Refining Cache Behavior Prediction Using Cache Miss Paths

KARTIK NAGAR and Y. N. SRIKANT, Indian Institute of Science

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 that can be easily solved using cache miss paths. Experimental results over a wide range of benchmarks demonstrate precision improvement in WCET of multipath programs over previous approaches, and we also show how to integrate our approach with other microarchitectural analysis such as pipeline analysis.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Cache analysis, worst case execution time estimation

ACM Reference Format:
Kartik Nagar and Y. N. Srikant. 2017. Refining cache behavior prediction using cache miss paths. ACM Trans. Embed. Comput. Syst. 16, 4, Article 103 (May 2017), 26 pages.
DOI: http://dx.doi.org/10.1145/3035541

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

Kartik Nagar was supported by Microsoft Corporation and Microsoft Research India (under the Microsoft Research India PhD Fellowship Award) and the IMPECS Project.

Authors' addresses: K. Nagar (corresponding author) and Y. N. Srikant, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India; emails: kartik.n.nagar@gmail.com, srikant@csa.iisc.ernet.in.

DOI: http://dx.doi.org/10.1145/3035541

due to microarchitectural features such as caches, pipelines, branch prediction, and so forth 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 that 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 that can be safely used for WCET estimation, but that 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 [Ferdinand and Wilhelm 1999; Mueller 2000]. 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 single prediction for every access that 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. In 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 exist access patterns for which they fail to identify cache accesses that should actually be classified as Always-Hit or Persistent.

In order to solve these precision issues, we use the concept of **cache miss paths** [Nagar and Srikant 2015], 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 neighborhood just before the access, and hence, in most cases, it is sufficient to analyze a small, fixed-size neighborhood before 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 that could prevent them from

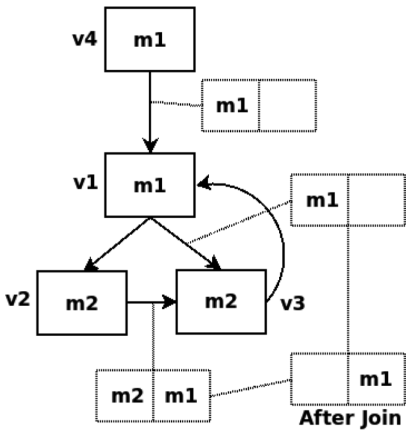

Fig. 1. Example demonstrating that must analysis does not determine all accesses that are guaranteed to hit the cache.

occurring either 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 $<Num\_misses, Iters>$, which denotes that the basic block can suffer *Num_misses* number of cache misses, only if it is executed once for every *Iters* number of iterations of its innermost enclosing loop. The scalability of our approach relies on the fact that we only focus on improving the combined cache behavior prediction of accesses inside the same basic block, and the analysis for every basic block is carried out independently. We also show how cache miss paths can be used to identify accesses that always hit the cache or are persistent, but are missed by the standard approaches.

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 approach can be integrated with pipeline analysis. This allows our 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. Further, our approach matches the precision improvement of (a slightly improved version of) the ILP-based approach proposed in Nagar and Srikant [2015] and performs better in benchmarks with floating-point operations, due to a smoother integration with pipeline analysis, which is not possible for the ILP-based approach. Due to the algorithmic nature of the proposed approach, it also has a lower asymptotic time complexity than the ILP-based approach.

2. EXAMPLES

In this section, we illustrate the precision issues with Abstract Interpretation (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. AI-based *must* analysis is used to find cache blocks that 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, we show that it can actually overlook accesses that can be actually guaranteed to always hit the cache. Consider Figure 1, which shows a portion of a program CFG and the instruction cache accesses therein. $m1, m2$ indicate cache

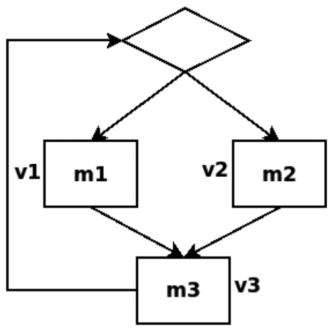

Fig. 2. Example demonstrating that persistence analysis does not determine all persistent accesses.

