Refining Cache Behavior Prediction using Cache Miss Paths

Kartik Nagar, Y. N. Srikant
IISc-CSA-TR-2016-2
June 2016

Abstract

Worst Case Execution Time (WCET) is an important metric for programs running on real-time systems, and finding precise estimates of a program’s WCET is crucial to avoid wastage of hardware resources and to improve the schedulability of task sets. Caches have a major impact on a program’s execution time, and accurate estimation of a program’s cache behavior can lead to significant reduction in its estimated WCET. The traditional approach to cache analysis generally targets the worst-case cache behavior of individual cache accesses, and provides a safe hit-miss classification for every individual access. In this work, we show that these classifications are not sufficient to precisely capture cache behavior, since they apply to individual accesses, and often, more precise predictions can be made about groups of accesses. Further, memory accesses inside loops may show the worst-case behavior only for a subset of the iteration space. In order to predict such behavior in a scalable fashion, we use the fact that the cache behavior of an access mostly depends only on the memory accesses made in the immediate vicinity, and hence we analyze a small, fixed size neighborhood of every access with complete precision and summarize the resulting information in the form of cache miss paths. A variety of analyses are then performed on the cache miss paths to make precise predictions about cache behavior. We also demonstrate precision issues in Abstract Interpretation based Must and Persistence cache analysis which can be easily solved using cache miss paths. Experimental results over a wide range of benchmarks demonstrate precision improvement in WCET of multi-path programs over AI-based approaches, and we also show how to integrate our approach with other micro-architectural analysis such as pipeline analysis.
1 Introduction

For real-time systems, the Worst Case Execution Time (WCET) of a program is an important parameter, used by scheduling policies to ensure that deadlines of all tasks are met. Since a task is generally allocated hardware resources for the entire duration of its WCET irrespective of the actual time it takes for execution, overestimation of WCET can lead to wastage of computational resources. The aim of timing analysis is to statically find precise upper bounds on the WCET, and this is a challenging problem due to micro-architectural features such as caches, pipelines, branch prediction, etc. used in modern processors.

Caches provide fast access to a small portion of the main memory contents accessed by the program. The latency of a memory access depends on the cache state, which in turn depends on the sequence of memory accesses made in the past by the program. If the requested memory block is present in the cache, then the access only suffers the cache latency, which is often orders of magnitude smaller than the main memory latency. As a result, predicting memory accesses which hit the cache can lead to a substantial reduction in the estimated WCET of a program. Of course, one also has to ensure that the estimated WCET is always greater than the actual execution time across all execution instances, for all program inputs. Cache analysis is further complicated by the exponentially large number of cache states that are possible during actual execution. Ideally, we want a scalable technique for performing cache analysis which can be safely used for WCET estimation, but which does not compromise on precision.

The standard approach for performing cache analysis is to use some form of abstraction, and find abstract cache states at every program point, which cover all actual cache states possible during execution. These abstract cache states are then used to assign a static hit-miss classification to every access. The net effect is that these approaches find the worst-case cache behavior of every memory access of the program individually, and then use this information to provide a safe hit-miss classification for every individual access. They lose information about the relative behavior of different accesses, and only make a cumulative prediction for every access which is assumed to hold across all instances of the access.

However, the worst-case behavior of two memory accesses may never occur simultaneously in the same execution instance. This could happen, for example, when the accesses responsible for causing misses to the two accesses may never be executed together since they are in different branches of a conditional statement. In such cases, even though we do not know which access will cause a miss, we know that only one miss can occur among the two accesses. For memory accesses inside loops, it is possible that multiple iterations of the enclosing loop are required to realize the worst-case behavior (i.e. to cause a cache miss). This could happen when the accesses responsible for causing misses cannot be executed in a single iteration of the enclosing loop.


By detecting such cases, we can provide a more accurate bound on the maximum number of misses caused by an access (or a group of accesses) inside a loop, which may only be a fraction of the number of iterations of the loop. Cache Hit-Miss Classifications such as Always-Hit or Persistent determined using standard cache analysis are not enough to capture such cache behavior. Further, there are precision issues with the standard approaches themselves, as there exists access patterns for which they fail to identify cache accesses which should actually be classified as Always-Hit or Persistent.

In order to solve these precision issues, we use the concept of cache miss paths [1], which are abstractions of program paths along which an access suffers a cache miss. The main insight behind the applicability of cache miss paths is that the cache behavior of an access mostly depends on the memory accesses made in the immediate vicinity of the access, and hence, in most cases, it is sufficient to analyze a small, fixed size neighborhood of an access with maximum precision to obtain complete information about the cache behavior of the access. We differentiate between every path in this small neighborhood, along which the access could miss the cache, and keep track of all the relevant accesses on these paths. Various analyses can then be performed on the cache miss paths of accesses to refine their cache behavior prediction.

For example, we analyze the program control flow graph (CFG) to find simple properties about the cache miss paths of different accesses which could either prevent them from occurring in the same execution instance or in the same iteration of an enclosing loop. For basic blocks inside loops, our approach finds worst-case profiles of the form <Max_misses, Iters >, which denotes that the basic block can suffer Max_misses number of cache misses, only if it is executed once for every Iters number of iterations of its innermost enclosing loop. We also show how cache miss paths can be used to identify accesses which always hit the cache or are persistent, but are missed by the standard approaches. Finally, we also propose an ILP formulation which integrates cache miss paths into the IPET ILP to find the exact cache behavior on the Worst Case Execution Path. This formulation is more precise than the ILP proposed in [1].

We have implemented the proposed techniques, focusing on instruction cache analysis, and experimented on a wide range of benchmarks from the Mälardalen, MiBench, and StreamIt benchmark suites, and also on the real-world DEBIE-1 program. The results show an average precision improvement of 11% in the WCET over Abstract Interpretation based cache analyses for Mälardalen benchmarks, and we also demonstrate adequate precision improvement in WCET of larger benchmarks from other sources. We also show how the proposed algorithmic approach can be integrated with pipeline analysis. This allows the algorithmic approach to be useful even in architectures with timing anomalies, and also allows it to statically detect any available instruction parallelism to further improve precision of the WCET.
2 Examples

In this section, we illustrate the precision issues with AI-based cache analysis, using several examples. Note that unless otherwise mentioned, a cache refers to a first-level instruction cache, with LRU replacement policy. Must analysis is used to find cache blocks which are guaranteed to be in the cache across all execution instances, so that accesses to these cache blocks can be classified as Always-Hit (AH). However, it can actually miss such scenarios for some programs. For example, consider Figure 1 which shows a portion of program CFG and the instruction cache accesses therein. \( m_1, m_2 \) indicate cache blocks, and also the access to those cache blocks. Assume that they map to the same cache set and the cache associativity is 2. The figure also shows the abstract Must cache states at various program points (in dotted lines, with more recently accessed cache blocks towards the left). As mentioned earlier, Must cache analysis computes, at a program point, all those cache blocks which are guaranteed to be present in the actual cache at that program point. It also maintains an upper bound on the ‘age’ of a cache block, which is simply the number of cache blocks more recently accessed. For a given access, if the accessed cache block is present in the Must cache, then it is classified as Always-Hit.

Consider the Must cache states at the end of basic blocks \( v_1, v_2 \), and the result of their join at the start of \( v_3 \). The access to \( m_2 \) in \( v_2 \) increases the age of \( m_1 \), but also brings \( m_2 \) in the cache. However, since join in Must analysis at a merge point selects only those cache blocks which are present in the must caches at the end of all predecessor basic blocks and also takes their maximum age, \( m_2 \) will not be present in the Must cache after join, but its effect on the age of \( m_1 \) will be retained. As a result, the access to \( m_2 \) in \( v_3 \) will evict \( m_1 \) out of the Must cache, due to which the access to \( m_1 \) in \( v_1 \) will not be classified as Always-Hit. However, it can be clearly seen that \( m_1 \) is guaranteed to be present in the cache at the start of \( v_1 \), and Must analysis is incorrectly adding the aging effect of accesses to the same cache block (\( m_2 \))
Figure 2: Precision issue with Persistence Analysis

to another cache block ($m_1$) multiple times.

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

For instruction caches, Persistence analysis is often more effective than Must analysis, because it identifies those cache blocks which are never evicted (within a fixed scope), and cache accesses inside loops frequently have this property. Such accesses will cause at most one cache miss for every entry to the scope in which they have been classified as persistent. The safe version of Persistence analysis ([2], [3]) determines, for every cache block $m$, the maximal set of younger cache blocks that may have been accessed since the last access to $m$ (within a specific scope). If the cardinality of this set is less than the cache associativity, then $m$ is declared as persistent. Hence, for the example of Figure 1, $m_1$ would be declared as persistent. However, there are other precision issues with Persistence analysis, and it may also miss cache accesses which are actually persistent.

Consider the program CFG shown in Figure 2. Assume that cache blocks $m_1, m_2, m_3$ map to the same cache set, and the cache associativity is 2. In this example, $m_1$ and $m_2$ are not persistent, but $m_3$ is persistent. However, the set of younger cache blocks of $m_3$ would contain both $m_1$ and $m_2$, and hence Persistence analysis would not able to identify $m_3$ as persistent.

