Path sensitive cache analysis using cache miss paths

Kartik Nagar and Y N Srikant

Indian Institute of Science,
Bangalore, India.
{kartik.nagar, srikant}@csa.iisc.ernet.in

Abstract. Cache analysis plays a very important role in obtaining precise Worst Case Execution Time (WCET) estimates of programs for real-time systems. While Abstract Interpretation based approaches are almost universally used for cache analysis, they fail to take advantage of its unique requirement: it is not necessary to find the guaranteed cache behavior that holds across all executions of a program. We only need the cache behavior along one particular program path, which is the path with the maximum execution time. In this work, we introduce the concept of cache miss paths, which allows us to use the worst-case path information to improve the precision of AI-based cache analysis. We use Abstract Interpretation to determine the cache miss paths, and then integrate them in the IPET formulation. An added advantage is that this further allows us to use infeasible path information for cache analysis. Experimentally, our approach gives more precise WCETs as compared to AI-based cache analysis, and we also provide techniques to trade-off analysis time with precision to provide scalability.

1 Introduction

Real time systems need a safe estimate of the execution time of a program, which should never be exceeded by any of the program's actual runs. Modern architectures use caches, out-of-order pipelines and all kinds of speculation to make programs run faster, and this has a significant impact on their execution times. The Worst Case Execution Time (WCET) of a program on a particular architecture is defined as the maximum execution time of the program across all its possible runs on that architecture. Ideally, we would like to find this WCET, but theoretically, it is not possible. Timing analysis techniques try to find an upper bound on the WCET of programs. Since the scheduler in a real-time system is likely to assign computational resources such as the processor to a program for the entire duration of its estimated WCET, it is desirable that the upper bound be as close as possible to the actual WCET to avoid wastage.

Caches have a major impact on execution time of programs, because of the huge difference in cache access latency and main memory latency in current architectures. The execution time of a memory-accessing instruction can change by a factor of 100, depending on whether the access hits the cache or goes to the

main memory. Cache analysis techniques try to find the accesses in a program which will hit the cache, so that the cache latency can be used for those accesses while finding the WCET.

The cache behavior of an instruction depends on the sequence of accesses made to the cache by the program before the instruction, which in turn depends on the program path taken to reach the instruction. Since we want to find the WCET, we would be interested only in the worst-case (WC) program path, which is the program path with the maximum execution time. If the WC path is known, then this information can be used to determine the accurate cache behavior of the accesses along it. However, this results in a 'boot-strapping' problem, because to find the WC path, we must know its execution time, which requires knowledge of the cache hits along the path. But, cache hits along the worst-case path cannot be accurately estimated unless we know the path leading to the cache accesses.

One way to solve this problem is to find the accesses which hit the cache irrespective of the program path taken to reach them. This is the approach taken by the almost universally used Abstract Interpretation (AI) based cache analysis [1], which finds the guaranteed cache hits in a program. All the accesses which are not guaranteed to hit the cache are classified as misses, and the resulting hit-miss classification is used to determine the latency of memory-accessing instructions. Once the execution time of each individual instruction is determined, the Implicit Path Enumeration Technique (IPET) [2] can be used to find the worst-case path in the program. IPET generates an Integer Linear Program (ILP), whose optimal solution encodes the worst-case path.

In our work, we target those accesses which will hit the cache along the worst-case path, but not necessarily along all other paths. To do this, we integrate a limited amount of cache analysis into the IPET formulation, thus taking advantage of the knowledge about the worst-case path to classify certain cache accesses. We propose the concept of cache miss paths of an access, *which are simply those program paths along which the access will suffer a cache miss*. We concentrate on the accesses which are not classified as hit by AI-based cache analysis, and find the cache miss paths of these accesses. We then integrate the miss paths into the IPET formulation to ensure that a cache access will be considered a miss only if the worst-case path contains a miss path of the access. Previously, we have used a similar concept of cache hit paths, to determine the effect of shared cache interference on cache hits [6].

There are many advantages of integrating cache analysis into the IPET formulation. Most of the imprecision of AI-based cache analysis stems from the fact it requires an access to hit the cache along all paths leading to the access. However, an access can be safely classified as hit, if it experiences a cache hit along the worst-case path. This will only happen if the worst-case path does not contain a miss path of the access.

Moreover, some programs have infeasible paths, which generally take the form of conflicting basic blocks, which will never be executed together. Information about infeasible paths can be obtained separately using abstract execution

[3], SMT solvers [5], model checking [7], etc. and is part of the program flow analysis stage. This stage generally occurs before timing analysis, and is primarily used to determine the program CFG, loop bounds, etc. A number of works have integrated infeasible path information into the IPET formulation, ensuring that infeasible paths will be ignored while finding the worst-case path in the program ([4], [5]). Since we integrate cache miss paths into the IPET formulation, our approach will have the added advantage of utilizing infeasible path information for cache analysis. Previous works ([7], [8]) have shown that substantial precision improvement in the WCET can be achieved by utilizing infeasible path information during cache analysis.

Experimentally, we found that our approach gave lower WCETs for 9 out of 27 Mälardalen benchmarks [15], as compared to AI-based cache analysis, with an average precision improvement of 22.54 % and with negligible increase in analysis time. Another advantage of our analysis is that it subsumes persistence analysis, which is used to classify accesses inside loops as First-Miss. Since our approach adds some portion of cache analysis to the IPET formulation and thus increases the size of the generated ILP and hence the time required to solve it, we also provide two methods to control the analysis time. Since the number of extra variables/constraints added to the ILP depend on the size and total number of cache miss paths, we allow user-controlled upper bounds on these values. We experimented with different bounds, and found that substantial precision improvement can be obtained even with very low bounds on the size of cache miss paths. We also propose a CEGAR-like strategy which introduces cache miss paths of accesses into the ILP one access at a time, by selecting the cache access which suffers a hit along the worst case path, but has the maximum number of cache misses in the ILP. This allows us to stop the refinement of the WCET at any iteration of the CEGAR loop, and thus trade-off precision with analysis time.