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 toward the left). As mentioned earlier, *must* cache analysis computes, at a program point, all those cache blocks that 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 $v1$, $v2$, and the result of their join at the start of $v3$. The access to $m2$ in $v2$ increases the age of $m1$, but also brings $m2$ in the cache. However, since join in *must* analysis at a merge point selects only those cache blocks that are present in the *must* caches at the end of all predecessor basic blocks and also takes their maximum age, $m2$ will not be present in the *must* cache after join, but its effect on the age of $m1$ will be retained. As a result, the access to $m2$ in $v3$ will evict $m1$ out of the *must* cache, due to which the access to $m1$ in $v1$ will not be classified as Always-Hit. However, it can be clearly seen that $m1$ is guaranteed to be present in the cache at the start of $v1$, and *must* analysis is incorrectly adding the aging effect of accesses to the same cache block ($m2$) to another cache block ($m1$) multiple times.

Note that real programs frequently exhibit such behavior for instruction caches. For example, the last instruction of $v1$ could be a conditional jump, with $v2$ being the fall-through basic block and $v3$ the target of the jump. In such a scenario, the instructions toward the end of $v2$ and beginning of $v3$ would be contiguous in the address space, and could map to the same cache block.

For instruction caches, Persistence analysis is often more effective than *must* analysis, because it identifies those cache blocks that 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 [Huynh et al. 2011; Cullman 2013] 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, $m1$ would be declared as persistent. However, there are other precision issues with Persistence analysis, and it may also overlook cache accesses that are actually persistent.

Consider the program CFG shown in Figure 2. Assume that cache blocks $m1, m2, m3$ map to the same cache set, and the cache associativity is 2. In this example, $m1$ and $m2$ are not persistent, but $m3$ is persistent. However, the set of younger cache blocks of

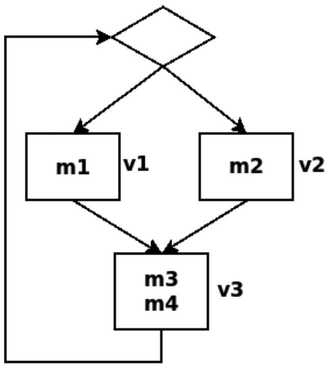

Fig. 3. Example demonstrating the limitation of per-access hit-miss classifications such as Always Hit and Persistent.

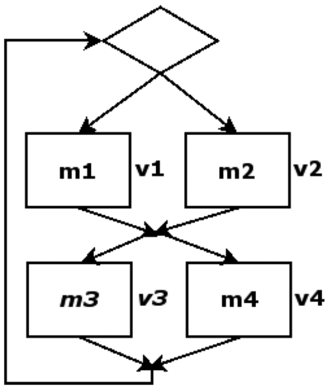

Fig. 4. Example demonstrating that inside loops, the worst-case cache behavior may only occur for a subset of the iteration space.

$m3$ would contain both $m1$ and $m2$, and hence Persistence analysis would not be able to identify $m3$ as persistent.

Further, classifications such as Always Hit and Persistent are not enough to capture precise cache behavior. These classifications apply to individual accesses, but often, a more precise prediction can be made about a group of accesses. For example, consider the program CFG shown in Figure 3. Assume that cache blocks $m1, m3$ map to cache set $s1$, while $m2, m4$ map to cache set $s2$ and the cache associativity is 1. We focus on the accesses to $m3, m4$ in basic block $v3$. Note that neither of these accesses can be classified as Always Hit or Persistent. $m1$ in $v1$ will evict $m3$, while $m2$ in $v2$ will evict $m4$. However, both these evictions cannot happen simultaneously in the same iteration. In other words, at least one cache hit in $v3$ is guaranteed to occur in every iteration (except possibly the first).