Further, classifications such as Always Hit and Persistent are not enough to capture precise cache behaviour. These classifications apply to individual accesses, but often, a more precise prediction can be made about a group of accesses. For example, consider the program CFG shown in Figure 3. Assume that cache blocks $m_1, m_3$
Figure 3: Example illustrating difference between worst-case behavior of individual accesses and a group of accesses

Figure 4: Example illustrating frequency of worst-case behavior for accesses in loops

map to cache set $s_1$, while $m_2, m_4$ map to cache set $s_2$ and the cache associativity is 1. We focus on the accesses to $m_3, m_4$ in basic block $v_3$. Note that neither of these accesses can be classified as Always Hit or Persistent. $m_1$ in $v_1$ will evict $m_3$, while $m_2$ in $v_2$ will evict $m_4$. However, both these evictions cannot happen simultaneously in the same iteration. In other words, at least one cache hit in $v_3$ is guaranteed to occur in every iteration (except possibly the first).

Accesses inside loops may not exhibit the worst-case behaviour in every iteration, but only in a subset of the iteration space. Consider the program CFG in Figure 4. Assume that $m_1$ and $m_2$ map to the same cache set, and the cache associativity is 1. Again, neither of the accesses to $m_1$ and $m_2$ can be classified as Always-Hit or Persistent.

However, for $m_1$ to miss the cache, basic block $v_2$ must have been executed in the previous iteration, and the same is true for $m_2$ and $v_1$. In other words, $m_1$ (or $m_2$) cannot cause a cache miss in every iteration of the loop, and the maximum number of misses caused by either is equal to half the total number of iterations. Every iteration can still result in one cache miss, but for that the execution must alternate between the two branches across iterations. If the execution time of one of the branches is
smaller than the other, then to cause a cache miss in all iterations, the worst-case execution path would have to pass through the branch with the smaller execution time for half of the iterations, which would lower the WCET estimate.

Finally, in all of the above examples, our aim has been to find the worst-case cache behaviour that nonetheless holds across all execution instances. However, knowledge about the worst case execution path can be used to find the exact cache behaviour which only holds in the worst-case execution instance. This is the most precise prediction for cache behaviour that can potentially be made, for WCET estimation, and there are instances where it will be more precise than any analysis which does not use WCEP information. For example, consider the program CFG shown in Figure 5. Along with the cache accesses made by basic blocks $v_1, v_2, v_3$, we also know the maximum execution time (i.e. WCET) of the basic blocks $v_1$ and $v_2$ to be 100 and 1000 cycles respectively. Then, the WCEP will pass through $v_2$. Assume that $m_1, m_1'$ map to cache set $s_1$, $m_2, m_2'$ map to cache set $s_2$ and $m_3, m_3'$ map to cache set $s_3$, and the cache associativity is 1. We focus on the basic block $v_3$. It is clear that even though the worst-case cache behaviour of $v_3$ is 2 cache misses in every iteration (accesses to $m_1$ and $m_2$), execution along the WCEP will only cause 1 cache miss (to $m_3$) in every iteration (except possibly the first).

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

In order to detect the behaviors illustrated in the previous section, we propose to use cache miss paths. Cache miss paths provide an efficient way to obtain a precise summary of the cache behavior of an access. The main insight behind the applicability of cache miss paths is that for LRU caches, if an access hits the cache, it is most likely that the same cache block has been accessed very recently in the past. Conversely, it should be possible to predict that an access misses the cache by observing only a small portion of the most recent accesses to the same cache set. If the cache associativity is $k$, then we only need $k$ accesses to other distinct cache blocks to guarantee that an access would miss the cache. Hence, given an access, we analyze a small portion of the program which can be executed just before the access in the backward direction with complete precision, keeping track of all relevant accesses across individual paths. The hope is that only a small number of paths leading to an access need to be differentiated to determine whether it will hit or miss the cache.

In this section, we give a formal definition of cache miss paths and present an AI-based approach to find cache miss paths. We also formally prove the correctness of the AI-based approach. Let $G = (V, E)$ be the control flow graph (CFG) of the program. $V$ is the set of basic blocks in the program, and the edges in $E \subseteq V \times V$ denote the control flow among them. Let $v_{start}$ be the unique start basic block. Let $\mathcal{B}$ be the set of all cache blocks accessed by the program, and $\mathcal{S}$ be the set of all cache sets. Let $Set : \mathcal{B} \rightarrow \mathcal{S}$ map every cache block to its cache set. Assume that the cache associativity is $k$.

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

A walk $\sigma$ in $G$ is a sequence of basic blocks $v_1v_2\ldots v_p$ such that $(v_i, v_{i+1}) \in E$, for all $i$, $1 \leq i \leq p - 1$ (note that repetition of basic blocks is allowed). We lift the $Acc$ function to $\sigma$ in a straightforward manner to give the total set of cache blocks mapped to a given cache set in the entire walk. We use the notation $|S|$ to mean the number of elements in the set $S$. We only concentrate on those accesses which are not classified as Always Hit by Must cache analysis.

**Definition 1.** A **concrete cache miss path** of an access to $m \in \mathcal{B}$, mapped to $s \in \mathcal{S}$, in basic block $v$ is defined as a walk $\sigma = v_1v_2\ldots v_pv$ in the CFG $G$ with the
following properties:

1. \( m \not\in \bigcup_{i=2}^{p} \text{Acc}(v_i, s) \),

2. \(|\text{Acc}_a(v_1, m) \cup \bigcup_{i=2}^{p} \text{Acc}(v_i, s) \cup \text{Acc}_b(v, m)| \geq k\) \lor (v_1 = v_{\text{start}} \land m \not\in \text{Acc}(v_{\text{start}}, s))\), and

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

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

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

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

In the abstract cache miss path of an access to \( m \), we only maintain information about those basic blocks in a concrete cache miss path which actually access the cache set of \( m \). In the example in Figure 3, \( v1v3 \) is a concrete cache miss path of the access to \( m3 \), and if \( T = 2 \), then \( \alpha^T_{v3,m3}(v1v3) = \{v1, v3\} \) is its abstract cache miss path.

We now present an AI-based approach to find the abstract cache miss paths of an access. For simplicity, we only provide a description of the approach to find all the abstract cache miss paths of a single access to \( m \) in basic block \( v \), mapped to cache set \( s \). The method can be easily extended to find the miss paths of all accesses in all basic blocks simultaneously. The abstract lattice is \( \mathcal{L} = (\mathcal{P}(\mathcal{P}(V)), \subseteq) \). Each element is a set of possible abstract cache miss paths. The analysis is carried out in the backward direction in the CFG. For every basic block \( w \), the approach maintains an IN element \( \text{IN}_w \in \mathcal{L} \) at the end of the basic block, and OUT element \( \text{OUT}_w \in \mathcal{L} \) at the beginning of the basic block. Every basic block \( w \) is also associated with a transfer function \( f_w : \mathcal{L} \to \mathcal{L} \) such that \( \text{OUT}_w = f_w(\text{IN}_w) \). Also, \( \text{IN}_w = \bigcup_{(w,u) \in E} \text{OUT}_u \).
We define the function $DB_{v,m} : \mathcal{P}(V) \to \mathbb{N}$ as $DB_{v,m}(\pi) = |\bigcup_{w \in \pi} \text{Acc}_a(w,m) \cup \text{Acc}_b(v,m)|$, to count the number of distinct cache blocks accessed in the basic blocks of $\pi$ and mapped to the cache set of $m$. We now describe the transfer function $f_w$ separately for different cases. Note that $P \in \mathcal{L}$.

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

$$f_w(P) = P$$

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

$$f_w(P) = \{\pi \in P : DB_{e,m}(\pi \setminus \{v\}) \geq k \lor |\pi| = T\}$$

$$\cup \{\pi \cup \{w\} : \pi \in P \land DB_{e,m}(\pi \setminus \{v\}) < k \land |\pi| < T\}$$

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

$$f_w(P) = \{\pi \in P : DB_{e,m}(\pi \setminus \{v\}) \geq k \lor |\pi| = T\}$$

$$\cup \{\pi \cup \{w\} : \pi \in P \land DB_{e,m}(\pi \setminus \{v\}) < k \land |\pi| < T\}$$

Case - 4 : $w = v$

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

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

$$f_{v_{\text{start}}}(P) = \{\pi \in P : DB_{e,m}(\pi \setminus \{v\}) \geq k \lor |\pi| = T\}$$

$$\cup \{\pi \cup \{v_{\text{start}}\} : \pi \in P \land DB_{e,m}(\pi \setminus \{v\}) < k \land |\pi| < T\}$$

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

Table 1 shows the $IN$ and $OUT$ values of basic blocks across different iterations of the fix-point loop, on applying the proposed approach in the example of Figure 1. Here, we perform the analysis with respect to the access to $m_1$ in basic block $v_1$,.
assuming that $T = \infty$. Note that the associativity $k = 2$. We start with all IN values being empty for every basic block. In the first iteration, only Case 4 of the transfer function applies for basic block $v_1$, resulting in an incomplete miss path $\{v_1\}$ added to $OUT_{v_1}$. Note that since the analysis is carried out in backward direction, the OUT values correspond to the beginning of a basic block, while the IN value correspond to the end of a basic block. In iteration 2, since $v_3$ is a predecessor of $v_1$, the miss path $\{v_1\}$ is propagated to $IN_{v_3}$.