2 Related Work

Few works have looked at the impact of infeasible paths on cache analysis ([7], [8]). However, none of these works have directly integrated cache analysis with the IPET formulation. In [7], the authors instrument the code by introducing variables to count the number of cache misses suffered by accesses which are not classified as Hit by AI-based analysis, and then use model checking to verify assertions on these variables. This approach requires code instrumentation, and is also known to have very high analysis time [8]. There is no way to reuse infeasible path information, which may already have been determined separately during the program flow analysis stage. Moreover, no information about the worst-case path is used to refine the cache analysis, and hence the approach will work only when there are actual infeasible paths (and model checking can identify them).

[8] modifies the AI-based approach for cache analysis, by annotating cache states with logic formulae, corresponding to the partial path along which the cache state would be realized. However, their work can only handle limited types

of infeasible paths. In particular, they only consider a maximum of two conflicting basic blocks, and because of their abstract lattice, the conflicting basic blocks must be close to each other in the program CFG (there cannot be more than one join between two conflicting basic blocks). Moreover, they also ignore the worst-case path information and consider only the impact of infeasible paths on cache analysis.

There have also been previous efforts in performing complete cache analysis using the ILP-based IPET formulation, most notably, the CSTG-based approach proposed by Lee et al. [9]. In this work, the authors first generate the cache state transition graph (CSTG), whose nodes are all possible cache states generated during execution, and edges show the transition between the cache states. Integer variables are introduced in the IPET ILP for all the edges in the CSTG, and these variables are then used to provide an upper bound on the number of hits experienced by accesses. However, as has been noted by [10], this approach introduces an exponentially large number of variables, and significantly increases the size of the ILP, rendering it non-practical even for small programs.

In our approach, we also introduce new variables for a cache access and their miss paths, but only for those accesses which are not classified as Hit by the AI-based cache analysis. In [9], the authors essentially find cache hit paths in the CSTG, constrain the number of cache hits using the variables in the CSTG, and then link these variables with the basic block counts in the CFG. In our case, we directly find the cache miss paths in the CFG, using an AI-based approach, and hence do not need to generate the CSTG.

3 Foundations

Caches store a small subset of main memory closer to the processor, and provide fast access to its contents. To take advantage of spatial locality of memory accesses, all data transfer between the main memory and cache takes place in equal-sized chunks called memory blocks (or cache blocks). To enable fast lookup, caches are further divided into cache sets. For an $A-way$ set associative cache, each cache set can contain maximum of A cache blocks. Given an access to a cache block, the cache subsystem first finds the unique cache set containing the accessed cache block, searches for it among the (at most) A cache blocks in the cache set, and if it is not present, brings it from the main memory.

Since the total number of cache blocks mapped to a cache set will usually be much greater than the associativity (A), the cache replacement policy decides which cache block should be evicted, if the cache set is full and a new cache block has to be brought in. The Least Recently Used (LRU) policy orders all cache blocks in a cache set according to their most recent accesses, and evicts the cache block which was accessed farthest in the past. We will assume LRU replacement policy in our work. We also assume a timing anomaly-free architecture.

Must Analysis [1] for caches produces abstract cache states at each program point, which contains those cache blocks which are guaranteed to be in the actual cache at the program point across all executions of the program. It is used to

classify accesses as cache hits. Similarly, May analysis produces abstract cache states which contain cache blocks which may be present in the actual cache during some execution. It is used to classify accesses as cache misses.

4 Cache miss paths

To keep things simple, we will now assume a single-level instruction cache. However, our approach can in general be applied to any level in a multi-level instruction cache hierarchy. First we formally define cache miss paths.

Given an instruction a which accesses the cache block m mapped to cache set s, a path π in the CFG of the program is called a **cache miss path** of a if

1. π ends with instruction a and has no other accesses to m other than a, and
2. either the number of distinct cache blocks mapped to s and accessed by instructions in π is equal to $A+1$ (where A is the cache associativity), or π begins from the start of the program.

Note that there are only two possible ways that an instruction a can suffer a cache miss: the accessed cache block m has not been brought into the cache at all from the start of the program, or it was brought but then evicted before a. Both these scenarios are captured in the definition of cache miss path. If miss path π begins from the start of the program, then since a is the only instruction which accesses m, m will not be brought into the cache along the path π, before a. Otherwise, if π contains accesses to $A+1$ distinct cache blocks, then the instructions of π before a must have accessed A distinct cache blocks different from m. Hence, by the time a is executed, m is guaranteed to be not present in the cache. Since the cache miss paths consider both the reasons for a cache miss, this shows that an access suffers a cache miss if and only if execution passes through a miss path of the access.

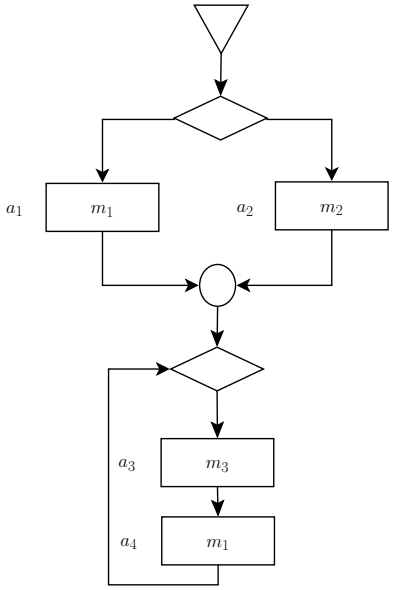

Fig. 1.

Consider the example program in Figure 1, which shows four cache accesses a_1, a_2, a_3, a_4, accessing cache blocks m_1, m_2, m_3, m_1 respectively. Assume the cache has an associativity of 2, and also assume that m_1, m_2, m_3 map to the same cache set. Let us concentrate on the access a_4, which accesses cache block m_1, and consider the program paths leading to this access.

The path $\triangleright - a_2 - a_3 - a_4$ begins from the start of the program, and does not access m_1 until a_4. Hence, this is a cache miss path of a_4. On the other hand, the path $a_1 - a_3 - a_4$ begins with an access to m_1, and accesses only 2

distinct cache blocks. Execution along this path will result in a cache hit for a_4, and hence it is not a cache miss path. The path $a_4 - a_3 - a_4$ lies entirely within the loop, and is again not a cache miss path, as it accesses only 2 distinct cache blocks. Hence, a_4 does not have a cache miss path entirely within the loop, and so is guaranteed to be a cache hit for all iterations except the first. In addition, it will be a cache hit in the first iteration if the worst-case path passes through a_1.