Accesses inside loops may not exhibit the worst-case behavior in every iteration, but only in a subset of the iteration space. Consider the program CFG in Figure 4. Assume that $m1, m2, m3$ map to the same cache set ($m4$ maps to a different cache set), and the cache associativity is 2. We focus on the access to $m3$ in basic block $v3$. Once $m3$ is brought into the cache, both $m1$ and $m2$ need to be accessed to evict $m3$ and cause the next access to become a miss. However, $m1$ and $m2$ are in different basic blocks, and only one of these basic blocks can be executed during an iteration. Hence, the access to $m3$ cannot cause a cache miss in every iteration of the loop. The execution must

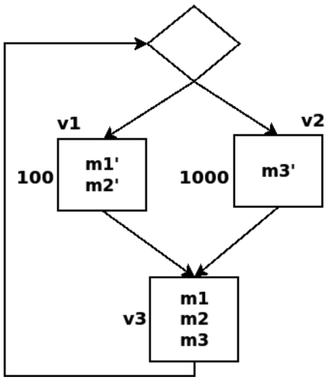

Fig. 5. Example demonstrating the difference between the worst-case cache behavior and the cache behavior in the worst-case execution path.

alternate between basic blocks $v1$ and $v2$, and it must skip $v3$ (and execute $v4$) in every alternate iteration. Note that the access to $m3$ cannot be classified as either Always-Hit or Persistent, but the maximum number of misses that it can cause is equal to half the total number of iterations of the loop.

Finally, in all of the aforementioned examples, our aim has been to find the worst-case cache behavior that nonetheless holds across all execution instances. However, knowledge about the WCEP can be used to find the exact cache behavior, which only holds in the worst-case execution instance. If it is known that the execution path with the maximum execution time can never pass through some basic block, then the destructive effects of this basic block need not be considered while analyzing cache behavior. Hence, using WCEP information can improve the precision of cache analysis.

For example, consider the program CFG shown in Figure 5. Along with the cache accesses made by basic blocks $v1, v2, v3$, we also know the maximum execution time (i.e., WCET) of the basic blocks $v1$ and $v2$ to be 100 and 1,000 cycles, respectively. Then, the WCEP will pass through $v2$. Assume that $m1, m1'$ map to cache set $s1$, $m2, m2'$ map to cache set $s2$, and $m3, m3'$ map to cache set $s3$, and the cache associativity is 1. We focus on the basic block $v3$. It is clear that even though the worst-case cache behavior of $v3$ is two cache misses in every iteration (accesses to $m1$ and $m2$), execution along the WCEP will only cause one cache miss (to $m3$) in every iteration (except possibly the first).

However, note that the estimation of the WCEP itself depends on the predicted cache behavior. In the previous example, even though the WCET of $v2$ is greater than $v1$, execution of $v1$ is also responsible for more cache misses, and if the latency of the extra cache misses caused by $v1$ in combination with the WCET of $v1$ is greater than the combined latency of the misses caused by $v2$ and its WCET, then the WCEP would pass through $v1$. 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 using 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 that 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 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. We assume that all function calls are virtually in-lined in the CFG. 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 $Acc : V \times \mathcal{S} \rightarrow \mathbb{P}(\mathcal{B})$ 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 $Acc_a : V \times \mathcal{B} \rightarrow \mathbb{P}(\mathcal{B})$ and $Acc_b : V \times \mathcal{B} \rightarrow \mathbb{P}(\mathcal{B})$, such that $Acc_a(v, m)$ 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 σ in G is a sequence of basic blocks $v_1 v_2 \ldots v_p$ such that $(v_i, v_{i+1}) \in E$, for all i, $1 \leq i$, $i \leq p - 1$ (note that repetition of basic blocks is allowed). We say that there exists a walk from v_1 to v_k if there exists a walk $v_1 v_2 \ldots v_k$ (i.e., a walk that begins with v_1 and ends with v_k). Note that a trivial walk exists between a basic block and itself. We lift the Acc function to σ 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 that are not classified as Always Hit by *must* cache analysis.

Definition 3.1. A **concrete cache miss path** of an access to $m \in \mathcal{B}$, mapped to $s \in \mathcal{S}$, in basic block v is defined as a walk $\sigma = v_1 v_2 \ldots v_p v$ in the CFG G with the following properties:

(1) $m \notin \bigcup_{i=2}^{p} Acc(v_i, s)$,
(2) $(|Acc_a(v_1, m) \cup \bigcup_{i=2}^{p} Acc(v_i, s) \cup Acc_b(v, m)| \geq k) \vee (v_1 = v_{start} \wedge m \notin Acc(v_{start}, s))$, and
(3) $|\bigcup_{i=2}^{p} Acc(v_i, s) \cup Acc_b(v, m)| < k$.