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

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

**Lemma 1.** The transfer function $f_w$ is distributive for all basic blocks $w$.

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

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$IN_{v_1}$</th>
<th>$OUT_{v_1}$</th>
<th>$IN_{v_2}$</th>
<th>$OUT_{v_2}$</th>
<th>$IN_{v_3}$</th>
<th>$OUT_{v_3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\phi$</td>
<td>${v_1}$</td>
<td>$\phi$</td>
<td>$\phi$</td>
<td>$\phi$</td>
<td>$\phi$</td>
</tr>
<tr>
<td>2</td>
<td>$\phi$</td>
<td>${v_1}$</td>
<td>$\phi$</td>
<td>$\phi$</td>
<td>$\phi$</td>
<td>$\phi$</td>
</tr>
<tr>
<td>3</td>
<td>$\phi$</td>
<td>${v_1}$</td>
<td>$\phi$</td>
<td>$\phi$</td>
<td>$\phi$</td>
<td>$\phi$</td>
</tr>
<tr>
<td>4</td>
<td>${v_1,v_3}$</td>
<td>${v_1}$</td>
<td>${v_1,v_3}$</td>
<td>$\phi$</td>
<td>$\phi$</td>
<td>$\phi$</td>
</tr>
<tr>
<td>5</td>
<td>${v_1,v_3}$</td>
<td>${v_1}$</td>
<td>${v_1,v_3}$</td>
<td>${v_1}$</td>
<td>${v_1}$</td>
<td>${v_1}$</td>
</tr>
<tr>
<td>6</td>
<td>${v_1,v_3}$</td>
<td>${v_1}$</td>
<td>${v_1,v_3}$</td>
<td>${v_1}$</td>
<td>${v_1}$</td>
<td>${v_1}$</td>
</tr>
<tr>
<td>7</td>
<td>${v_1,v_3}$</td>
<td>${v_1}$</td>
<td>${v_1,v_3}$</td>
<td>${v_1}$</td>
<td>${v_1}$</td>
<td>${v_1}$</td>
</tr>
</tbody>
</table>

Table 1: AI-based Miss Path Analysis applied to the example of Figure 1
Given a concrete cache miss path \(\mathcal{P}\) of all abstract values possible at the start of \(w\), let \(w_{\text{end}}\) be the unique end basic block (i.e. \(\not\in w\), such that \((w_{\text{end}}, w) \in E\)). Given a walk \(\sigma = v_1v_2\ldots v_p\), let \(f_\sigma = f_{v_1} \circ f_{v_2} \circ \ldots \circ f_{v_p}\) be the cumulative transfer function of \(\sigma\) (in reverse direction). For a basic block \(w\), let \(\Sigma_w\) be the set of all walks in \(G\) from \(w\) to \(w_{\text{end}}\).

**Lemma 2.** For all basic blocks \(w\), \(\text{OUT}_w = \bigcup_{\sigma \in \Sigma_w} f_\sigma(\phi)\).

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

**Lemma 3.** Given a concrete cache miss path \(\sigma = v_1v_2\ldots v_p\) of access to \(m\) in \(v\), \(\alpha_{v,m}^T(\sigma) \in f_{v_1} \circ f_{v_2} \circ \ldots \circ f_{v_p}(\{v\})\).

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

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

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

**Theorem 1.** For every concrete cache miss path \(\sigma\) of access \(r\) in basic block \(v\), there exists an abstract cache miss path \(\pi \in \text{OUT}_{v_{\text{start}}}\) such that \(\pi = \alpha_{T,v,m}(\sigma)\).

**Proof.** Let \(\sigma = v_1 \ldots v_p v\). Let \(\sigma_e\) be a walk in \(G\) from \(v\) to \(w_{\text{end}}\) which does not pass through \(v\). Then \(f_{\sigma_e}(\phi) = \phi\). By Lemma 5, \(\alpha_{v,m}(\sigma) \in f_{v_1} \circ f_{v_2} \circ \ldots \circ f_{v_p}(f_v(\phi)) = f_\sigma(\phi)\). If \(v_1 = v_{\text{start}}\), then \(\sigma\sigma_e\) is a walk from \(v_{\text{start}}\) to \(w_{\text{end}}\), and hence, by Lemma 3, \(f_{\sigma\sigma_e}(\phi) \in \text{OUT}_{v_{\text{start}}}\).

If \(v_1 \neq v_{\text{start}}\), then let \(\sigma_s\) be a walk from \(v_{\text{start}}\) to \(v_1\). Now, either \(DB_{v,m}(\alpha_{T,v,m}(\sigma)) \geq k\) or \(|\alpha_{v,m}(\sigma)| = T\), and hence, for all \(w\) in \(\sigma_s\), \(f_w(\{\alpha_{v,m}(\sigma)\}) = \{\alpha_{T,v,m}(\sigma)\}\). Hence, \(\alpha_{T,v,m}(\sigma) \in f_{\sigma_s\sigma_e}(\phi)\). Again by Lemma 3, this means that \(\alpha_{T,v,m}(\sigma) \in \text{OUT}_{v_{\text{start}}}\).

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

### 4 Algorithms

We now show how cache miss paths can be used to tackle the various precision issues discussed in Section 2.

#### 4.1 The precision issue with Must analysis

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

**Theorem 2.** If an access to \(m\) in \(v\) does not have any abstract cache miss paths, then it is guaranteed to cause a cache hit.

**Proof.** By Theorem 1, if \(m\) does not have any abstract cache miss paths, then it also does not have any concrete cache miss paths. This implies that it can never cause a cache miss. □

In the example in Figure 1, the access to \(m1\) in \(v1\) does not have any cache miss paths, and hence we can conclude that the access is guaranteed to hit the cache.
4.2 The precision issue with Persistence analysis

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

**Theorem 3.** If an access to $m$ in $v$ does not have any abstract cache miss paths which are completely inside an enclosing loop $L$, then $m$ is persistent in loop $L$.

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

In general, if $m$ is persistent in loop $L$, $L'$ is the parent loop of $L$ (i.e. $L$ is immediately nested inside $L'$), and $B_L$ is the maximum execution count of $L'$, then the maximum number of cache misses caused by $m$ would be $B_L$. We use the following strategy to perform scope-aware persistence analysis using miss paths: for an access to $m$ inside $v$, if $L$ is the inner-most loop which completely contains an abstract cache miss path of $v$, and if $B_L$ is the maximum execution count of loop $L$, then $m$ can cause at most $B_L$ cache misses. In the example of Figure 2, both $m_1$ and $m_2$ have miss paths inside the loop ($\{v_2,v_3\}$ and $\{v_1,v_3\}$ resp.), but $m_3$ does not have a miss path inside the loop, and hence is persistent.

4.3 Finding maximum number of cache misses in a basic block

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

**Definition 3.** Given cache accesses $r_1$ and $r_2$ in basic block $v$, and their miss paths $\pi_1$ and $\pi_2$ respectively, we say that $\pi_1$ and $\pi_2$ do not conflict with each other if there exists a walk $\sigma = v_1 \ldots v_p v$ in $G$ such that $\forall i, v \neq v_i$ and $\pi_1 \cup \pi_2 \subseteq \{v_1, \ldots, v_p\}$. If such a walk does not exist, then we say that $\pi_1$ and $\pi_2$ conflict with each other.

14
If two miss paths conflict, then they cannot cause cache misses together. In the example in Figure 3, miss path \( \pi_1(=\{v1\}) \) of \( m_3 \) and \( \pi_2(=\{v2\}) \) of \( m_4 \) conflict with each other, and hence cannot cause cache misses together. The following results provide the necessary and sufficient conditions required to automatically find such miss paths.

**Lemma 4.** Given a set of basic blocks \( W = \{v_1, \ldots, v_n\} \) and basic block \( v (v \not\in W) \), if \( \forall v_i, v_j \in W \), there exists a walk in \( G \) either from \( v_i \) to \( v_j \) or \( v_j \) to \( v_i \) which does not pass through \( v \), then there exists a walk in \( G \) which contains all the basic blocks in \( W \) and also does not pass through \( v \).

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

**Lemma 5.** Miss paths \( \pi_1 \) and \( \pi_2 \) of two accesses in \( v \) do not conflict \( \iff \forall w_1 \in \pi_1, \forall w_2 \in \pi_2 \), there exists a walk in \( G \) either from \( w_1 \) to \( w_2 \) or from \( w_2 \) to \( w_1 \) which does not pass through \( v \).

