concrete transfer function will only increase the age of cache blocks in h, the cache blocks in the updated concrete cache hierarchy at all levels will be present in the updated May cache and will also respect the lower bounds.

The ages of cache blocks in the persistence cache are updated based on the contents of the Must cache. Since the memory block accessed by m is not present in the Must cache at all levels $i < r$, this will result in an increase in the age of all cache blocks in the Persistence cache by atleast 1. Moreover, cache blocks are never evicted from the persistence cache at any level. The concrete transfer function will increase the ages of all cache blocks upto level r by 1, which matches the increase in age by the abstract transfer function. Hence, the upper bound set by the updated persistence cache will be obeyed by the cache blocks in the updated concrete caches.

Finally, no new cache blocks are brought in the Must cache at any level by the abstract transfer function. Before the update, all the cache blocks in the Must cache at all levels are present in h at the appropriate level. We have to show that if a cache block in the concrete cache at level $i < r$(which is also present in the Must cache at the same level) is removed by the concrete transfer function, then this cache block will also be removed by the abstract transfer function from the Must cache. If a cache block in the concrete cache at level i is evicted, then it must be at position A_i(i.e., the associativity at level i) in terms of its age. Since the upper bound of a cache block is decided by the Must cache(actually by the Persistence cache, but the upper bounds are the same), hence this cache block must also be in the must cache at position A_i. Now we have shown that that the access m is guaranteed to be present in $Access_i^{May}$, since $i < r$, and the abstract trasnfer function updates the ages of cache blocks in the Must cache according to $Access_i^{May}$. Also, the age of all cache blocks in the Must cache at level i is guaranteed to increase by atleast 1, which is enough to remove the cache block under discussion from the Must cache. This proves the Abstraction result for Multi-reference Accesses as well.

10.4 Proof of Termination

Kildall's Algorithm to compute the fixed-point elements of the abstract lattice in the above Abstract interpretation framework is guaranteed to terminate because the abstract lattice is finite and the abstract transfer function is monotonic. The set of all abstract cache hierarchy states($\hat{H}$) is finite, and the abstract lattice is simply the cross-product of three finite sets. The abstract transfer function uses the transfer functions for Must, May and Persistence caches which have already been proved monotonic. Hence, our abstract tranfer function is also monotonic.

10.5 Proof of Safety(With Partial Blocks)

To accomodate partial blocks in the mathematical framework given in Section 3, we alter the definition of Abstract Set state. Let

$$M_x^{'} = \bigcup_{i=1}^{x} M_i$$

Abstract Set State is a function $\hat{s}_{i,x} : f_{i,x} \rightarrow 2^{M_x^{'} \cup \perp}$. Similar to the previous definition, it maps a line in the abstract set $f_{i,x}$ to a set of cache blocks. The difference is that the size of these cache blocks need not necessarily be $linesize_x$, and in fact could range from $linesize_1$ to $linesize_x$. This allows partial blocks to be present in the abstract cache along with the actual blocks.

The size of a partial block is determined by the number of basic cache blocks present in it(as explained in Chaper 7.2). The subset relation among cache blocks is defined as follows : For cache blocks $m_1, m_2 \in M_n^{'}$, $m_1 \subseteq m_2$ if and only if all the basic cache blocks of m_1 are also the basic cache blocks of m_2.

Given a partial block m_p at level x, the full block corresponding to it is given by the function $parent_x : M_x^{'} \rightarrow M_x$. $parent_x(m_p) = m, m \in M_x$ if and only if $m_p \subseteq m$. Since all the cache blocks in M_x are disjoint, there will be a unique full block corresponding to a given partial block.

The presence of a partial block consisting of r_x basic blocks in the abstract Must cache at level y forces either the full block to be present in the concrete cache at level y or a subset of the block to be present at level x. The function $\gamma^{'}$ expresses the above requirement mathematically:

$$\gamma^{'}(\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per}) = \{h \in H | \forall y, 1 \leq y \leq n, \forall i, 1 \leq i \leq sets_y, \forall a, 1 \leq a \leq A_y,$$
$$if\ m_p \in ((\hat{h}^{Must}(F_y))(f_{i,y}))(l_{i,y}^a) and\ |m_p| = r_x,\ where\ x < y,\ then\ if\ set_x(m_p) = j,$$
$$either\ \exists b, 1 \leq b \leq A_x - (A_y - a)\ such\ that((h(F_x))(f_{j,x}))(l_{j,x}^b) = m_p\ or\ \exists c, 1 \leq c \leq a,$$
$$such\ that\ ((h(F_y))(f_{i,y}))(l_{i,y}^c) = parent_y(m_p)\}$$

The final concretization function for the analysis with Partial blocks must generate concrete cache hierarchies which take into account the limitations set by the partial blocks in the Must cache and the contents of the May and Persistence cache.