Property (1) states that the cache block m is not accessed in any intermediate basic block of the cache miss path. Property (2) states that either the total number of distinct cache blocks mapped to cache set s and accessed on the cache miss path exceeds k or the cache miss path begins from the start basic block, and m is not accessed anywhere on the miss path. Property (3) states that any suffix of the cache miss path is not a cache miss path. Together, these properties ensure that a concrete cache miss path is a walk in G along which the access will suffer a cache miss.

Concrete cache miss paths provide a necessary and sufficient condition for cache misses.[1] However, they can be arbitrarily large, and hence we abstract them in two

[1]Note that a concrete miss path may be infeasible, and hence may never be actually executed. However, if a concrete miss path is executed, it will cause a cache miss.

ways: we allow "gaps" in the miss paths and include only those basic blocks that 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. Note that $T > 0$.

Given a concrete cache miss path $\sigma = v_1 v_2 \dots v_p v$ of access to m in v, let $\alpha_{v,m}(\sigma) = \{v_i : Acc(v_i, Set(m)) \neq \phi\}$. We define $\alpha^T_{v,m}(\sigma) = \alpha_{v,m}(\sigma)$ if $|\alpha_{v,m}(\sigma)| \leq T$; otherwise, if j is the largest subscript in σ (in other words, $v_j v_{j+1} \dots v_p v$ is the smallest suffix of σ) such that $|\alpha_{v,m}(v_j v_{j+1} \dots v_p v)| = T$, then $\alpha^T_{v,m}(\sigma) = \alpha_{v,m}(v_j \dots v_p v)$.

Definition 3.2. Given a concrete cache miss path σ 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 that 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. On the other hand, if $T = 1$, then $\alpha^T_{v3,m3}(v1v3) = \{v3\}$. In general, it is recommended that T must be strictly greater than 1, because otherwise every access would only get a single miss path that would consist of the basic block containing the access. In Figure 4, $v1v4v2v3$ is a concrete miss path of access $m3$ in $v3$, and with $T = 3$, $\alpha^T_{v3,m3}(v1v4v2v3) = \{v1, v2, v3\}$, because $v4$ does not access any cache block mapped to the same cache set as $m3$.

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 $\mathcal{L} = (\mathbb{P}(\mathbb{P}(V)), \subseteq)$. Each element is a set of possible abstract cache miss paths. The analysis is carried out in the backward direction in the CFG. For every basic block w, the approach maintains an IN element $IN_w \in \mathcal{L}$ at the end of the basic block and an OUT element $OUT_w \in \mathcal{L}$ at the beginning of the basic block. Every basic block w is also associated with a transfer function $f_w : \mathcal{L} \rightarrow \mathcal{L}$ such that $OUT_w = f_w(IN_w)$. Also, $IN_w = \bigcup_{(w,u) \in E} OUT_u$.

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

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

$$f_w(P) = P$$

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

$$\begin{aligned} f_w(P) = &\{\pi \in P : DB_{v,m}(\pi \setminus \{v\}) \geq k \vee |\pi| = T\} \\ &\cup \{\pi \cup \{w\} : \pi \in P \wedge DB_{v,m}(\pi \setminus \{v\}) < k \wedge |\pi| < T\} \end{aligned}$$

Case 3 : $w \neq v$, $m \in Acc(w, s)$

$$\begin{aligned} f_w(P) = &\{\pi \in P : DB_{v,m}(\pi \setminus \{v\}) \geq k \vee |\pi| = T\} \\ &\cup \{\pi \cup \{w\} : \pi \in P \wedge DB_{v,m}(\pi \setminus \{v\}) < k \wedge |\pi| < T \\ &\qquad \wedge DB_{v,m}((\pi \cup \{w\}) \setminus \{v\}) \geq k\} \end{aligned}$$

Case 4 : $w = v$

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

Table I. AI-Based Miss Path Analysis Applied to the Example of Figure 1

Iteration	IN_{v_1}	OUT_{v_1}	IN_{v_2}	OUT_{v_2}	IN_{v_3}	OUT_{v_3}
1	ϕ	$\{\{v_1\}\}$	ϕ	ϕ	ϕ	ϕ
2	ϕ	$\{\{v_1\}\}$	ϕ	ϕ	$\{\{v_1\}\}$	ϕ
3	ϕ	$\{\{v_1\}\}$	ϕ	ϕ	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$
4	$\{\{v_1, v_3\}\}$	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$	ϕ	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$
5	$\{\{v_1, v_3\}\}$	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$	$\{\{v_1, v_2, v_3\}\}$	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$
6	$\{\{v_1, v_3\}, \{v_1, v_2, v_3\}\}$	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$	$\{\{v_1, v_2, v_3\}\}$	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$
7	$\{\{v_1, v_3\}, \{v_1, v_2, v_3\}\}$	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$	$\{\{v_1, v_2, v_3\}\}$	$\{\{v_1\}\}$	$\{\{v_1, v_3\}\}$