**Proof.** The forward direction is trivial, since we can take the required sub-walk from the walk \( \sigma \) which contains all basic blocks of \( \pi_1 \) and \( \pi_2 \). For the reverse direction, we simply take \( W = \pi_1 \cup \pi_2 \), and apply Lemma 4, which implies that there is walk in \( G \) which contains all the basic blocks of \( \pi_1 \) and \( \pi_2 \) and does not pass through \( v \). Note that by definition of miss paths, there always exists a walk between two basic blocks of the same miss path which does not pass through \( v \), and there is walk in \( G \) from every basic block in the miss path to \( v \). This shows that \( \pi_1 \) and \( \pi_2 \) do not conflict with each other. \( \Box \)

Hence, to find whether two miss paths conflict with each other, we need to determine whether there is a walk between every pair of basic blocks from the miss paths which does not pass through \( v \). A simple way to do this is to formulate a Data-flow Analysis (DFA) [4]. For basic block \( v \), the DFA \( D_v \) determines, for all other basic blocks \( w \) in the program, the set of basic blocks \( IN_w \), such that there exists a walk in \( G \) from every basic block in \( IN_w \) to \( w \) which does not pass through \( v \).
Table 2: Data flow analysis $D_v$ to determine conflict information for basic block $v$.

<table>
<thead>
<tr>
<th>Basic block $v$</th>
<th>GEN Set</th>
<th>KILL Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>For every other basic block $w$</td>
<td>$\phi$</td>
<td>${w}$</td>
</tr>
</tbody>
</table>

For $D_v$, the data-flow domain is $D = V$, the set of all basic blocks in the program. The GEN and KILL sets for all basic blocks are given in Table 2. The DFA fixpoint algorithm calculates the $IN_w$ and $OUT_w$ sets for all basic blocks $w$, which obey the following equations:

\[
IN_w = \bigcup_{u : (u,w) \in E} OUT_u \quad \text{and} \quad OUT_w = GEN_w \cup (IN_w \setminus KILL_w).
\]

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

**Lemma 6.** Given miss paths $\pi_1$ and $\pi_2$ of two accesses in $v$, $\pi_1$ and $\pi_2$ do not conflict if and only if $\forall w_1 \forall w_2 \in \pi_1 \cup \pi_2, (w_1 \in IN_{w_2} \lor w_2 \in IN_{w_1})$.

**Proof.** By Lemma 5 and the correctness of the DFA $D_v$. \(\square\)

**Lemma 7.** Given miss paths $\pi_1, \ldots, \pi_n$, of accesses in $v$, there exists a walk in $G$ which contains all the miss paths and contains $v$ at the end if and only if there is no pairwise conflict in the set $\{\pi_1, \ldots, \pi_n\}$.

**Proof.** The forward direction is trivial, because if there exists a walk which contains every basic block of all miss paths, then it will contain a walk between every pair of basic blocks which does not pass through $v$, and hence none of the miss paths will conflict with each other. For the reverse direction, we take $W = \bigcup_{i=1}^{n} \pi_i$. Since there is no pairwise conflict between the miss paths, by Lemma 5, there exists a walk between $v_i$ and $v_j$ which does not pass through $v$, $\forall v_i, v_j \in W$. By Lemma 4, this means that there exists a walk in $G$ which contains all the basic blocks of $W$ and does not pass through $v$. \(\square\)

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

**Theorem 4.** Given the MPCG $G_M$ of basic block $v$, the size of the maximum clique in $G_M$ is an upper bound on the maximum number of cache misses that can occur in $v$. 16
Proof. Suppose \( \{r_1, \ldots, r_m\} \) is a set of accesses in \( v \) that can become misses together. Then, there exist concrete cache miss path \( \sigma_i \) for each \( r_i \) such that a walk in \( G \) contains all the concrete miss paths \( \sigma_i \). By Theorem 1, for every \( \sigma_i \), there exists an abstract cache miss path \( \pi_i = a^{T_{v,r_i}}(\sigma_i) \). This implies that there exists a walk in \( G \) which contains all the basic blocks of \( \pi_i \) (for all \( i, 1 \leq i \leq m \)). By Lemma 7, this means that there is no pairwise conflict in the set \( \{\pi_1, \ldots, \pi_m\} \), and hence these abstract cache miss paths will form a clique in the MPCG \( G_M \) of \( v \).

Algorithm 1: Algorithm to find the maximum number of cache misses in every basic block of the program

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

Algorithm 1 summarizes the various steps discussed so far. We concentrate on the NC (non-classified) accesses, perform the data flow analysis for finding conflict information, and then create the MPCG. Finally, we find the size of the maximum clique in the MPCG, which gives us the maximum number of misses.

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

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

**Lemma 8.** Given access \( r \) in basic block \( v \) which accesses \( m \), if \( v \) is the only basic block in the program which accesses \( m \), and if for all concrete cache miss paths \( \sigma \) of \( r \), \( |\alpha_{v,m}(\sigma)| \leq T \), then execution of any abstract cache miss path of \( r \) is guaranteed to cause a cache miss.

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

The implication of the above result (along with Lemma 7) is that if \( T \) is selected to be sufficiently large, and \( \text{Misses}_v \) is the size of the maximum clique in the MPCG of \( v \), then there exists a walk in \( G \) which contains all the miss paths in the clique, and hence will cause \( \text{Misses}_v \) number of cache misses in \( v \). In instruction caches, because of small cache block sizes and comparatively larger basic block sizes, there are many cases where a cache block is accessed solely inside a single basic block, and the above lemma guarantees the most precise analysis for the accesses to such cache blocks.

### 4.4 Finding worst-case profiles of basic blocks

For basic block inside loops, the maximum number of misses calculated using Algorithm 1 may not be possible for every iteration. In the example of Figure 4, the maximum number of cache misses in both \( v_1 \) and \( v_2 \) is 1, but they cannot cause 1 cache miss in every iteration. Hence, instead of just finding the maximum number of cache misses possible in a basic block, we would instead like to find all possible worst-case profiles of a basic block, in the form \( \left< \text{Max misses, iters} > \right. \), which says that \( \text{Max misses} \) number of cache misses can happen in the basic block, only if it is executed once for every \( \text{iters} \) number of iterations. In the example of Figure 4, the worst-case profiles of both \( v_1 \) and \( v_2 \) are \( < 1, 2 > \).

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

In the following, we only consider those miss paths which are completely present in the innermost loop containing the basic block (although the approach can be applied at all nesting levels). As a result, the following results do not apply to the cache behaviour of a basic block when it is executed for the first time (for every entry to the loop containing the basic block from outside), but instead are applicable to the cache behaviour for the rest of the iterations. Consider basic block \(v\), and let \(L\) be the innermost loop containing \(v\). Let \((V_M, E_M)\) be the MPCG of \(v\), and let \(V_M^L \subseteq V_M\) be those miss paths whose basic blocks are all present either inside loop \(L\) or in one of its inner loops. Let \(E_M^L \subseteq E_M\) be the edges incident on \(V_M^L\). Let \(v_h\) be the unique entry basic block of loop \(L\). Due to the assumption that the CFG \(G\) is reducible, every path from outside the loop \(L\) must enter \(L\) through \(v_h\), and all the back edges of \(L\) must also be incident on \(v_h\). In other words, every iteration of \(L\) must begin with \(v_h\).

We use the notation \(v_1 \leadsto_w v_2\) to mean that there exists a walk in \(G\) from \(v_1\) to \(v_2\) which does not pass through \(w\). Consider a set of miss paths \(\{\pi_1, \ldots, \pi_k\} \subseteq V_M^L\) of accesses in \(v\). By Lemma 7, we know that if \(\forall v_1, v_2 \in \cup_{i=1}^k \pi_i\), either \(v_1 \leadsto_v v_2\) or \(v_2 \leadsto_v v_1\), then all the miss paths \(\pi_i\) can occur together on a walk ending in \(v\), and hence can cause \(k\) misses in \(v\). The following two lemmas give the necessary and sufficient condition for a set of miss paths to occur in the same iteration.

**Lemma 9.** Miss paths \(\pi_1\) of access \(r_1\), \(\pi_2\) of access \(r_2\), \ldots, \(\pi_k\) of access \(r_k\) in \(v\) can cause \(k\) misses in \(v\) in consecutive iterations of \(L\) \iff there exists a walk from \(v\) to \(v\) which contains exactly one instance of \(v_h\) and contains all the miss paths.

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

**Lemma 10.** Given miss paths \(\pi_1\) of access \(r_1\), \ldots, \(\pi_k\) of access \(r_k\) in \(v\), there exists a walk from \(v\) to \(v\) containing all the miss paths and exactly one instance of \(v_h\) \iff \(\forall v_1, v_2 \in \cup_{i=1}^k \pi_i\), \(v_1 \leadsto_v v_2 \lor v_2 \leadsto_v v_1\) and \(\forall v_1, v_2 \in \cup_{i=1}^k \pi_i \cup \{v\}\), \(v_1 \leadsto_{v_h} v_2 \lor v_1 \leadsto_{v_h} v_2\).

