Interdependent Cache Analyses for better precision and safety

Kartik Nagar, Y.N. Srikant
Dept. of Computer Science and Automation,
Indian Institute of Science,
Bangalore, India.
Emails : kartik.nagar@csa.iisc.ernet.in, srikant@csa.iisc.ernet.in

***Abstract*—One of the challenges for accurately estimating Worst Case Execution Time(WCET) of executables is to accurately predict their cache behaviour. Various techniques have been developed to predict the cache contents at different program points to estimate the execution time of memory-accessing instructions. One of the most widely used techniques is Abstract Interpretation based Must Analysis, which determines the cache blocks guaranteed to be present in the cache, and hence provides safe estimation of cache hits and misses. However, Must Analysis is highly imprecise, and platforms using Must Analysis have been known to produce blown-up WCET estimates. In our work, we propose to use May Analysis to assist the Must Analysis cache update and make it more precise. 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 propose an improvement in the original May Analysis, to make it more precise, especially for Data Cache Analysis.**

I. 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 system 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.

The Abstract Interpretation based approach for WCET estimation combines analysis at code level and processor 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 memory blocks which will be accessed by each instruction in the program. The next step is abstract interpretation based cache analysis, which uses the accessed memory 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 capacity, 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 worst case execution times of basic blocks, subject to structural constraints and loop bounds(thus incorporating the program structure), to determine the worst case exection 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[1] 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 Abstract Interpretation based 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 memory 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 memory 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 following approach to tackle this issue : May Analysis determines all memory 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. For a cache block in the Must cache, using May analysis, we count the maximum number of memory blocks that can be younger and then determine whether this cache block should be evicted from the Must cache. Another cache analysis which has been widely used instead of Must Analysis 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, and hence is not perfect for WCET estimation for hard real time systems. However, it can classify accesses inside loops as first miss, which means that the first access to a memory 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 Ferdinand and Wilhelm[2], 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 [3] and [4]. 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.

Finally, just as imprecise address analysis results in loss of precision for Must Analysis, it also effects the precision of May Analysis. We propose a simple but safe change in the May Analysis cache update, which will provide better precision in certain special cases, while being as precise as the original May analysis for all programs.

II. Related Work

Abstract Interpretation is one of the more successful techniques employed for Cache Analysis. The pioneering work was done by Ferdinand et al.[2], 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 intruction caches yields precise results[6]. Earlier work on Cache Analysis for data caches concentrated more on finding techniques to make the address Analysis more precise[7].

Much of the recent research activity in the area of cache analysis has been concentrated in extending it to multi-level caches and multicores. [8] proposed a natural extension of the single-level cache analysis of Ferdinand and Wilhelm to multi-level caches. They use the original Must Analysis at all cache levels, and use May and Persistence Analysis to determine which accesses will reach a particular cache level and which cache blocks may get evicted, respectively. Since they use the original Must analysis, their approach suffers from its imprecision, and its effect will be particularly felt at higher cache levels, which generally have high associativities. [9] 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'. However the updates of these live caches is similar to the Must analysis update, and their main concern is to safely estimate the writeback effect for evicted dirty cache blocks.

[3] points out the error in the original Persistence Analysis, and they argue that it arises because of mismatched cache update and join function. The original Persistence Analysis uses the cache update of Must Analysis with the join of May Analysis, which results in overapproximation of the cache contents and unsafe cache update. 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. [4] augment the persistence analysis by keeping track of younger sets of all cache blocks that may enter the cache, and use its cardinality to perform safe cache update.

III. 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 every cache block. In the middle lie the Set-associative caches, where the cache is partitioned into cache sets, which are collections of cache blocks. In set-associative caches, given a memory reference, the cache set containing the reference is unique, but within the set the cache block satisfying the reference can be placed at any location. A cache F with total size $capacity$, cache block size $linesize$ and associativity A has $blocks = capacity/linesize$ cache blocks which are distributed in $sets = blocks/A$ sets. A cache block with address $addr$ will be present in the set $addr\%sets$.