The miss path of an access can be determined by traversing backward in the CFG starting from the access and keeping track of the cache blocks encountered along different paths. If the number of distinct cache blocks encountered along a path (without encountering the accessed cache block) becomes greater than cache associativity, the path can be deemed as a miss path and further accounting of cache blocks along the path can be stopped. On the other hand, if the accessed cache block itself is encountered on a path, then such paths can be discarded, as they cannot become cache miss paths of the access. For accesses inside loops, we may have to take the back-edges (in the reverse direction) more than once to find all cache miss-paths.

5 AI formulation

We use Abstract Interpretation to find the cache miss paths of accesses. We concentrate only on those accesses which are not classified as Hit by the AI-based Must analysis, or Miss by May cache analysis. Let Acc be the set of all cache accesses made by the program. Since cache accesses are made by the instructions in a program, we use the terms 'access' and 'instruction' interchangeably . Let Acc_{NC} be the set of accesses not classified as Hit or Miss ($Acc_{NC} \subseteq Acc$). Accesses in Acc_{NC} will have at least one cache-miss path. Each cache-miss path π can be viewed as a set of accesses which satisfies the required properties (it is not necessary to keep track of their sequence, because if all instructions in a miss path of an access are executed, then irrespective of their order of execution, the access will suffer a miss). We use the special symbol $\dashv$ to indicate that the cache-miss path has ended, which means that it has accessed $A+1$ distinct cache blocks, and no new accesses should be added to it. Hence $\pi \in \mathcal{P}(Acc \cup \{\dashv\})$. ($\mathcal{P}(S)$ denotes the powerset of S).

An access can have multiple cache-miss paths, and hence we maintain a set of miss paths for every access in Acc_{NC}. Our abstract lattice is the set of functions $F = \{f | f : Acc_{NC} \rightarrow \mathcal{P}(\mathcal{P}(Acc \cup \{\dashv\}))\}$. For $f_1, f_2 \in F$, we say that $f_1 \preccurlyeq f_2$ if and only if $\forall a \in Acc_{NC}, f_1(a) \subseteq f_2(a)$. This is the standard power-set lattice formulation, with the join being defined as point-wise union: $(f_1 \sqcup f_2)(a) = f_1(a) \cup f_2(a)$.

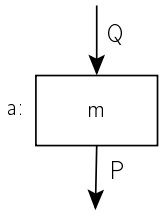

Fig. 2.

We now define the transfer function. Let $cb(a)$ and $cs(a)$ denote the cache block and cache set accessed by instruction a respectively. Given a set of instructions, π, $dist\_blocks(\pi)$ gives the number of distinct cache blocks accessed by instructions in π. Hence, $dist\_blocks(\pi) =$

$|\{cb(a)|a \in \pi\}|$. The direction of the analysis will be backward, assigning an empty set of cache-miss paths initially for all accesses in Acc_{NC}. Since all miss-paths of an access end with the access itself, as soon as an access in Acc_{NC} is encountered, the collection of its miss-paths will begin. As shown in Figure 2, suppose instruction a accesses cache block m mapped to cache set s. The transfer function $\mathcal{T}_{PQ}$ for this instruction takes as input function $f_P \in F$, and outputs function f_Q:

$$f_Q(a') = \begin{cases} \{\{a\}\} \cup \{\pi : \pi \in f_P(a) \wedge \dashv \in \pi\}, & \text{if } a = a', \\ \{\pi : \pi \in f_P(a') \wedge \dashv \in \pi\}, & \text{else if } m = cb(a') \\ \{\pi \cup \{a\} : \pi \in f_P(a') \wedge \dashv \notin \pi \wedge dist\_blocks(\pi \cup \{a\}) \leq A\} \\ \cup \{\pi \cup \{a, \dashv\} : \pi \in f_P(a') \wedge \dashv \notin \pi \wedge dist\_blocks(\pi \cup \{a\}) > A\} \\ \cup \{\pi : \pi \in f_P(a') \wedge \dashv \in \pi\}, & \\ & \text{else if } cs(a) = cs(a') \\ f_P(a') & \text{otherwise} \end{cases}$$

First, we consider the miss paths of the access a itself (if $a \in Acc_{NC}$). We add the singleton path $\{a\}$ to start the collection of miss paths of a, while any existing paths of a which have been already been ended are retained. An existing path of a will be present in $f_P(a)$ when a is inside a loop, and it has already been encountered once during an earlier AI iteration. If an existing path of a has not ended, then it would have accessed at most A distinct cache blocks (including m). The access a will bring m to the cache, but there would not be enough cache blocks in this path to evict m before the next access by a. Such a path will never be a cache-miss path of a, and hence must be discarded.

In the second case, we consider the paths of those instructions a' which access the same cache block m. Since this is a backward analysis, any existing path of a' which reaches a would indicate that there is a path from a to a'. Since a brings cache block m into the cache, any path from a to a' will be a cache miss path only if there are enough cache blocks accessed on it to evict m. In this case, only the existing paths which have already been ended will be retained, while all other paths of a' will be discarded. This is because any path which has already been ended would have accessed $A + 1$ distinct cache blocks, or A distinct cache blocks other than m. Hence, m would have been evicted by the time a' is executed. On the other hand, any path which has not been ended will not have accessed enough cache blocks to evict m.

In the third case, we consider the paths of instruction a' which access a different cache block $cb(a')$, mapped to the same cache set $cs(a)$. In this case, cache block m will conflict with $cb(a')$ and therefore the access a should be added to any existing path of a'. In addition, if the number of distinct cache blocks accessed along a path (after adding a) becomes greater than A, such a path would now become a cache-miss path and hence can be ended. Also, miss paths which have already ended are retained without any modification. Finally, in the last case, paths of instructions which do not access the cache set $cs(a)$ are not modified.

It is easy to see that the transfer function is monotonic, as it operates separately on every path of a cache access. It either adds a new path, discards existing paths or adds new accesses to a path, but this depends solely on the properties of the access or the path itself. We also give a formal proof in Section 7. Moreover, the abstract lattice F is finite, and hence termination of the analysis is guaranteed. All the cache-miss paths of accesses in Acc_{NC} will be gathered at the start of the program in the fixpoint solution.