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

For the reverse direction, we redefine \( W_{\rightarrow} \) and \( W_{\leftarrow} \) as follows: \( W_{\rightarrow} = \{ w \in W | v \sim_{v_h} w \} \) and \( W_{\leftarrow} = \{ w \in W | w \sim_{v_h} v \} \). Now, \( \forall v_1, v_2 \in W_{\rightarrow} \), \( v_1 \sim_{v_h} v_2 \) or \( v_2 \sim_{v_h} v_1 \). Assume that \( v_1 \sim_{v_h} v_2 \). This walk will also not pass through \( v \), because otherwise \( v_2 \sim_{v_h} v \), and this would imply a walk between two instances of \( v \) which does not contain \( v_h \), which is a contradiction because \( v_h \) is the entry block of the innermost loop containing \( v \). Now, since \( \forall v_1, v_2 \in W_{\rightarrow} \), \( v_1 \sim_{v_h} v_2 \) or \( v_2 \sim_{v_h} v_1 \), by Lemma 4, there exists a walk \( \sigma_{\rightarrow} \) which contains all basic blocks in \( W_{\rightarrow} \) and does not pass through \( v_h \). This walk will also not pass through \( v \). Similarly, there exists a walk \( \sigma_{\leftarrow} \) which contains all basic blocks in \( W_{\leftarrow} \) and does not pass through \( v_h \) and \( v \). Now, the walk from \( v \) to the first basic block in \( \sigma_{\rightarrow} \), followed by \( \sigma_{\leftarrow} \), followed by the walk from the last basic block in \( \sigma_{\rightarrow} \) to \( v_h \), followed by the walk from \( v_h \) to the first basic block in \( \sigma_{\leftarrow} \), followed by the walk \( \sigma_{\rightarrow} \) is the required walk between two instances of \( v \) which does contain all basic blocks in \( W \) and does not contain \( v_h \).

The implication is that if every walk between two basic blocks in a loop must pass through the entry basic block \( v_h \), (for example, between \( v_1 \) and \( v_2 \) in Figure 4), then such a walk must skip the basic block under analysis \( v \) for at least one iteration. Hence miss paths containing such basic blocks cannot cause cache misses in consecutive iterations. The next lemma gives the minimum number of iterations required to execute such miss paths.

**Lemma 11.** Given basic blocks \( w_1, \ldots, w_k \) in loop \( L \) (or one of its inner loops), every walk containing these basic blocks contains at least \( k - 1 \) instances of \( v_h \) \( \iff \forall w_i, w_j, 1 \leq i < j \leq k \), neither \( w_i \sim_{v_h} w_j \) nor \( w_j \sim_{v_h} w_i \).

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

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

**Theorem 5.** Given miss paths \( \pi_1 \) of access \( r_1, \ldots, \pi_k \) of access \( r_k \) in \( v \), where \( \pi_1, \ldots, \pi_k \in V^L \), if there exists \( W_C \subseteq \bigcup_{i=1}^k \pi_i \cup \{ v \} \) such that \( \forall w, w' \in W_C \) neither \( w \sim_{v_h} w' \) nor \( w' \sim_{v_h} w \) then a walk from \( v \) to \( v \) containing all the basic blocks in \( W_C \), with \( v \) only coming at the endpoints, requires at least \( |W_C| \) instances of \( v_h \).

**Proof.** Let \( n = |W_C| \). First, consider the case where \( v \notin W_C \). Since \( v \notin W_C \), we know that \( \forall w \in W_C \), either \( v \sim_{v_h} w \) or \( w \sim_{v_h} v \). However, if \( \exists w, w' \in W_C \) such that \( v \sim_{v_h} w \) and \( w' \sim_{v_h} v \), then this would imply \( w' \sim_{v_h} w \) which is a contradiction. Hence, either there is a walk from \( v \) to all \( w \) in \( W_C \), or there is a walk from all \( w \) in \( W_C \) to \( v \), which does not contain \( v_h \). Suppose all walks are only from \( v \) to all basic blocks in \( W_C \). Now, by Lemma 11, a walk containing all \( n \) basic blocks in \( W_C \) requires at least \( n - 1 \) instances of \( v_h \). If \( w_1 \) is the start basic block, and \( w_n \) is the end basic block of this walk, then a walk from \( w_n \) to \( v \) will require another instance of \( v_h \). Hence, a walk from \( v \) to \( v \) containing all \( n \) basic blocks will require \( n \) instances of \( v_h \). The case when there is a walk from all \( w \) in \( W_C \) to \( v \) without passing through \( v_h \) can be proved in a similar manner.

If \( v \in W_C \), then by Lemma 11, a walk containing all basic blocks in \( W_C \) will require \( n - 1 \) instances of \( v_h \). If such a walk starts with \( v \) and ends with some basic block \( w \in W_C \), then since there is no walk from \( w \) to \( v \) which does not pass through \( v_h \), for such a walk to end \( v \) will require one more instance of \( v_h \). Similarly, if such a walk ends with \( v \) but starts with some basic block \( w \in W_C \), then a walk from \( v \) to \( w \) will require another instance of \( v_h \).

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

We keep finding maximum cliques (of possibly decreasing sizes) in the MPCG until we find a clique which requires only 1 iteration (lines 3-9). We store the size and the number of iterations required by all cliques discovered in this process (line 8). Note that we only keep one WC profile for a given number of iterations \( iters \), the one with the maximum number of misses. The function \( I_{iters}(S) \) finds the number of iterations required by clique \( S \) (lines 11-18). It does so by creating the Basic Block Conflict Graph (BBCG), whose vertices \( V_B \) represent basic blocks in the miss paths in \( S \) and \( v \) (line 12). We add an edge between two basic blocks if there does not
**ALGORITHM 2:** Algorithm to find all worst-case profiles of a basic block $v$ inside a loop with entry block $v_h$

1: Max_misses ← Size of maximum clique in MPCG $(V_M^L, E_M^L)$ of $v$
2: Perform Data flow conflict analysis $D_{v_h}$
3: repeat
4: while clique of size Max_misses does not exist in the MPCG do
5: Max_misses ← Max_misses - 1
6: end while
7: $S$ ← Clique of size Max_misses in the MPCG
8: Add $\langle$Max_misses, iters($S$) $\rangle$ to the worst-case profiles of $v$
9: until iters($S$) $= 1$
10: function iters ($S$)
11: $V_B$ ← Set of all basic blocks in miss paths of $S$ and $v$
12: for all $w_1, w_2 \in V_B$ do
13: if $w_1 \notin IN_{w_2} \land w_2 \notin IN_{w_1}$ then
14: Add $(w_1, w_2)$ to $E_B$
15: end if
16: end for
17: return Size of maximum clique in $(V_B, E_B)$
exist a walk between them which does not pass through \(v_h\) (lines 13-17). A clique \(S_B\) in the BBCG means (by Theorem 5) that a walk which starts and ends at \(v\) and passes through these basic blocks in \(G\) will require at least \(|S_B|\) iterations. Hence, the maximum clique in the BBCG would correspond to the minimum number of iterations required by miss paths in clique \(S\) (line 18). If the size of the maximum clique in the BBCG is 1, then this means that there is a walk between every pair of basic blocks in the miss paths which does not pass through \(v_h\), and by Lemmas 9 and 10, this guarantees that the miss paths can cause misses in \(v\) in consecutive iterations.

The complexity of Algorithm 2 remains exponential in the number of cache accesses inside a single basic block, since the maximum number of cliques in the MPCG is also exponential in its size, and in the worst case, the function \(\text{Iters}\) can be called for each clique. The complexity of the \(\text{Iters}\) function is exponential in the size of the BBCG, but since the maximum length of a miss path as well as the number of miss paths are constants, the size of the BBCG will be linear in the number of accesses within a single basic block.

We now present a modified version of the IPET ILP [5] which allows us to use the different worst-case profiles of a basic block while finding the worst-case path. For basic block \(v\), let \(c_v\) be the maximum possible execution count. For basic blocks inside loops, the maximum execution count can be obtained by multiplying the loop bounds of all its enclosing loops. For every worst-case profile \(p = <\text{Max misses}_p, \text{iters}_p>\) of basic block \(v\), let \(e_{v,p}\) be the WCET of \(v\) assuming \(\text{Max misses}_p\) number of cache misses in \(v\), and let \(y_{v,p}\) be the integer variable storing the execution count of this worst-case profile on the worst-case path. Finally, let \(y_v\) be the integer variable storing the total execution count of \(v\) on the worst case path. The modified ILP formulation is presented below:

Maximize

\[
\sum_{v \in V} \sum_{\text{WC profile } p \text{ of } v} e_{v,p} y_{v,p}
\]

Subject to

\[
\forall v \in V, \sum_{(w,v) \in E} z_{w,v} = y_v = \sum_{(v,u) \in E} z_{v,u}
\]

\[
\forall v \in V, \begin{cases}
  y_v = \sum_{\text{profile } p \text{ of } v} y_{v,p} \\
  \sum_{\text{profile } p \text{ of } v} \text{iters}_p y_{v,p} \leq c_v
\end{cases}
\]

In the objective function to be maximized, we add execution times of all profiles of basic blocks. Note that basic block outside loops will only have one worst-case profile,
with the maximum number of cache misses obtained using Algorithm 1. Equation 2 ensures proper control flow across basic blocks, through variables $z_{u,v}$, which store the number of times execution passes through the edge $(u, v) \in E$. The sum of execution counts for different profiles of a basic block will be equal to the total execution count the basic block on the WC path (Equation 3). Every occurrence of the worst case profile $p$ will consume $\text{iters}_p$ number of iterations, and the maximum number of iterations is upper bounded by $c_v$. This establishes an upper bound on $y_{u,p}$, which is the maximum number of times $v$ can cause $Max\_misses_p$ misses (Equation 4). While the WC path obtained using the original IPET formulation follows a single path in all iterations of a loop, the above ILP formulation allows the possibility of following different paths across iterations, and uses the appropriate WCET for basic blocks in such cases. Finally, we note that in Algorithm 2, it is not necessary to continue searching for cliques until we find a clique which requires only 1 iteration. We can stop the loop at any point and simply assume that the last clique found requires 1 iteration. This allows a trade-off between analysis precision and efficiency.