$$\gamma_{Pr}^{\hat{H}} = \gamma^{\hat{H}} \cap \gamma^{'}$$

The cache hierarchies given by $\gamma^{'}$ only contain those cache blocks which correspond to partial blocks in the Must cache. Since there is no restriction on the other cache blocks, $\gamma^{'}$ will actually generate a large number of concrete cache hierarchies, with the other cache blocks present at all possible ages (or not present at all). The restriction on these other cache blocks comes from the May and Persistence caches. $\gamma^{\hat{H}}$(defined in the previous section) uses the full blocks in the Must cache, as well as the contents of May and Persistence cache to decide which cache blocks should be present in the concrete cache and their ages. Note that the full blocks corresponding to the partial blocks will be present in the May cache(with a lower age) and in the Persistence cache(with a higher age). The partial blocks themselves will now force the actual blocks to be present in all concrete cache hierarchies(as expressed by $\gamma^{'}$) and a more strict bounds on their ages.

The abstraction function($\alpha^{\hat{H}}_{Pr}$) is an extension of the abstraction function($\alpha^{\hat{H}}$) defined in the previous section. Given a set S of concrete cache hierarchies, the May and Persistence caches are exactly the same as those given by $\alpha^{\hat{H}}$. Moreover, the full cache blocks in the Must cache are also determined in the same way as in $\alpha^{\hat{H}}$. For determining which partial blocks will be present in the Must cache, the following rule is used:

If $\exists x, y, 1 \leq x < y \leq n$ and $\exists m, m^{'} \in M^{'}_{y}, m \subseteq m^{'}$, such that $\forall h \in S$ either $\exists a, 1 \leq a \leq A_y, m^{'} \in ((h(F_y))(f_{i,y}))(l^{a}_{i,y})$, (in which case let $h \in S_1$) or $\exists b, 1 \leq b \leq A_x, m \in ((h(F_x))(f_{j,x}))(l^{b}_{j,x})$, (in which case let $h \in S_2$), then $m^{Pr} \in ((\hat{h}^{Must}(F_y))(f_{i,y}))(l^{c}_{i,y})$, where $c = A_y - Min(Min_{h \in S_1}(A_y - a_h), Min_{g \in S_2}(A_x - b_g))$. a_h and b_g are the ages of the cache blocks $m^{'}$ and m in cache hierachies h and g from S_1 and S_2 respectively.

With both $\gamma^{\hat{H}}_{Pr}$ and $\alpha^{\hat{H}}_{Pr}$ fully defined, let us now prove one of the properties of Galois Connection:

$$\alpha^{\hat{H}}_{Pr}(\gamma^{\hat{H}}_{Pr}((\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per}))) = (\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per})$$

Proof Let $\gamma^{\hat{H}}_{Pr}((\hat{h}^{Must}, \hat{h}^{May}, \hat{h}^{Per})) = S$. We will show that every cache block in $\hat{h}^{Must}$ ($\hat{h}^{May}$, $\hat{h}^{Per}$) will also be present in the Must(May, Persistence) cache given by $\alpha(S)$ with the same age. For cache blocks present in $\hat{h}^{May}$ and $\hat{h}^{Per}$ which do not have corresponding partial blocks in $\hat{h}^{Must}$, the upper and lower bounds on their ages in S will be determined by their ages in $\hat{h}^{May}$ and $\hat{h}^{Per}$ respectively. Hence, these cache blocks will be present in the May and Persistence cache produced by $\alpha(S)$ with the same age as in $\hat{h}^{May}$ and $\hat{h}^{Per}$.

For cache blocks of $\hat{h}^{May}$ which have corresponding partial blocks, $\gamma^{'}$ only imposes a new upper bound on their ages, thus not changing the lower bound of such cache blocks among all concrete cache hierarchies in S. For cache blocks of $\hat{h}^{Per}$ which have corresponding partial blocks, the upper bound imposed by the partial blocks will be greater than or equal to the age of the cache block in $\hat{h}^{Per}$. When partial blocks are created in the Must cache, their ages are greater than equal to the ages of the corresponding actual blocks present in the Must cache, and the ages of a cache block in the Must and Persistence caches are the same (both maintain an upper bound). Hence, we have shown that the May and Persistence caches as determined by $\alpha(S)$ will be the same as $\hat{h}^{May}$ and $\hat{h}^{Per}$.