Case 5 : $w = v_{start}$

$$f_{v_{start}}(P) = \{\pi \in P : DB_{v,m}(\pi \setminus \{v\}) \geq k \vee |\pi| = T\}$$
$$\cup \{\pi \cup \{v_{start}\} \ : \pi \in P \wedge DB_{v,m}(\pi \setminus \{v\}) < k \wedge |\pi| < T\}$$

An incomplete miss path is a set of basic blocks π such that $|\pi| < T$ and $DB_{v,m}(\pi \setminus \{v\}) < k$. The transfer function does not change the incoming state if the basic block does not access the cache set s (Case 1). If the basic block w does not access m but accesses some other cache block mapped to s, then w is simply added to those miss paths that are incomplete, in addition to retaining completed miss paths (Case 2). If the basic block w does access m, then only those miss paths that 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 ϕ.

Table I 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 a backward direction, the *OUT* values correspond to the beginning of a basic block, while the *IN* values 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.

The following theorem asserts that the AI-based approach determines abstract miss paths for all concrete miss paths of m; that is, $OUT_{v_{start}} = \{\alpha^{T}_{v,m}(\sigma) : \sigma$ is a concrete miss path of $m\}$. The proof can be found in Online Appendix A.

THEOREM 3.3. *For every concrete cache miss path σ of access r in basic block v, there exists an abstract cache miss path $\pi \in OUT_{v_{start}}$ such that $\pi = \alpha^{T}_{v,m}(\sigma)$.*

4. ALGORITHMS

We now show how cache miss paths can be used to tackle the various precision issues discussed in Section 2. The proofs of all the lemmas and theorems in this section can be found in Online Appendix B.

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

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 basic block. We say that a miss path is completely inside loop L if every basic block of the miss path is inside either L or an inner loop of L. An access is said to be persistent in a loop L if it can cause at most one cache miss every time execution enters L from outside.

THEOREM 4.2. *If an access to m in v does not have any abstract cache miss paths that are completely inside an enclosing loop L, then m is persistent in loop L.*

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 number of iterations 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 innermost loop that 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 $m1$ and $m2$ have miss paths inside the loop ($\{v2, v3\}$ and $\{v1, v3\}$, respectively), but $m3$ 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 behavior of a group of cache accesses, where the individual accesses themselves might not always hit the cache or be persistent. Here, we focus only on those accesses that are classified as neither Always-Hit nor Persistent and are present inside the same basic block. Our approach is based on finding cache miss paths of such accesses that can never be executed together.

Definition 4.3. Given cache accesses r_1 and r_2 in basic block v, and their miss paths π_1 and π_2, respectively, we say that π_1 and π_2 do **not conflict** with each other if there

Table II. Data-Flow Analysis $\mathcal{D}_v$ to Determine Conflict Information for Basic Block v

	GEN Set	KILL Set
Basic block v	ϕ	V
For every other basic block w	$\{w\}$	ϕ

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 π_1 and π_2 **conflict** with each other.

If two miss paths conflict, then they cannot cause cache misses together. In the example in Figure 3, miss path $\pi_1 (= \{v1\})$ of $m3$ and $\pi_2 (= \{v2\})$ of $m4$ conflict with each other, and hence cannot cause cache misses together. The following results provide the necessary and sufficient conditions required to automatically find such miss paths.