5 ILP-based Approach

While the algorithms in the previous section can tackle the precision issues illustrated in Figures 1-4, in order to solve the precision issue of Figure 5, we need information about the WCEP. However, as explained earlier, the WCEP is closely tied with the predicted cache behaviour, and hence one cannot directly use the WCEP obtained using the IPET formulation. In this section, we show how to integrate cache-miss paths into the IPET formulation, so that an access suffers a cache miss only if the WCEP contains a miss path of the access. As an added advantage, this approach automatically solves the various issues illustrated in Figures 1-4, so that the algorithms of the previous sections are not needed.

We build our ILP formulation on top of the IPET ILP, and introduce new integer variables for every access Not Classified (NC) by AI-based cache analysis, as well as for each cache-miss path of these accesses. The number of cache misses suffered along a cache-miss path will be constrained by the execution counts of the basic blocks in the miss path.

Since we only focus on NC accesses, only the first access to cache block $m$ in basic block $v$ needs to be considered. Let $Acc_f : V \rightarrow B$, $Acc_f(v)$ gives the set of cache blocks accessed in $v$, such that the first access to these cache blocks in $v$ is non-classified. For each basic block $v$, let $MP_v : Acc_f(v) \rightarrow \mathcal{P}(\mathcal{P}(V))$ give the set of abstract cache miss paths of NC-accesses in $v$. These abstract miss paths are obtained using the AI-based approach of Section 3. For $m \in Acc_f(v)$, let $BB_{v,m} = \bigcup_{\pi \in MP_v(m)} \pi$ be the set of basic blocks in cache miss paths of $m$. Let $e_v$ be the estimated WCET of basic block $v$ obtained by assuming that all NC-accesses hit the cache.

24
For each basic block \( v \), \( y_v \) stores the execution count of \( v \) on the worst-case execution path. For an edge between basic blocks \( v \) and \( w \) in the CFG, the variable \( z_{v,w} \) stores the number of times execution passes from \( v \) to \( w \) on the WCEP. The objective is to find the WCEP, i.e. the execution counts of basic blocks which maximizes the execution time of the program. Following is our proposed ILP:

\[
\text{Maximize } \sum_{v \in V} (e_v y_v + \sum_{m \in \text{Acc}_v(v)} \text{CMP } x_{v,m}) \quad (5)
\]

subject to

\[
\forall v \in V, \quad y_v = \sum_{w \in \text{pred}(v)} z_{w,v} = \sum_{w' \in \text{succ}(v)} z_{v,w'} \quad (6)
\]

\[
\forall v \in V, \forall m \in \text{Acc}_f(v), \quad \begin{cases} 
  x_{v,m} \leq \sum_{\pi \in \text{MP}_v(m)} x_{v,m}^\pi \\
  \forall w \in \text{BB}_{v,m}, \quad \sum_{\pi \in \text{MP}_v(m), w \in \pi} x_{v,m}^\pi \leq y_w \quad (8)
\end{cases}
\]

The product \( e_v y_v \) is the contribution of \( v \) to the execution time of the program, assuming that all NC-instructions are cache hits. The variable \( x_{v,m} \) accounts for the cache misses suffered by access to \( m \) in \( v \). Each cache miss causes an additional execution time of \( \text{CMP} \). Hence, the objective function is the sum of the total execution times of all basic blocks on the WCEP (Equation 5). Equation 6 ensures proper control flow across basic blocks, through variables \( z_{u,v} \), which store the number of times execution passes through the edge \((u, v) \in E\).

For each miss path \( \pi \) of the access to \( m \) in \( v \), the variable \( x_{v,m}^\pi \) counts the number of misses suffered by the access along \( \pi \). If an access has multiple cache miss paths, then any of those miss paths could be present on the WCEP. Moreover, for an access inside a loop, multiple cache-miss paths may actually be present on the WCEP (for example, different miss paths could be activated in different iterations). Hence, the total number of misses suffered by an access to \( m \) in \( v \) \((x_{v,m})\) is bounded by the sum of its \( x_{v,m}^\pi \) variables (Equation 7). A miss path is present on the WCEP if the execution counts of all basic blocks on the miss path are non-zero. Further, the maximum number of times that a miss path is executed would be equal to the minimum among the execution count of the basic blocks on the miss path. Finally, if a basic block is present on multiple miss paths of the same access, then a single execution of the basic block can activate at most one miss path during actual execution, and hence cause at most one miss. The set of constraints represented by Equation 8 encode all the above requirements.
In addition to the above constraints, loop constraints, which will bound the execution count of loop headers and infeasible path constraints can also be added. An infeasible path generally takes the form of a set of basic blocks, which will never be executed together (due to their execution being guarded by conditionals whose conjunction is not satisfiable). The constraints will place an upper bound on the sum of the execution counts \((y_v)\) of such basic blocks. By appending them to the above ILP, we can not only guarantee that the worst case path will not contain the infeasible path, but also that no cache miss path will contain such basic blocks, and thus the cache misses caused due to infeasible paths will be ignored.

The above formulation is more precise than the ILP formulation proposed in [1]. The major difference between the two formulations is in Equation 8, where in [1], a separate equation is used for each variable \(x^{\pi}_{v,m}\) specifying an upper bound of \(y_w\) for all \(w \in \pi\). On the other hand, Equation 8 specifies an upper bound of \(y_w\) for the sum of number of misses caused along all miss paths containing \(w\). As a result, if a misspath \(\pi_1\) is completely contained inside another miss path \(\pi_2\), and if the WCEP contains \(\pi_2\) then the formulation in [1] will incorrectly count the same miss twice, in the variables \(x^{\pi_1}_{v,m}\) and \(x^{\pi_2}_{v,m}\). On the other hand, the above formulation will always provide an upper bound on \(x^{\pi_1}_{v,m} + x^{\pi_2}_{v,m}\).

### 6 Experimental Evaluation

#### 6.1 Setup

We have implemented the proposed techniques on top of the Chronos WCET analyzer [6]. Chronos takes as input the binary of the program whose WCET is to be determined (compiled for SimpleScalar ISA), along with annotations describing loop bounds and infeasible path information. In this work, we focus exclusively on instruction cache analysis, and assume a perfect data cache. Further, we assume a single level hierarchy, although the proposed techniques can also be applied for analysis of the L1 cache in a multi-level cache hierarchy.

For cache analysis, Chronos uses Abstract Interpretation based Must and May cache analysis [7] and the safe version of persistence analysis [2] to provide a hit-miss classification to every memory access. For pipeline analysis, Chronos builds a separate execution graph [8] for every basic block, modeling all the pipeline stages of every instruction in the basic block, their execution time estimates, and their interdependencies. Note that this requires a safe hit-miss classification for every cache access. These execution graphs are then used to determine the WCET estimate of every basic block.

We implement the proposed techniques in the following manner. We work on the virtually function-inlined CFG of the program reconstructed by Chronos, and perform
AI-based Must, May and Persistence analysis to obtain safe hit-miss classifications for every memory access. We then focus on the accesses which remain Not Classified (NC) by AI-based analysis, and find the cache miss paths of such accesses using the proposed AI-based technique (Section 3).

We then separately implement the ILP-based and the algorithmic approaches. The ILP-based approach (Section 5) requires as input the maximum execution time of every basic block, assuming all NC-accesses hit the cache (the constant $e_v$ for basic block $v$). It also assumes that every cache miss suffered by an NC access will add a constant cache miss penalty ($CMP$) to the final execution time. This does not necessarily hold true for architectures with pipelines, since accesses to main memory will frequently occur in parallel with instructions running in other stages, so that some portion of the cache miss penalty may not be visible in the final execution time. Hence, assuming a constant cache miss penalty for every cache miss may over-estimate the WCET by ignoring the effect of instruction parallelism.

To make matters worse, some architectures may exhibit strong impact timing anomalies [9], which happen when a local increase in the execution time of an instruction results in a greater global increase in the total execution time of the program. In our case, the increase in the execution time of the program due to an instruction cache miss may become greater than the main memory latency. [9] lists the two main factors which may cause strong impact timing anomalies: (1) an out-of-order functional unit which allows instructions to be dispatched in an order different from the program order or (2) multiple, in-order, non-uniform functional units. An example of the latter is an architecture with separate functional units for floating point and integer computations. For architectures with strong impact timing anomalies, the ILP-based approach must assume in the worst-case that every cache miss can potentially cause a strong impact timing anomaly, and the cache miss penalty ($CMP$) must take into the account the maximum possible increase in execution time due to a strong impact timing anomaly. This would further hamper the precision of the ILP-based approach.