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 are more optimal for replacement. We will assume that the cache replacement policy is LRU. In each set, we will order the cache blocks by the time of their last accesses, and the position at which the cache block resides will be its age. The most recently accessed cache block will have an age of 1, while the least recently accessed block will reside at position A. This has no relation, whatsoever, with the actual physical arrangement in the cache set.

Abstract interpretation[10] 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 the 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 the state of the cache-i.e. the cache blocks present in the cache and their ages. The concrete lattice specified below is similar to the one used in [2]:

The cache F is modelled as the set $F = \{f_1, f_2 \ldots, f_{sets}\}$, where f_i denotes the ith cache set. Each cache set f_i is modelled as the set $f_i = \{l_i^1, l_i^2, \ldots, l_i^A\}$. Note that the line number also signifies the age of the memory block present in that line. Hence, l_i^1 contains the most recently used cache block in the ith set. Let $memsize$ be the size of the main memory(in bytes), then the main memory M is modelled as the set $M = \{m_1, \ldots, m_{memsize/linesize}\}$, where $m_1, \ldots, m_{memsize/linesize}$ are memory blocks of size $linesize$.

Concrete Set State is a function $s_i : f_i \rightarrow M \cup \{\perp\}$, where $\perp$ signifies the empty memory block. Let S_i be the set of all such functions.

Concrete Cache State is a function $c : F \rightarrow \bigcup_{i=1}^{sets} S_i,\ such\ that\ c(f_i) \in S_i, \forall 1 \leq i \leq sets$. Let C be the set of all such functions.

Each concrete cache state basically gives the contents of the cache and also specifies an LRU order in each cache set. Each element of the concrete lattice is a subset of C, and hence is a set of concrete cache states. The concrete transfer function for this lattice takes a concrete cache state and a memory reference, and outputs another concrete cache state. This transfer function just mimics the actual LRU update in a real cache, by making the accessed memory block the most recently accessed in its cache set, and suitably updating the ages of other cache blocks. Note that the concrete transfer function only takes single-reference accesses for concrete cache update. For the abstract lattice, we have the following definitions:

Abstract Set State is a function $\hat{s}_i : f_i \rightarrow 2^{M \cup \{\perp\}}$. In an abstract set state, we allow a cache line to contain multiple cache blocks. Again, let $\hat{S}_i$ be the set of all such functions.

Abstract Cache State is a function $\hat{c} : F \rightarrow \bigcup_{i=1}^{sets} \hat{S}_i,\ such\ that\ \hat{c}(f_i) \in \hat{S}_i, \forall 1 \leq i \leq sets$. Let $\hat{C}$ be the set of all such functions.

Using the above definitions, Must, May and Persistence Analysis can now be formally defined as follows: *Must Analysis* produces an abstract cache state at each program point such that every possible concrete cache state at that program point contains all the cache blocks present in the abstract state, and the age of a cache block in the abstract state is an upper bound on the age of the block in all concrete states. Intuitively, the Must analysis determines those cache blocks that are guaranteed to be present in the cache, and thus accesses to such blocks can be classified as always hit.

May Analysis produces an abstract cache state at each program point such that no possible concrete cache state at that program point can contain a cache block not present in the abstract state. Hence, the abstract state is in some sense a superset of all concrete states possible under all executions. Also, the age of a cache block in the abstract state is a lower bound on the age of the cache 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 cache block in the abstract state is an upper bound on the age of the cache block in all concrete states. To indicate that a cache block can be evicted, in which case its age would be $A + 1$, a special eviction line is added to the abstract set state, which contains those cache 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).