The non-partial cache blocks in $\hat{h}^{Must}$ will be present in all the cache hierarchies in S, and hence will be present in the Must cache given by $\alpha(S)$, with the same age as in $\hat{h}^{Must}$. We will now show that all the partial blocks in $\hat{h}^{Must}$ will also be present in the Must cache given by $\alpha(S)$, with the same age.
Consider a partial block $m_p \in ((\hat{h}^{Must}(F_y))(f_{i,y}))(l^{a}_{i,y})$ of size r_x.
According to $\gamma^{'}$, $\forall h \in S$ either $parent_y(m_p)$ will be present in at level y in h with a maximum age of a, or m_p will be present at level x in h with a maximum age of $A_x - (A_y - a)$.
Now, according to the abstraction function defined, this will result in the partial block m_p to be present in the Must cache given by $\alpha(S)$ at level y with the eviction distance to be minimum of the eviction distance of the block across all hierarchies in S. The minimum eviction distance at level y across all $h \in S$ would be $A_y - a$ (since the maximum age is a), while the minimum eviction distance at level x would be $A_x - (A_x - (A_y - a))$ (since the maximum age is $A_x - (A_y - a)$). Hence, the age of the partial block is given by

$$\begin{aligned} c &= A_y - Min(A_y - a, A_x - (A_x - (A_y - a))) \\ &= A_y - Min(A_y - a, A_y - a) \end{aligned}$$

$$
\begin{aligned}
&= A_y - (A_y - a) \\
&= a
\end{aligned}
$$

Thus, the partial block will be present in the Must cache given by $\alpha(S)$ with the same age as in $\hat{h}^{Must}$. ▮

The other property of Galois connection, i.e. $\gamma_{Pr}^{\hat{H}}(\alpha_{Pr}^{\hat{H}}(S)) \supseteq S, \forall S \in 2^H$, was proved in the previous section for Multi-level cache analysis without Partial blocks. We consider $h \in S$ and consider cache block m at level x in h, with age a. By the argument of the previous section, $\gamma^{\hat{H}}(\alpha_{Pr}^{\hat{H}}(S))$ contains a concrete cache hierarchy which contains m at the same level with the same age. We will show that $\gamma^{'}(\alpha_{Pr}^{\hat{H}}(S))$ also contains a cache hierarchy with m at the same level and same age. If m does not generate a partial block, then there is no restriction on the age or presence of m, and hence the above result will be true. If m does generate a partial block (either at level x or a higher level y), then the eviction distance of this partial block will be less than or equal to the eviction distance of m in h. For a partial block with eviction distance e, $\gamma^{'}$ generates cache hierachies with the actual block at all eviction distances greater than or equal to e. Hence, $\gamma^{'}$ will generate a concrete cache hierarchy with m at the same level x and age a.

Chapter 11

Conclusion

In our work, we have shown that there is scope for improvement in the precision of the original Must Analysis which is the cornerstone of most of the theories for WCET estimation. The impact of the imprecise Address Analysis for Data caches on the precision of Must analysis can be lessened by our improved Must Analysis for a class of programs. For cache levels with high associativity and high block size, our approach for Must Analysis fares even better, and hence is most suited for Multi-level cache analysis. We have also detected and rectified a flaw in the original Persistence Analysis. While the flaw has been detected by others as well, our approach is more precise than other approaches, and is able to detect more persistent blocks while still ensuring safety.

We have also extended the Abstract Interpretation based cache analysis from single level to multi-level caches. Previous efforts have either neglected the writeback effect or have completely changed the abstract lattice. Our approach just involves a simple extension to the original single-level abstract lattice. We clearly state the interdependencies between Must, May and Persistence analysis, while dealing with multi-level, non-inclusive caches. We argue that for multi-level caches, along with must analysis, to account for the filtering and writeback effect, May and Persistence analysis is also necessary at each level. To our knowledge, our work is the first effort for safely performing Persistence analysis in the presence of writebacks. We further improve the precision of Multi-level Must analysis by introducing partial and pseudo blocks in the cache hierarchy. This also justifies our decision in treating the entire cache hierarchy as one unit, and performing the various analyses on the entire hierarchy. We implemented our analysis on top of an existing prototype for WCET estimation, and showed that introducing partial blocks does result in more precise estimates for some benchmarks.

References

[1] WCET projects / benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[2] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS. Springer-Verlag*, 1996.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction and approximation of fixpoints. In *POPL*, 1977.

[4] Christoph Cullmann. Cache persistence analysis - a novel approach theory and practice. In *LCTES*, 2011.

[5] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES. Springer-Verlag*, 1998.

[6] Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[7] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *RTAS*, 2011.

[8] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. WCET analysis of multi-level set-associative date caches. In *Intl. Workshop on Worst-Case Execution Time WCET Analysis*, 2009.

[9] Frank Mueller. Timing analysis for instruction caches. In *Real-Time Systems Journal*, 2000.

[10] Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT*, 2007.

[11] Tyler Sondag and Hridesh Rajan. A more precise abstract domain for multi-level caches for tighter wcet analysis. In *RTSS*, 2010.

[12] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, May 2000.

[13] R.T. White, C.A. Healy, and et al. Timing analyses for data caches and set associative caches. In *RTAS*, 1997.