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

We now describe our implementation of the algorithmic approach (Section 4). We
first find those accesses which do not have any cache miss paths, and change their classification to AH (Section 4.1). We then use the method described in Section 4.2 to find persistent accesses inside loops, which may have been missed due to the precision issue in AI-based persistence analysis. Finally we use Algorithms 1 and 2 to find the worst-case profiles of all basic blocks, which gives the maximum number of cache misses in a basic block, and if the basic block is inside a loop, then the minimum number of iterations required to cause them.

For every worst-case profile $p$ of basic block $v$, we perform pipeline analysis in the following manner: if the maximum number of misses possible in basic block $v$ is $k$ ($= \text{Max misses}_p$), and the number of NC-accesses in $n$, then we create $(\binom{n}{k})$ versions of $v$, corresponding to every choice of $n-k$ accesses. For a given version corresponding to a particular choice of $n-k$ accesses, we change the hit-miss classification of these $n-k$ accesses from NC to AH, then perform the pipeline analysis of $v$ to obtain an execution time estimate. We do this for all versions, and then take the maximum of the execution time estimates, which will be used for the constant $e_{v,p}$ in the modified IPET ILP formulation described towards the end of Section 4.4. Note that in every version corresponding to profile $p$ of basic block $v$, the maximum number of misses would be $n-(n-k) = k$, which establishes the safety of our approach. By performing a separate pipeline analysis for every version, we can also statically find instances when the cache miss latency of accesses which do miss the cache could be hidden by instructions executing in other stages, thus reducing the overall WCET estimate. We also note that this allows the algorithmic approach to be used for architectures with timing anomalies, which will be detected by pipeline analysis.

Since pipeline analysis requires a hit-miss classification for every access, but the ILP-based approach defers finding this classification to the final stage of WCET estimation process, it is not possible to integrate the ILP-based approach with pipeline analysis, and certain assumptions (lack of strong impact timing anomalies, constant penalty for every cache miss) need to be made. The algorithmic approach also does not provide a hit-miss classification for every individual access, but instead provides upper bounds on the number of cache misses for every basic block. However, this information can still be translated to multiple safe hit-miss classifications for individual accesses, and hence the algorithmic approach can be tightly integrated with pipeline analysis.

We used lp_solve to solve the generated ILPs, and the cliquer\textsuperscript{1} library for finding maximal cliques in graphs. Our experiments were conducted on a 4-core Intel i5 CPU with 4 GB memory. We experimented on benchmarks chosen from Mälardalen, MiBench, StreamIt and DEBIE suites (obtained from the TACLeBench collection\textsuperscript{2}). For each benchmark, we use a different L1 instruction cache, with cache size approximately equal to 10% of the code size. The L1 cache hit latency is 1 cycle.

\textsuperscript{1}http://users.aalto.fi/pat/cliquer.html
\textsuperscript{2}http://www.tacle.eu/index.php/activities/taclebench
while the main memory latency is 30 cycles. While determining the cache miss paths of accesses, the maximum miss path length ($T$) is restricted to 16, and the maximum number of miss paths per access ($N$) is restricted to 100. If an access has too many miss paths, then it is highly likely that one of the miss paths will be on almost every program path. A large number of miss paths also reduces the likelihood of finding conflicting accesses which may never occur together. We experimented with different bounds and found that a larger bound on the number of miss paths does not have any impact on the precision of the final WCET.

6.2 Mälardalen Benchmarks

We compare the WCETs obtained using the proposed ILP-based and algorithmic approaches, with the WCET obtained using AI-based approach. Figure 6 shows the precision improvement (in %) of the WCET computed using ILP-based and algorithmic approaches, as compared to the WCET computed using AI-based approach, for benchmarks from the Mälardalen suite. The precision improvement is computed as $\frac{WCET_{AI} - WCET_x}{WCET_{AI}} \times 100$, where $x \in \{\text{Algorithmic, ILP}\}$. Note that we experimented on all benchmarks from the Mälardalen suite, and the figure shows the results only for those benchmarks for which the proposed approaches showed non-zero precision improvement.

The average precision improvement is 11.3 % for the ILP-based approach, and 10.2 % for the algorithmic approach. The results demonstrate that even though the algorithmic approach does not use the global worst-case execution path information, it still matches the precision improvement of the ILP-based approach for most benchmarks. Hence, the precision issues tackled by the algorithmic approach have a more
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No. of accesses/BBs which benefit from approach of Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.1</td>
</tr>
<tr>
<td>adpcm</td>
<td>0</td>
</tr>
<tr>
<td>binarysearch</td>
<td>0</td>
</tr>
<tr>
<td>expint</td>
<td>1</td>
</tr>
<tr>
<td>compress</td>
<td>5</td>
</tr>
<tr>
<td>crc</td>
<td>0</td>
</tr>
<tr>
<td>ndes</td>
<td>2</td>
</tr>
<tr>
<td>qurt</td>
<td>6</td>
</tr>
<tr>
<td>ud</td>
<td>4</td>
</tr>
<tr>
<td>countnegative</td>
<td>0</td>
</tr>
<tr>
<td>lms</td>
<td>2</td>
</tr>
<tr>
<td>qsort-exam</td>
<td>0</td>
</tr>
<tr>
<td>select</td>
<td>0</td>
</tr>
<tr>
<td>sqrt</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3: Break-down of precision improvement

significant impact than knowledge about the worst-case execution path, and they account for most of the precision improvement of the ILP-based approach. From the computational point of view, these precision issues do not require information about the worst-case execution path, and can be determined separately for each basic block. This also allows a better integration with pipeline analysis.

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

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

Table 3 shows the exact break-down of the precision improvement shown by the algorithmic approach, in terms of the various precision issues. For each benchmark, the second column contains the number of cache accesses which should be classified as AH, but are missed by must analysis, determined using the approach of Section 4.1. The third column contains the number of cache accesses which should be classified as Persistent, but are missed by persistence analysis, determined using the approach of Section 4.2. The fourth column contains the number of basic blocks for which the worst-case number of misses were less than the number of NC accesses, determined using Algorithm 1. Finally the fifth column contains the number of basic blocks which have worst-case profiles requiring more than one iteration of their enclosing loop, determined using Algorithm 2.

The results show that all four approaches are successful to different degrees in different benchmarks. Across the benchmarks, the approaches of Section 4.1 and 4.4, which find AH-accesses missed by must analysis and worst-case profiles of basic blocks inside loops are slightly more successful. The precision issue in the must analysis requires only if-statements (even without corresponding else segments) to manifest, which are more frequently found in the benchmarks. A large number of basic blocks also manifest worst-case profiles which require more than one iterations of their enclosing loops. Again, this precision issue only requires loops which have multiple paths from the entry to the exit of the loop. On the other hand, the approaches of Section 4.2 and 4.3 are slightly less successful. Most of the persistent accesses are identified by AI-based persistence analysis, however, there are benchmarks where our approach is able to identify higher number of persistent accesses. Similarly, our approach is also able to identify basic blocks where the maximum number of misses is strictly less than the number of NC-accesses.

Note that the number of accesses/BBs which benefit from the proposed approaches itself does not have a direct bearing on the amount of precision improvement in the WCET, since the execution counts of those accesses/BBs on the WCEP would also matter. The ILP-based approach only considers those accesses as misses whose miss paths are present on the worst-case path. This allows it to not only automatically subsume the precision improvement of all the four techniques used by the algorithmic approach, but also identify precision issues caused due to the worst-case path (as shown in the example of Figure 5). However, the ILP-based approach cannot take advantage of instruction parallelism. Finally, we note that since these benchmarks
Figure 7: Graph showing ratio of WCET obtained using ILP/Algorithmic approach as compared with AI-based approach for benchmarks with floating point operations are fairly small, the analysis time for all benchmarks was in the range of few seconds.

6.3 Benchmarks with Floating Point Operations

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

Figure 7 shows the WCET ratio \( \frac{WCET_x}{WCET_{AI}} \), where \( x \in \{ILP, Algorithmic\} \) for the four selected benchmarks. Note that a lower WCET ratio corresponds to larger precision improvement. The WCET ratio is greater than 1 for the ILP-based approach, for all the 4 benchmarks, which means that the WCET determined by the ILP-based approach is greater than the WCET determined by the AI-based approach. This has happened because any precision improvement due to better prediction of cache behavior has been overshadowed by the loss in precision due to lack of integration with pipeline analysis. In particular, in these benchmarks it is more likely that cache miss penalties are hidden by floating point operations occurring in parallel. Both the AI-based and algorithmic approach would be able to identify such scenarios statically during the pipeline analysis stage. The algorithmic approach continues to determine lower WCET estimates, with the average precision improvement over the 4 benchmarks being 2.4 %.
Table 4: DEBIE-1 Tasks

<table>
<thead>
<tr>
<th>DEBIE-1 Task</th>
<th>Root function</th>
<th>Code Size</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task - 1</td>
<td>TC_InterruptService</td>
<td>78 KB</td>
<td>4 KB</td>
</tr>
<tr>
<td>Task - 2</td>
<td>HandleTelecommand</td>
<td>420 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>Task - 3</td>
<td>HandleHealthMonitoring</td>
<td>321 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>Task - 4</td>
<td>TM_InterruptService</td>
<td>152 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>Task - 5</td>
<td>HandleHitTrigger</td>
<td>92 KB</td>
<td>4 KB</td>
</tr>
<tr>
<td>Task - 6</td>
<td>HandleAcquisition</td>
<td>466 KB</td>
<td>16 KB</td>
</tr>
</tbody>
</table>

6.4 DEBIE Benchmarks

We also experimented with a real-world benchmark, DEBIE-1, to show the scalability of the proposed approaches. The DEBIE-1 software is used to control the DEBIE-1 instrument, which is placed on a satellite to observe micro-meteoroids and small space debris by detecting impacts on sensors. The DEBIE-1 software itself contains six tasks, and in this experiment, we separately compute the WCET of the six tasks using AI-based, ILP and Algorithmic approach. Table 4 shows the root function of these tasks, along with the code size and the L1 cache size that we used while performing the experiments.