IV. 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 actaul 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.

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 a memory reference, the set f_i containing the cache block satisfying the reference 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 are all younger than 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

$shiftctr(X_i, h) = |\{m \in X_i | (\exists a, h < a \leq A,\ m \in \hat{s}_i(l_i^a)) \vee (\forall a, 1 \leq a \leq A,\ m \notin \hat{s}_i(l_i^a))\}|$. Intuitively, the above function gives the number of accessed cache blocks who will now be younger than the cache blocks present at position h in the Must cache. Hence, $shiftctr(X_i, h)$ gives the worst case increase in the age of cache blocks at position h. So the cache blocks at position h are now shifted to position $h + shiftctr(X_i, h)$, or evicted if this number is greater than the cache associativity.

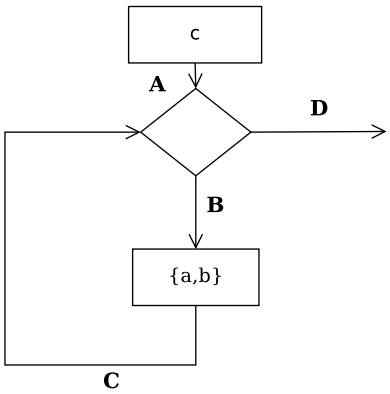

Fig. 1. CFG for Example Program

While the above transfer function is safe, it suffers from lack of precision. Consider the loop represented by the CFG in the Figure 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 3 in the must cache at program point C. Join in the Must Analysis is Intersection of the corresponding set states, while taking the maximum age. Hence, in the next fixed point iteration of Must Analysis, 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 3 at program point B. Again $shiftctr(\{a, b\}, 3) = 2$, hence the cache block c will now 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

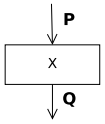

Fig. 2. Must Cache update

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 this will be the new position of cache blocks who were previously at position h in the Must cache before the access. In the example, at program point C, $Min(3 + shiftctr(\{a, b\}, 3), MaxYoung(\{a, b\}, 3)) = Min(5, 3) = 3$, hence the cache block c will remain at position 3 in the Must cache after the access.

Define $NewPos(X_i, h)$ as $Min(h + shiftctr(X_i, h), MaxYoung(X_i, h))$. Formally, the transfer function of the improved Must Analysis can be given as follows:

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

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

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

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

If X is non-singleton, then

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

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 $shiftctr < A$), 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.

V. Improved May Analysis

The Abstract lattice for May analysis is simply the set of all abstract cache states, i.e. $\hat{C}$. The join of two abstract cache states is the abstract cache state obtained by the union of all the corresponding abstract set states, while taking the minimum age. The transfer function $\hat{U}^{\hat{C}}_{May}$ for the abstract cache in the May analysis takes the cache state and the access and returns a new cache state. It simply applies the transfer function for abstract sets, $\hat{U}^{\hat{S}_i}_{May}$ to each set along with the accessed memory blocks mapping to the set. Below is the transfer function for single-reference accesses[2]:

$$\hat{U}^{\hat{S}_i}_{May}(\hat{s}_i, \{m\}) = \begin{cases} l_i^1 \mapsto \{m\}, \\ l_i^a \mapsto \hat{s}_i(l_i^{a-1}),\ 2 \leq a \leq h \\ l_i^{h+1} \mapsto \hat{s}_i(l_i^{h+1}) \cup (\hat{s}_i(l_i^h) - m), \\ l_i^b \mapsto \hat{s}_i(l_i^b),\ h+2 \leq b \leq A; \\ \quad if \exists h, 1 \leq h \leq A, m \in \hat{s}_i(l_i^h), \\ l_i^1 \mapsto \{m\}, \\ l_i^a \mapsto \hat{s}_i(l_i^{a-1}),\ 2 \leq a \leq A; \\ \quad otherwise \end{cases}$$

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

This transfer function takes as input only one memory reference, 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 be to any cache block 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 already present in any of the cache sets, as May analysis needs to maintain the lower bound on ages. As a result, the transfer function for multi-reference access proposed in [2] simply brings all the cache blocks containing the references to the first position in their respective sets, without increasing the ages of any of the cache blocks already present in the May cache.