LEMMA 4.4. *Given a set of basic blocks $W = \{v_1, \ldots, v_n\}$ and basic block v ($v \notin W$), 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 that does not pass through v, and then there exists a walk in G that contains all the basic blocks in W and also does not pass through v.*

LEMMA 4.5. *Miss paths π_1 and π_2 of two accesses in v do not conflict $\Leftrightarrow \forall w_1 \in \pi_1$, $\forall w_2 \in \pi_2$, and there exists a walk in G either from w_1 to w_2 or from w_2 to w_1 that does not pass through v.*

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 that does not pass through v. A simple way to do this is to formulate a Data-Flow Analysis (DFA) [Aho et al. 1986]. For basic block v, the DFA $\mathcal{D}_v$ 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 that does not pass through v.

For $\mathcal{D}_v$, 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 II. The DFA fixpoint algorithm calculates the IN_w and OUT_w sets for all basic blocks w, which obey the following equations : $IN_w = \bigcup_{u:(u,w)\in E} OUT_u$ and $OUT_w = GEN_w \cup (IN_w \setminus KILL_w)$. At the end of the analysis, IN_w will contain all basic blocks that 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 that 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 4.6. *Given miss paths π_1 and π_2 of two accesses in v, π_1 and π_2 do not conflict $\Leftrightarrow \forall w_1 \forall w_2 \in \pi_1 \cup \pi_2,\ (w_1 \in IN_{w_2} \vee w_2 \in IN_{w_1})$.*

LEMMA 4.7. *Given miss paths $\pi_1, \ldots, \pi_n$, of accesses in v, there exists a walk in G that contains all the miss paths and contains v at the end if and only if there is no pairwise conflict in the set $\{\pi_1, \ldots, \pi_n\}$.*

To find the maximum number of miss paths that 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 that neither are 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 π_i of access r_i and miss path π_j of access r_j if $r_i \neq r_j$ and π_i and π_j do not conflict with each other.

THEOREM 4.8. *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.*

For example, the MPCG for the basic block $v3$ in Figure 3 will have two vertices, corresponding to the miss paths of $m3$ and $m4$. Since these miss paths conflict with each other, there will be no edge between them, so that the size of the maximum clique in the MPCG will be 1.

ALGORITHM 1: Algorithm to Find the Maximum Number of Cache Misses in Every Basic Block of the Program

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

Algorithm 1 summarizes the various steps discussed so far. We concentrate on the NC (nonclassified) 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. By Theorem 3.8, the output of Algorithm 1 can be safely used to find the WCET of a basic block.

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 to 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 total complexity of the algorithm is $O(|V|(|E| + 2^m))$.

The size of the maximum clique in the MPCG can still overestimate the maximum number of cache misses, because of various abstractions used while determining abstract cache miss paths, that is, allowing gaps in the miss path, bounding its maximum size, infeasibility of paths, and so forth. Due to these abstractions, execution of an abstract miss path may not necessarily result in a cache miss.

4.4. Finding Worst-Case Profiles of Basic Blocks

For basic blocks 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 $v3$ is one, but it cannot cause one cache miss in every iteration. Hence, for basic blocks inside loops, along with finding the maximum number of cache misses, we would also like to find the minimum number of iterations required to cause those misses. We call this information the worst-case profile of a basic block, represented as the tuple, $<Num\_misses, iters>$, which says that *Num_misses* number of cache misses can happen in the basic block, only if it is executed once for every *iters* number of iterations of its innermost enclosing loop. In the example of Figure 4, the worst-case profile of $v3$ is $<1, 2>$. Note that a basic block can have multiple profiles. For example, $<0, 1>$ is also a profile of $v3$.

In order to find all 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, and the fact that the CFG is reducible. Since every iteration of the loop must begin with the execution of v_h, all miss paths can occur together in a single iteration only if there exists a walk that does not pass through v_h between every pair of basic blocks in the miss paths.

In the following, we only consider those miss paths that are completely present in the innermost loop containing the basic block (although the approach can be applied at all nesting levels). 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 $v_1 \rightsquigarrow_w v_2$ to mean that there exists a walk in G from v_1 to v_2 that 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 4.7, we know that if $\forall v_1, v_2 \in \cup_{i=1}^k \pi_i$, either $v_1 \rightsquigarrow_v v_2$ or $v_2 \rightsquigarrow_v v_1$, and then all the miss paths π_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 4.9. *Miss paths π_1 of access r_1, π_2 of $r_2, \ldots, \pi_k$ of r_k in v can cause k misses in v in consecutive iterations of $L \Leftrightarrow$ there exists a walk from v to v that contains exactly one instance of v_h and contains all the miss paths.*