Out of the 6 tasks, the proposed approaches show zero or negligible precision improvement for 3 tasks. The precision improvement of the other 3 tasks is shown in Figure 8. The average precision improvement of the ILP-based approach is 1.6 %, while for the algorithmic approach, it is 1.9 %. The algorithmic approach provides higher precision improvement than the ILP-based approach for 2 out of the 3 tasks. Note that the DEBIE-1 tasks contain floating point operations, which would explain slightly better performance of the algorithmic approach. The average precision improvement of both the approaches, however, has decreased when compared with their performance on the smaller Mälardalen benchmarks. Since the DEBIE-1 tasks are
significantly larger, improving the cache prediction of some cache accesses does not have the same impact on the final WCET. On the other hand, the final WCET values of the DEBIE-1 benchmarks are also much larger as compared with the Mälardalen benchmarks, so that the decrease in WCET, in terms of the number of processor cycles, by the proposed approaches in the DEBIE-1 tasks is actually comparable to the total WCET values of majority of the Mälardalen benchmarks.

Table 5 shows the break-down of the precision improvement in terms of the various techniques used by the algorithmic approach (similar to table 3) for the DEBIE-1 Tasks and the benchmarks of Section 6.3. Again, all four techniques are successful to varying degrees across benchmarks. Notably, all benchmarks contain accesses which should be classified as AH but are missed by Must analysis. Majority of the benchmarks also have basic blocks whose worst-case profiles require multiple iterations of their enclosing loop.

Table 6 contains details about the time taken (in seconds) by various approaches. The second column contains the time taken by the AI-based approach, which includes the time taken for Must, May and Persistence analysis and solving the IPET ILP. The third column contains the time taken to find cache miss paths of accesses (the AI-based approach of Section 4.3). The fourth column contains the time taken to solve the ILP of Section 4.5. The last column contains the time taken by the algorithmic approach, which includes the time taken by Algorithms 1 and 2, and to solve the ILP of Section 4.4.5. Since both the ILP-based and algorithmic approach need cache miss paths, the total time taken by the ILP-based approach would be the sum of columns 3 and 4, while the total time taken by the algorithmic approach would be the sum of

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No. of accesses/BBs which benefit from approach of Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.1</td>
</tr>
<tr>
<td>basicmath</td>
<td>3</td>
</tr>
<tr>
<td>susan</td>
<td>7</td>
</tr>
<tr>
<td>fmref</td>
<td>63</td>
</tr>
<tr>
<td>audiobeam</td>
<td>9</td>
</tr>
<tr>
<td>DEBIE-1 T1</td>
<td>15</td>
</tr>
<tr>
<td>DEBIE-1 T4</td>
<td>10</td>
</tr>
<tr>
<td>DEBIE-1 T5</td>
<td>244</td>
</tr>
</tbody>
</table>

Table 5: Break-down of precision improvement
Benchmark | AI-based approach | Miss Path analysis | ILP-based Approach | Algorithmic Approach
--- | --- | --- | --- | ---
basicmath | 10 | 70 | 22 | 15
susan | 2.5 | 20 | 31 | 3
fmref | 2 | 57 | 4 | 2
audiobeam | 2 | 70 | 10 | 4
DEBIE-1 T1 | 6 | 113 | 23 | 13
DEBIE-1 T4 | 37 | 503 | 130 | 90
DEBIE-1 T5 | 8 | 61 | 43 | 13

Table 6: Analysis Time (in seconds)

The time taken to find miss paths dominates the analysis time of both the ILP-based and the algorithmic approach. One of the reasons could be that we perform a separate AI analysis for every basic block to find the cache miss paths, and finding the fix-point in the AI-based analysis requires a constant number of traversals of the entire CFG. However, the total analysis time is still reasonably small for both the ILP-based and algorithmic approaches even for larger benchmarks.

### 7 Related Work

Abstract Interpretation (AI) based approaches ([10, 7, 2, 3]) are widely used for cache analysis, because they guarantee safety, give adequately precise results, and scale well for large programs. However, these approaches are not sufficient to precisely capture cache behavior for WCET estimation. State space explosion is a serious problem while analyzing cache behavior, due to the exponential number of cache states that can arise during the program execution, especially as the cache associativity increases. Hence, some form of abstraction is necessary to ensure scalability, resulting in a trade-off between analysis precision and efficiency. Instead of maintaining several actual cache states which may be possible at a program point and which could be exponential in number, AI-based approaches determine a single abstract cache state at every program point which safely encodes the bare minimal information that is necessary to predict cache behavior. Such an approach is only sufficient to classify the behavior of individual cache accesses into a small number of classes. However, there are various instances of cache behavior which can be safely used for WCET estimation, but which
cannot be specified in terms of the hit-miss classifications determined using AI-based approaches.

Data Flow Analysis has also been used to perform cache analysis [11]. Conceptually, this approach is similar to AI-based cache analysis, as it defines a data flow analysis framework to find cache contents which may enter the cache across all executions, or cache contents which must be present in the cache across all executions. It also suffers from the same precision issues as AI-based analysis, as it uses the same classes for hit-miss classification of individual accesses.

There have been multiple efforts to use Model Checking for WCET analysis ([12, 13, 14, 15]). These approaches essentially explore the entire state space of all possible actual cache states and give precise cache analysis results, but they do not provide any bounds on the analysis time [16]. Most of these approaches have only been tested on small benchmarks, and some of them only consider the impact of processor pipeline, assuming absence of a cache.

There have also been efforts in combining cache analysis with path analysis to find the exact cache behavior along the worst case execution path (WCEP), most notably, the CSTG-based approach proposed by [17]. However, this approach can potentially introduce a very large number of variables and constraints in the ILP, and, is considered non-practical even for small programs [16].

Another factor which can have an impact on the precision of cache analysis is the presence of infeasible paths. There are few works ([18, 19]) which have used SAT-solvers to directly search for infeasible paths which can affect cache behaviour. In [18], the authors instrument the code by introducing variables to count the number of cache misses suffered by accesses, and then use SAT solvers to verify assertions on these variables. [19] modifies the AI-based approach for cache analysis, by annotating cache states with logic formulae, corresponding to partial paths along which the cache state would be realized. We note that cache miss paths provide an easy avenue for utilizing infeasible path information to improve the precision of cache analysis. Since a cache miss path is essentially a set of basic blocks, we can directly check the feasibility of the execution of all the basic blocks of a cache miss path in a single execution instance. Alternatively, if infeasible path information is already available, we can directly integrate it into our proposed ILP formulation which also includes cache miss paths.

The precision loss of the AI-based approaches mostly arises due to the imprecise nature of the join of abstract cache states at merge points. This issue has been identified before, but the join of abstract cache states is also responsible for a substantial reduction in state space being searched and reduction in the analysis time. In [20], the authors make a trade-off between the precision of the join and the expressiveness of the abstract cache states, by strengthening the former but significantly weakening the latter. They formulate a separate AI-based analysis for each basic block, and instead of maintaining complete information about cache states, they only maintain
information relative to the cache blocks accessed by the basic block under analysis. This allows them to avoid the loss in precision due to the join, and they find all the relative cache states possible at the start of the basic block under analysis, which are then used to find the maximum number of cache misses in the basic block.

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

8 Conclusion

In this work, we have proposed a new approach for cache analysis which does not provide cumulative hit-miss classifications to individual cache accesses. Instead, we aim to find worst-case behavior of groups of cache accesses and upper bounds on the number of misses caused by accesses inside loops. Such cache behavior cannot be depicted using hit-miss classifications such as Always-Hit and Persistent determined by previous approaches to cache analysis. Our approach works in two steps: we first analyze a small, fixed-size neighborhood of each access with complete precision and summarize the resulting information in the form of cache miss paths. We then perform a variety of analyses on the cache miss paths of accesses to obtain worst-case profiles of basic blocks, which indicate the maximum number of misses in the basic block as well as the minimum number of iterations of an enclosing loop required to cause those misses. We also propose an ILP-based approach to find the exact cache behavior in the worst-case execution instance. Experimentally, we show precision improvement in the WCET values as compared to previous approaches, and also show the scalability of our approach. We also specify how our approach can be easily integrated with pipeline analysis (which requires hit-miss classifications to individual accesses), thus taking advantage of available instruction parallelism to improve the WCET estimate.
References