6 ILP formulation

Table 1. Notation

Symbol	Explanation
y_i	Integer variable storing the execution count of basic block b_i
x_{ij}	Integer variable storing the total number of cache misses suffered by instruction a_{ij}
x_{ij}^{π}	Integer variable storing the number of cache misses of instruction a_{ij} along cache-miss path π
w_{ij}	Integer variable storing the execution count of edge between basic blocks b_i and b_j
e_i	Execution time of basic block b_i assuming NC-instructions as cache hits
c_p	Cache miss penalty

We now integrate the cache-miss paths into the IPET formulation [2]. We introduce new integer variables for every access in Acc_{NC}, as well as for each cache-miss path of these accesses. The number of cache misses suffered along a cache-miss path will be constrained by the execution counts of the accesses along the path.

Let $b_1, \ldots, b_n$ be the basic blocks of the program, and let $a_{i1}, \ldots, a_{il_i}$ be the instructions in b_i which are not classified (NC) as Hit or Miss. Table 1 contains all the notations used in the ILP. Note that e_i is the estimated execution time of b_i obtained by using the AI-based cache hit-miss classification, and assuming cache hit latency for all instructions classified as NC. Let $BB(a)$ denote the index of the basic block containing instruction a.

We first give a brief description of the IPET formulation. For each basic block b_i, y_i stores the execution count of this basic block on the worst-case path. For an edge between basic blocks b_i and b_j in the CFG, the variable w_{ij} stores the number of times execution passes from b_i to b_j on the worst case path. The objective is to find the worst-case path, i.e. the execution counts of basic blocks which maximizes the execution time of the program. The execution counts are constrained by the program structure, which basically places the restriction that the number of times execution enters a basic block (through an incoming edge in the CFG) must be the same as the number of times execution leaves the basic

block (through an outgoing edge), and this will also be the execution count of the basic block. Hence, the sum of the w variables for all incoming edges to a basic block will be the same as the sum of w variables for all outgoing edges. Following is our proposed ILP, which is based on the IPET formulation:

$$\text{Maximize} \sum_{i=1}^{n} (e_i y_i + \sum_{j=1}^{l_i} c_p x_{ij}) \tag{1}$$

subject to

$$\forall i, \quad y_i = \sum_{j \in \text{pred}(b_i)} w_{ji} = \sum_{k \in \text{succ}(b_i)} w_{ik} \tag{2}$$

$$\forall i \forall j, \quad x_{ij} \leq y_i \tag{3}$$

$$\forall i \forall j, \quad x_{ij} \leq \sum_{\text{all miss paths } \pi \text{ of } a_{ij}} x_{ij}^{\pi} \tag{4}$$

$\forall i \forall j \forall \pi$ where π is a miss path of $x_{ij}, \pi = \{a_{\pi 1}, a_{\pi 2}, \ldots, a_{\pi k}\}$

$$x_{ij}^{\pi} \leq y_{BB(a_{\pi 1})} \tag{5}$$

$$\vdots$$

$$x_{ij}^{\pi} \leq y_{BB(a_{\pi k})}$$

Loop Constraints ...

Infeasible path constraints ...

The product $e_i y_i$ is the contribution of b_i to the execution time of the program, assuming that all NC-instructions are cache hits. The variable x_{ij} accounts for the cache misses suffered by access a_{ij}. Each cache miss causes an additional execution time of c_p. Hence, the objective function is the sum of the total execution times of all basic blocks on the worst-case path (Equation 1). Equation 2 encodes the flow constraint for each basic block. The maximum number of misses suffered by an access will be the execution count of the basic block containing the access, and this gives a trivial upper bound on x_{ij} (Equation 3). For each miss path π of a_{ij}, the variable x_{ij}^{π} counts the number of misses suffered by a_{ij} along π.

For a_{ij} to experience a miss along miss path π, all the accesses in the miss-path should happen. Hence, x_{ij}^{π} is upper-bounded by the execution counts of all the basic blocks which contain the instructions present in π (Equation 5 onwards). If an access has multiple cache miss paths, then it can suffer a miss along any of its miss paths. Moreover, for an access inside a loop, multiple cache-miss paths of the access may be executed (for example, in different iterations). Hence, the total number of misses suffered by an access (x_{ij}) is bounded by the sum of its x_{ij}^{π} variables (Equation 4). Since the two notions of an access suffering a cache miss, and its cache-miss path being executed are equivalent, and the AI-based approach will determine all the cache miss paths, the above ILP is guaranteed to account for all the cache misses suffered by an access.

In addition to loop constraints, which will bound the execution count of loop headers, infeasible path constraints can also be provided in the above ILP. An infeasible path generally takes the form of a set of conflicting basic blocks, which will never be executed together. The constraints will place an upper bound on the sum of the execution counts (y_i) of conflicting basic blocks. By appending them to the above ILP, we not only guarantee that the worst case path will not contain the infeasible path, but also that no cache miss will be caused due to it. If the cache miss path of an access is infeasible, then the contribution of cache misses along such a path would become zero.

Multi-level cache hierarchy: Our technique can be applied at any level in a multi-level cache hierarchy. To apply the technique at level x in a cache hierarchy with L levels, *Acc* will consist of all accesses which may reach level x, while Acc_{NC} will consist of accesses which are not classified as Hit at level x. The AI-based approach to find the cache miss paths can be directly applied, except that a miss-path of an access can be discarded in the transfer function, only when there is a guaranteed access to the same cache block, and the miss-path has not ended. Similarly, the same ILP formulation can also be used, except that the cache miss penalty (c_p) of an access will now depend on whether it hits any cache level beyond x, or if it has to go to the main memory.

7 Scalability

Previous approaches [9] at integrating cache analysis into the IPET formulation have struggled with the exponential increase in the size of the ILP due to the addition of extra variables and constraints. However, these approaches did not perform any prior cache analysis, and hence relied solely on the ILP for the hit-miss classification of all cache accesses made by the program. In our case, we are weeding out the cache accesses classified as Hit or Miss by the AI-based cache analysis, and only rely upon the ILP for the remaining accesses.