Consider the case where all the accessed cache blocks map to the same cache set. Now, consider the accessed cache block which is already present in the May cache and which has the minimum age amongst all the accessed blocks. We can safely increase the ages of all the cache blocks in the cache set having age less than this minimum age accessed cache block. This is safe because it is guaranteed that one of the cache blocks in the cache set will actually be accessed, and hence those cache blocks which are younger than the youngest accessed block will definitely see an increase in their ages. If all the accessed cache blocks are not present in the May cache, we can safely increase the age of all cache blocks in the cache set by 1. Formally the transfer function is:

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

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

Note that the above transfer function can be used only in the case when all the accessed cache blocks map to the same cache set. Otherwise, we use the normal transfer function. By recognizing the special case where we can safely increase the ages of cache blocks, our improved transfer function maintains a tighter lower bound on the ages.

VI. 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 abstract cache state determined by Persistence Analysis 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 cache 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 [2] 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 cachhe along all execution paths. May Analysis can be used to determine this number just as it was used for 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, for a general multi-reference access: The transfer function for the abstract cache state, $U^{\hat{C}}_{Per}$, applies the transfer function for abstract set state, $U^{\hat{S}_i}_{Per}$ to all sets, which takes as input the abstract set state and the cache blocks in the access mapping to the set. Let $\hat{c}^{Must}$ and $\hat{c}^{May}$ be Must and May caches respectively at the program point(before their own updates). Let X_i be set of the accessed cache blocks mapped to set i.

We use the function $shiftctr(X_i, h) = |\{m \in X_i | (\exists a, h < a \leq A,\ m \in \hat{s}_i^{Must}(l_i^a)) \vee (\forall a, 1 \leq a \leq A,\ m \notin \hat{s}_i^{Must}(l_i^a))\}|$. 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) will now become younger. The 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.

$MaxYoung(X_i, h) = |X_i \cup \bigsqcup_{a=1}^{h} (\hat{s}_i^{May}(l_i^a) - X_i)|$ 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 $NewPos(X_i, h) = Min(h + shiftctr(X_i, h), MaxYoung(X_i, h))$. If X_i is singleton, then the accessed cache block will be brought in the first position while the rest of cache blocks in $\hat{s}_i$ will follow the above rule. If X_i is non-singleton, all the accessed blocks cannot be brought into the first position. Let X_i' be the set of cache blocks in X_i which are not present in $\hat{s}_i$ in the Persistence cache and let $z = |X_i'|$. Then, the cache blocks in X_i' will be brought into position z in the persistence cache. For the cache blocks that are present 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, along with the ages of all un-accessed cache blocks in $\hat{s}_i$.

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

$$= \begin{cases} \begin{cases} l_i^a \mapsto \bot, \quad 1 \leq a < z, \\ l_i^z \mapsto X_i', \\ l_i^c \mapsto \bigsqcup_{\substack{b, \\ NewPos(X_i,b)=c}} \hat{s}_i(l_i^b), \quad z < c \leq A \\ \quad if\ z \leq A \end{cases} \\ \begin{cases} l_i^a \mapsto \bot, \quad 1 \leq a \leq A \\ l_i^\top \mapsto X_i' \cup \bigsqcup_{a=1}^{A} \hat{s}_i(l_i^a) \\ \quad otherwise \end{cases} \end{cases}$$

The problem with the original Persistence Analysis as well as approaches to overcome it have been proposed in [3] and [4]. Cullmann's approach[3] 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 oldest cache blocks 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.