However, we are also introducing extra variables for each cache-miss path of NC accesses, and extra constraints for each basic block present in a cache-miss path. In general, the number of cache miss paths, and their sizes can be exponentially large in the size of the program. Even though a cache-miss path will access at most $A+1$ distinct cache blocks, this does not place any restrictions on its size, as multiple instructions could access the same cache block. Hence, we propose two changes in the original AI-formulation to limit the size of generated cache-miss paths and hence the size of the final ILP. *We note that this is main advantage of using cache miss paths, as more abstractions can be used to speed up the analysis time, at the cost of precision, but without jeopardizing the safety requirements of cache analysis.* There could be other ways in which cache miss paths can be combined/ignored without under-estimating the number of cache misses, to tradeoff precision with analysis time.

We first modify the transfer function to limit the size of each miss-path to a maximum threshold (T). For miss path π of access a, let $|\pi|$ denote its size, i.e., the number of accesses present in π, excluding the access a. Referring back to the original transfer function defined in Section 5, an access a was added to

a miss-path of access a' if they accessed different cache blocks mapped to the same cache set and the miss-path had not ended. In the new transfer function, we will add a new access to an existing miss-path only if the size of the expanded miss-path does not exceed T. A miss-path will be ended when its size reaches T, even though the number of distinct cache blocks accessed on the path may not have reached $A + 1$. The following equation shows the only change in the transfer function $\mathcal{T}_{PQ}$ of an access a, proposed in Section 5:

$$f_Q(a') =$$

$$\begin{cases} \{\pi \cup \{a\} : \pi \in f_P(a') \wedge \dashv \notin \pi \wedge dist\_blocks(\pi \cup \{a\}) \leq A \ \wedge |\pi \cup \{a\}| < T\} \\ \cup \{\pi \cup \{a, \dashv\} : \pi \in f_P(a') \wedge \dashv \notin \pi \wedge (dist\_blocks(\pi \cup \{a\}) > A \vee |\pi \cup \{a\}| = T)\} \\ \cup \{\pi : \pi \in f_P(a') \wedge \dashv \in \pi\} \\ \qquad\qquad \text{if } cb(a) \neq cb(a') \text{ and } cs(a) = cs(a') \end{cases}$$

The new transfer function ends a miss-path either when its size becomes equal to the threshold, or if it accesses more than A distinct cache blocks. Note that there is no restriction on the threshold T, and it can take any value. It is possible that paths determined using above restriction may not actually be cache miss paths. However, we will not lose any actual cache miss paths, because if the length of any actual miss path is greater than T, then its sub-path of length T will be considered as a miss-path by the analysis. This is safe in the context of the ILP as well, since an upper bound on the number of cache misses along a miss path, obtained using the entire path, will be smaller than the upper bound obtained using only its sub-path. Hence, we will only be overestimating the number of misses along the shortened miss-path.

The analysis will lose precision with lower values of T, as more paths which access less than $A + 1$ distinct cache blocks may be treated as cache miss-paths, and the upper bound on the number of cache misses along the more shortened paths will also not be precise. In our experiments, we were able to achieve good precision by setting T to be twice the cache associativity. By limiting the maximum size of cache-miss path, we also decrease the number of cache-miss paths of an access.

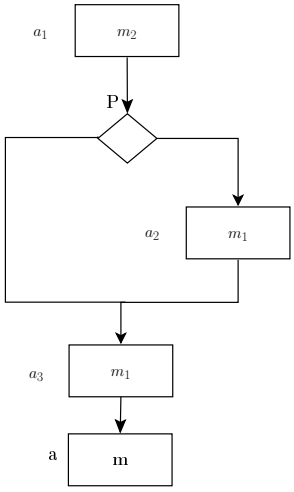

Fig. 3.

The other modification is made to the join in the abstract lattice. In the original formulation, at the join points, we simply took the union of the incoming miss paths for every cache access. However, some miss paths may be entirely contained in other miss paths, and in such a scenario, it is safe to discard the larger miss paths, if they access the same number of distinct cache blocks as the smaller miss paths present inside them.

For example, consider the program fragment shown in Figure 3 which shows cache accesses a, a_1, a_2 and a_3 all accessing the same cache set. Assume that the cache associativity is 2. We concentrate on the miss paths of the access a to cache block m. While finding the fixpoint (in the backward direction), at program point P, we will get two different paths of a, $\pi_1 = \{a_3, a\}$ and $\pi_2 = \{a_2, a_3, a\}$. Clearly $\pi_1 \subseteq \pi_2$, and both paths access the same number of distinct cache blocks. In this case, it is safe to discard π_2 during the join, because if π_2 were to eventually become a cache miss-path by adding some accesses (for example, by adding a_1), then the same accesses will also make π_1 a cache miss-path. Moreover, in the ILP formulation, $x_a^{\pi_1} \geq x_a^{\pi_2}$. Hence, the contribution of cache misses along π_2 will be accounted for by the path π_1.

Experimentally, we have found that such scenarios occur very often in benchmarks, and using the modified join can substantially decrease the number of miss-paths. Moreover, this also has a positive impact on the ILP, as we will not count the same cache miss multiple times along different miss paths. In the example, the access a suffer a cache miss along both the miss paths $\{a_1, a_3, a\}$ and $\{a_1, a_2, a_3, a\}$. In actual execution, a will only suffer one cache miss, but if we did not discard π_2, then we would have counted two misses along both the miss-paths in the ILP. We now give a formal definition of the join. Given $f_1, f_2 \in F$, we define $f_1 \sqcup f_2$ as follows:

$$\begin{aligned}\forall a \in Acc_{NC}, (f_1 \sqcup f_2)(a) = (f_1(a) \cup f_2(a)) \setminus \{\pi \in (f_1(a) \cup f_2(a)) : |\pi| < T-1 \\ \wedge (\exists \pi' \in (f_1(a) \cup f_2(a)) \setminus \{\pi\}, (\pi' \subseteq \pi) \\ \wedge (dist\_blocks(\pi) = dist\_blocks(\pi')))\}\end{aligned}$$

From the pointwise union of miss paths from f_1 and f_2, we remove those miss paths which contain less than $T-1$ accesses and for which a sub-path accessing the same number of distinct cache blocks is also present in the union. Note that both the miss path which is being removed and its subpath will also access the same set of cache blocks. The ordering relation $\preccurlyeq$ in the lattice F now becomes: $f_1 \preccurlyeq f_2 \Leftrightarrow \forall a \in Acc_{NC}, \forall \pi \in f_1(a)$, if $|\pi| \geq T-1$, then $\pi \in f_2(a)$, and if $|\pi| < T-1$ then $\exists \pi' \in f_2(a), \pi' \subseteq \pi$ and $dist\_blocks(\pi) = dist\_blocks(\pi')$.

To see why the new transfer function remains monotonic with the new join, let us define a relation on the miss paths, $\sqsubseteq$. For $\pi_1, \pi_2 \in 2^{Acc \cup \{\dashv\}}$, $\pi_1 \sqsubseteq \pi_2 \Leftrightarrow \pi_1 = \pi_2 \vee (|\pi_1| < T-1 \wedge \pi_2 \subsetneq \pi_1 \wedge dist\_blocks(\pi_1) = dist\_blocks(\pi_2))$. Then, for $f_1, f_2 \in F$, $f_1 \preccurlyeq f_2 \Leftrightarrow \forall a \in Acc_{NC}, \forall \pi_1 \in f_1(a), \exists \pi_2 \in f_2(a)$, such that $\pi_1 \sqsubseteq \pi_2$.

To prove that the new transfer function $\mathcal{T}_{PQ}$ is monotonic, we have to show that if $f_1 \preccurlyeq f_2$, then $\mathcal{T}_{PQ}(f_1) \preccurlyeq \mathcal{T}_{PQ}(f_2)$. Assume that the access a' happens between program points Q and P. Let $\mathcal{T}_{PQ}(f_x) = \hat{f}_x,\ x = 1, 2$. We have to show that $\forall a \in Acc_{NC}, \forall \pi_1' \in \hat{f}_1(a), \exists \pi_2' \in \hat{f}_2(a)$, such that $\pi_1' \sqsubseteq \pi_2'$.

Now, for $\pi_1' \in \hat{f}_1(a)$, π_1' would have been obtained from some $\pi_1 \in f_1$. Otherwise, $\pi_1' = \{a\}$, which is the singleton miss-path added when $a = a'$. In this case, $\{a\}$ would be present in $\hat{f}_2(a)$ as well.

The transfer function will add the new access a' and possibly end the miss-path to obtain $\pi_1' \in \hat{f}_1(a)$ from π_1. We know that there exists $\pi_2 \in f_2(a)$ such

that $\pi_1 \sqsubseteq \pi_2$. Suppose $\pi_1 = \pi_2$, then $\pi_1 \in f_2(a)$. Hence, $\pi_1' \in \hat{f}_2(a)$. For the original transfer function and join defined in Section 5, this proof will be sufficient to prove that $\mathcal{T}_{PQ}$ is monotonic.

On the other hand, suppose $|\pi_1| < T - 1$, and $\exists \pi_2 \in f_2(a)$ such that $\pi_2 \subsetneq \pi_1$ and $dist\_blocks(\pi_1) = dist\_blocks(\pi_2)$. Since $|\pi_1| < T - 1$, even if the transfer function adds the new access a' to π_1, its length will not reach the threshold T. Let $\pi_2' \in \hat{f}_2(a)$ be the miss-path obtained from π_2. Since $|\pi_2| < |\pi_1|$, adding a new access to π_2 will also not violate the threshold. Also, if $\pi_1' = \pi_1 \cup \{a'\}$, then $\pi_2' = \pi_2 \cup \{a'\}$, because both π_1 and π_2 access the same number of distinct cache blocks. Similarly, if $\pi_1' = \pi_1 \cup \{a', \dashv\}$ then $\pi_2' = \pi_2 \cup \{a', \dashv\}$. This shows that $\pi_2' \subsetneq \pi_1'$, with $dist\_blocks(\pi_1') = dist\_blocks(\pi_2')$. Hence, $\pi_1' \sqsubseteq \pi_2'$.

8 Experimental Results

We have implemented our approach for path sensitive cache analysis in the Chronos framework [14]. Chronos performs AI-based Must and May cache analysis, and classifies cache accesses as one of Hit, Miss, or NC. In addition, Chronos also provides the option of performing persistence cache analysis to further improve the classification of NC-cache accesses to Persistent (PS). We use *lp_solve* to solve the generated ILPs. Our experiments were conducted on a 4-core Intel i5 CPU with 4 GB memory.

If a cache access is classified as PS, then the accessed cache block will never get evicted during execution. This means that such accesses can experience at most one cache miss. PS classification is very useful for accesses inside loops, where the first iteration will bring the accessed block into the cache, and the block will stay in the cache for subsequent iterations. In our terminology, it would mean that the access has a cache-miss path which begins outside the loop, but has no cache-miss path entirely within the loop itself. Hence, our approach can identify persistent cache accesses, and make persistence analysis redundant.

For the experiments, we assume a 1 KB L1 instruction cache with block size 32 bytes and associativity 4. The L1 hit latency is 1 cycle, while the miss latency is 30 cycles. We use Must and May cache analysis as our baseline cache analysis. We apply our approach for all NC-accesses. We restrict the threshold value (i.e. the maximum miss-path length) to 8 (twice the cache associativity). Further, if the number of cache miss paths of an access exceeds 100, then we ignore all the miss-paths and simply classify the access as a cache miss. We experimented on 27 benchmarks from the Mälardalen WCET benchmark suite [15], and found that our approach was able to improve the WCET estimate for 9 benchmarks, with an average precision improvement of 22.54 %, compared to the WCET obtained using the baseline cache analysis.