For the program represented by the CFG shown in Figure 3, 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 is present in the persistence cache, and is also 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

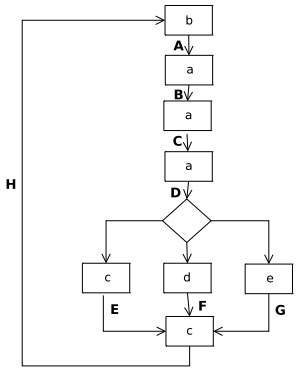

Fig. 3. CFG for Example Program

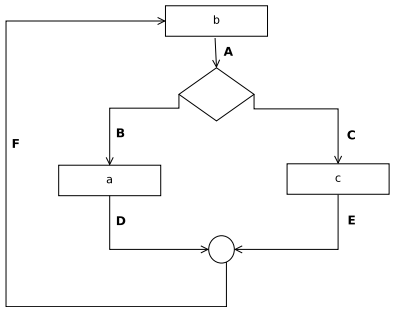

Fig. 4. CFG for Example Program

to block c by Cullmann's analysis. Hence, cache block b is classifed 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 [4] 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 4, 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 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 $1 + shiftctr = 1 + 1 = 2$. 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.

VII. Experimental evaluation

We have implemented our improved Must Analysis on top of the prototype for WCET estimation used by [1]. 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 have not made 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 hit/miss. We have replaced this part with our improved Must analysis. The configuration of the cache memory used is : Associativity = 4, Cache Block Size = 32 bytes, Cache sets = 128. We also assume the following latencies : Read/Write hit latency = 1 cycle, Read/Write Miss latency = 6 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 26360 cycles, while the WCET estimated using improved Must analysis is 26164 cycles. The difference 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 greatly 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 [1]. 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 aggregrate information from all the members of the collections will have nested loops. The inner loops would analyze individual members, while the outer loop would aggregrate information. For such programs, our analysis would be able to provide better estimates.

As stated earlier, our approach would be highly beneficial for caches with high associativity and high block size. The high block size may result in lesser number of cache blocks accessed per cache set in a multi-reference access(especially for array accesses). Higher level caches in a multi-level cache hierarchy generally have both high associativity and large block size. While doing Multi-level cache analysis, due to the filtering effect of lower level caches, accesses satisfied by the lower levels do not reach higher level caches. Hence, the number of accessed cache blocks in a multi-reference access which reach the higher levels would be low, further decreased by the higher block sizes. All these circumstances along with the high associativity would highly favour our analysis. In fact, we intend to use the Improved Must, May and Persistence Analysis in our Multi-level Cache Analysis model, which is in the implementation and testing phase. Our approach does have higher space and time complexity for single level caches, as it requires all the both Must and May analyses to run simultaneously. However, for multi-level caches, to account for the filtering effect of the cache hierarchy and the writeback effect, the original Must, May and Persistence analysis would be required at all cache levels[8]. Thus, our approach does not add to the space and time complexity for multi-level cache analysis.

VIII. Proof of Safety for Improved Must Analysis

The improved Must Analysis differs from the original Must analysis in two places : the abstract lattice and the transfer function. 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. 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, the concretization function 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 the concrete cache state must be present in 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:

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

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

The abstraction function takes as input a set of concrete cache states and outputs the corresponding Must and May cache. Briefly, the Galois connection property can be expressed as follows : Given a set of concrete cache states(S), first apply the abstraction function to obtain the Must and May cache, and then apply the concretization function to get another set of concrete cache states(S'). Then S must be a subset of S', which ensures that none of the concrete cache states in S are lost during the abstraction process. Similarly, given a Must and May cache, on applying the concretization function to get a set of concrete cache states, and then applying the abstraction function on this set gives the same Must and May cache. To ensure the Galois connection, the following natural definition of the abstraction function suffices : 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. Now, with these definitions of the abstraction and concretization function, it is clear that the Galois connection properties will be satisfied.

Before proving the correctness of the transfer function, we will prove the following **Lemma** : 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 the concretization function 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 the concretization function 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 conctribute 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.

IX. 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 well-used 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. Moreover, for multi-level caches, even the original Must analysis requires May and Persistence analyses at all levels. Hence, using the Improved Must analysis for multi-level caches will have the same time and space complexity as the original Must 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. Finally, we have also proposed a slight adjustment to the original May analysis to improve its precision for Data caches.

References

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

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

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

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

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

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

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

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

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

[10] 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.