Some of the precision improvement would be due to persistent cache blocks, and to find their contribution, we compare the WCETs obtained using Persistence analysis with our approach. Figure 4 compares the precision improvement obtained by performing persistence analysis and the improvement obtained using

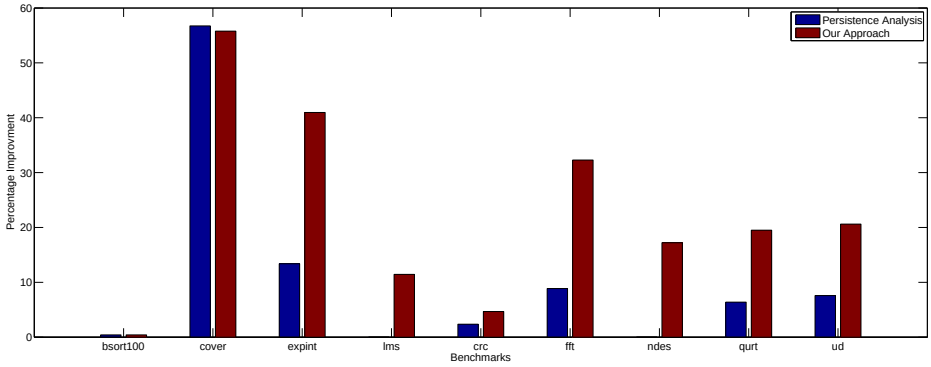

Fig. 4. Graph showing percentage improvement of WCET obtained using (1) Persistence analysis and (2) Our approach, over baseline cache analysis

our approach. It can be seen that our approach gives higher precision improvement for 8 out of the 9 benchmarks, and is very close to persistence analysis for *cover*. Our approach works better because apart for identifying persistent accesses, it also takes into account the worst-case path information while classifying accesses as cache misses. Note that this precision improvement is obtained without adding any infeasible path information.

The total time taken to determine the WCET (including the time to solve the ILP) was less than 1 second for all 27 benchmarks except *nsichneu* and *statemate*. For *statemate*, the AI analysis took 3.16 seconds, while solving the ILP required 0.6 seconds. For *nsichneu*, the AI analysis took 63.87 seconds, while solving the ILP required 3 seconds. For both these benchmarks, neither persistence analysis nor our approach showed any precision improvement. For most of the accesses in *nsichneu*, the number of cache miss paths were greater than 100, and hence these accesses were classified as cache misses. Note that *nsichneu* has a large number of program paths.

In general, there is no correlation between the effectiveness of our approach, and factors such program size, number of accesses, number of program paths, etc. However, in almost all the benchmarks programs where our approach was successful, there were accesses inside loops which had small number of cache miss paths, whose classification was refined by our approach. If an access has large number of cache miss paths, then it is highly likely that the worst-case path will contain one of them, and such accesses will not benefit from our approach. As the program size increases, the number of cache accesses will also increase, which in turn will increase size of the ILP and the time required to solve it. It is not necessary to find the cache miss paths of all accesses which are classified as NC. Accesses which are more likely to affect the WCET (for example, accesses inside loops) can be selected for miss-path based analysis, while the rest of the accesses can be simply considered as cache misses.

CEGAR-like approach: To test the effectiveness of our approach when applied only on selected cache accesses, we used a strategy similar to Counterexample guided Abstraction refinement (CEGAR) [12]. We start with IPET ILP (with no cache miss path information) and solve it to obtain the worst-case (WC) path. Then we determine the actual cache states along this path, to find the accesses which were considered as cache misses in the ILP but actually hit the cache along the WC path. Among such accesses, we pick the access with the maximum number of cache misses in the ILP (this is the counter-example), and find the cache miss paths of this access. This is equivalent to an abstraction refinement for this access, as we will now take into account its cache behavior along different paths. These miss paths are then integrated into the (current) ILP to find the new WCET (and possibly the new WC path), and the process is repeated again in the next iteration.

Since the selected cache access was actually hitting the cache along the WC path (of that iteration), no cache miss path of the access will be contained in the WC path. Hence, by integrating the cache miss path information of this access into the ILP, we would be forcing the ILP to either classify the access as a hit, or to find a new WC path which contains a miss path of the access. The new WCET is guaranteed to be less than or equal to the previous WCET. At each iteration, the size of the ILP will increase, as new cache miss path information will be added (note that the miss path information added during earlier iterations is retained). An important advantage of this approach is that the refinement process can be stopped at any time, and the WCET that was obtained after the last completed iteration can be safely used.

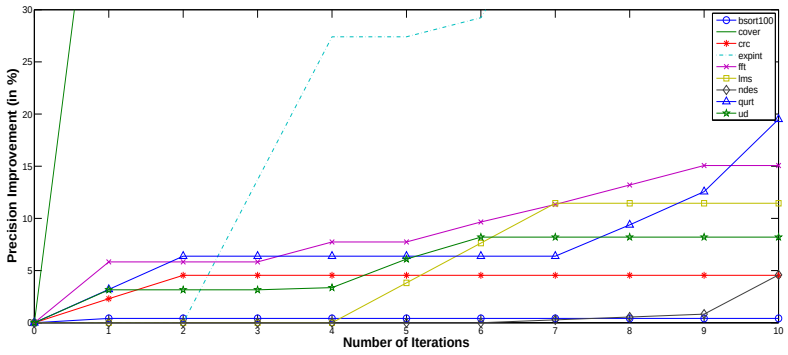

Fig. 5. Graph showing precision improvement in WCET obtained at different iterations of our CEGAR-like approach, over baseline cache analysis

Figure 5 shows the precision improvement in WCET of 9 benchmarks over the baseline cache analysis, obtained after different number of iterations of the above approach, ranging from 1 to 10. Most of the benchmarks start showing lower WCETs from the first iteration itself, with increasing precision improvement as the number of iterations increase. For 5 benchmarks, the maximum

precision improvement is achieved within 10 iterations, while the other benchmarks continue to show precision improvement after the first 10 iterations. This demonstrates that our approach is useful even when applied to limited number of cache accesses, if they are selected appropriately.

Note that we continue to use a threshold of 8 on the size of cache miss paths. The above strategy is motivated by a similar CEGAR-like strategy used for WCET estimation in [13], in which the accesses for abstraction refinement are selected in a similar manner. However, for the refinement itself, [13] uses AI-based Must and May cache analysis on the cache set containing the selected access. Hence, the information about the worst-case path is still ignored during the refinement process.

Decreasing the threshold: Restricting the length of the miss paths is another avenue for trading off precision with analysis time, since this will decrease both the time required to find the miss paths and the size of the ILP. We experimented with different thresholds for the maximum miss path length, noting the number of extra variables in the final ILP (as compared to the ILP generated by IPET), and the precision improvement of WCET. Table 2 shows the precision improvement in WCET, and the extra number of variables, for each of the 9 benchmarks of Figure 4, with different threshold values, ranging from 1 to 8. Note that in this experiment, we find and integrate miss paths of all NC-accesses into the ILP.

Table 2. Effect of different thresholds of miss path length on size of ILP and WCET

Benchmark	**Precision Improvement (%) Threshold =**				**Extra variables Threshold =**			
	1	**2**	**4**	**8**	**1**	**2**	**4**	**8**
bsort100	0.42	0.42	0.42	0.42	8	8	8	8
cover	52.41	52.41	50.49	55.78	15	15	23	12
expint	13.66	40.9	40.9	40.9	22	24	25	25
lms	0	3.82	3.82	11.45	33	34	37	26
crc	0	0	4.25	4.67	186	231	472	576
fft	2	3.2	22.48	33.3	211	230	255	259
ndes	0.7	0.7	6.1	17.3	312	316	543	574
qurt	9.7	9.7	19.38	25.28	329	328	366	500
ud	5	5	5	20.6*	323	390	972	897*

Concentrating on the precision improvement, it is interesting to see that even with low thresholds, several benchmarks show considerable precision improvement. With a threshold of 2 (half the cache associativity), all benchmarks except *crc* experience non-zero precision improvement, while for a threshold value equal to the cache associativity, all benchmarks show improvement. For all the benchmarks, the maximum precision improvement is obtained at the highest threshold value (8). The caveat with increasing the maximum miss path length is the in-

crease in the size of the ILP. For most of the benchmarks, the maximum number of extra variables are added at the maximum threshold. Note that for these benchmarks, the number of added variables is still small enough for *lp_solve* to solve it very fast.

For some benchmarks, (e.g. *cover*, *lms*) the number of variables decrease on increasing the threshold value from 4 to 8. The reason is that some of the miss paths determined with a threshold of 4 would not be actual cache miss paths, but the analysis does not recognize this due to the restriction on length. Once the allowable length is increased, the analysis will be able to determine this, and discard them, thus decreasing the number of variables. It should be noted that for the benchmark *nsichneu*, for a threshold of upto 4, all the accesses had less than 100 miss paths. The number of extra variables in the ILP for *nsichneu* with a threshold of 4 were 2832, with 52 seconds required to compute the miss paths, and 4 seconds required to solve the ILP (970 extra variables were required for thresholds of 1 and 2). In general, the above results suggest that by lowering the threshold on the length of miss-paths, the size of the ILP can be controlled. Also, even with a low threshold, it is possible to improve the precision of the WCET using our approach.

While we have not experimented with the impact of infeasible paths on cache analysis, we note that previous techniques which integrate infeasible path information into the IPET ILP ([4], [5]) can be directly applied on our modified ILP which has cache miss path information added to it. We have only concentrated on instruction caches, because although it is possible to use cache miss paths for data caches with few modifications, it may not have the same impact on improving the precision. Address analysis for data caches is highly imprecise, and may only estimate a set of cache blocks (instead of a single cache block) accessed by an instruction. Hence, while finding cache miss paths, we may quickly exceed A distinct cache blocks, which may result in short and imprecise miss paths.

9 Conclusion

In this work, we have presented a new approach to cache analysis which does not completely rely on Abstract interpretation, but instead uses AI to obtain path-sensitive information about cache accesses, in the form of cache miss paths. This information is then integrated into the IPET ILP, thus allowing us to take advantage of the worst-case path information and find the cache behavior of accesses along this path. Since our AI-based analysis is path-sensitive to a limited extent, to control the size of the ILP, we also provide user-defined thresholds and a CEGAR-like approach to trade-off analysis time with precision. Experimentally, our approach provides lower WCETs for 9 out of 27 Mälardalen benchmarks, with an average precision improvement 22.5 %, with a negligible increase in analysis time. Our approach also provides the opportunity to use already available infeasible path information for cache analysis.

10 Acknowledgements

This work was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award. We would also like to thank the anonymous reviewers for their suggestions.

References

1. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. In: Real-Time Systems 17(2-3), 131-181 (1999)
2. Li, Y.T.-S., Malik, S., Wolfe, A.: Efficient microarchitecture modeling and path analysis for real-time software. In: 16th IEEE Real-Time Systems Symposium, 1995, pp.298-307, 1995
3. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In: 27th IEEE Real-Time Systems Symposium, 2006, pp.57-66, Dec. 2006
4. Engblom, J., Ermedahl, A.: Modeling complex flows for worst-case execution time analysis. In: 21st IEEE Real-Time Systems Symposium, 2000, pp.163-174, 2000
5. Blackham, B., Liffiton, M., Heiser, G.: Trickle:automated infeasible path detection using all minimal unsatisfiable subsets. In : 20th IEEE Real-time and Embedded Technology and Applications Symposium, 2014
6. Nagar, K., Srikant, Y N.: Precise shared cache analysis using optimal interference placement. In : 20th IEEE Real-time and Embedded Technology and Applications Symposium, 2014
7. Chattopadhyay, S., Roychoudhury, A.: Scalable and Precise Refinement of Cache Timing Analysis via Model Checking. In: 32nd IEEE Real-Time Systems Symposium, 2011 , pp.193-203, 2011
8. Banerjee, A., Chattopadhyay, S., Roychoudhury, A.: Precise micro-architectural modeling for WCET analysis via AI+SAT. In: 19th IEEE Real-Time and Embedded Technology and Applications Symposium, 2013, pp.87-96, 2013
9. Li, Y.T.-S., Malik, S., Wolfe, A.: Cache modeling for real-time software: beyond direct mapped instruction caches. In: 17th IEEE Real-Time Systems Symposium, 1996, pp.254-263, 1996
10. Wilhelm, R.: Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. In: Bernhard, S., Giorgio, L. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 309-322. Springer, Berlin Heidelberg (2004)
11. Bach Khoa Huynh, Lei Ju, Roychoudhury, A.: Scope-Aware Data Cache Analysis for WCET Estimation. In : 17th IEEE Real-Time and Embedded Technology and Applications Symposium, pp.203-212, 2011
12. Clarke, R., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. In : J. ACM 50, 5 (September 2003), pp. 752-794, 2003
13. Cerny, P., Henzinger, T., Radhakrishna, A.: Quantitative abstraction refinement. In : Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), pp.115-128, 2013
14. Li, X., Liang, Y., Mitra, T., Roychoudhury, A.: Chronos: A Timing Analyzer for Embedded Software. In : Science of Computer Programming, 69 (1-3), pp. 56-67 (2007).
15. WCET Projects / Benchmarks, `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`