LEMMA 4.10. *Given miss paths π_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 $v_h \Leftrightarrow$ $\forall v_1, v_2 \in \cup_{i=1}^k \pi_i$, $v_1 \rightsquigarrow_v v_2 \vee v_2 \rightsquigarrow_v v_1$ and $\forall v_1, v_2 \in \cup_{i=1}^k \pi_i \cup \{v\}$, $v_1 \rightsquigarrow_{v_h} v_2 \vee v_1 \rightsquigarrow_{v_h} v_2$.*

The implication is that if every walk between two basic blocks in a loop must pass through the entry basic block v_h (e.g., between $v1$ and $v2$ 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 4.11. *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 $v_h \Leftrightarrow \forall w_i, w_j$, $1 \leq i < j \leq k$, neither $w_i \rightsquigarrow_{v_h} w_j$ nor $w_j \rightsquigarrow_{v_h} w_i$.*

THEOREM 4.12. *Given miss paths π_1 of access $r_1, \ldots, \pi_k$ of access r_k in v, where $\pi_1, \ldots, \pi_k \in V_M^L$, if there exists $W_C \subseteq \cup_{i=1}^k \pi_i \cup \{v\}$ such that $\forall w, w' \in W_C$ neither $w \rightsquigarrow_{v_h} w'$ nor $w' \rightsquigarrow_{v_h} w$, 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.*

We now apply these results to find the worst-case profiles, as depicted in Algorithm 2. We start with the MPCG (V_M^L, E_M^L) and find the clique of maximum size (line 1). We perform the data-flow analysis $\mathcal{D}_{v_h}$ to find conflict information, so that IN_w (for all w)

ALGORITHM 2: Algorithm to Find All Worst-Case Profiles of a Basic Block v Inside a Loop with Entry Block v_h

```
1: Num_misses ← Size of maximum clique in MPCG ($V_M^L$, $E_M^L$) of $v$
2: Perform Data-flow conflict analysis $\mathcal{D}_{v_h}$
3: repeat
4:    for all Clique $S$ of size Num_misses in MPCG do
5:       Add <$Num\_misses$, $iters(S)$> to the worst-case profiles of $v$
6:    end for
7:    Num_misses ← Num_misses - 1
8: until $iters(S) = 1$ or Num_misses = 0
9: if Num_misses = 0 then
10:    Add <0, 1> to the worst-case profiles of $v$
11: end if
12:
13: function iters ($S$)
14: $V_B$ ← Set of all basic blocks in miss paths of $S$ and $v$
15: for all $w_1, w_2 \in V_B$ do
16:    if $w_1 \notin IN_{w_2} \wedge w_2 \notin IN_{w_1}$ then
17:       Add $(w_1, w_2)$ to $E_B$
18:    end if
19: end for
20: return Size of maximum clique in ($V_B$, $E_B$)
```

will consist of those basic blocks that have a walk in G to w that does not pass through v_h (line 2). This ensures that if for w_1 and w_2 neither of the two is present in the IN set of the other, then neither $w_1 \rightsquigarrow_{v_h} w_2$ nor $w_2 \rightsquigarrow_{v_h} w_1$.

We keep finding cliques (of possibly decreasing sizes) in the MPCG until we find a clique that requires only one iteration or we exhaust all cliques (lines 3–8). We store the size and the number of iterations required by all cliques discovered in this process (line 5). The function $Iters(S)$ finds the number of iterations required by clique S (lines 13–20). 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 14). We add an edge between two basic blocks if there does not exist a walk between them that does not pass through v_h (lines 15–19). A clique S_B in the BBCG means (by Theorem 4.12) that a walk that 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 20). 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 that does not pass through v_h, and by Lemmas 4.9 and 4.10, this guarantees that the miss paths can cause misses in v in consecutive iterations. Note that after the algorithm terminates, we do some postprocessing to ensure that we only have one WC profile for a given number of iterations $iters$, which is the one with the maximum number of misses.

For example, let us apply Algorithm 2 to find the worst-case profiles of $v3$ in Figure 4. Since $v3$ has only one access ($m3$), and $m3$ only has a single miss path ($\pi_1 = \{v1, v2\}$), the MPCG will contain only one vertex, and the size of the maximum clique in the MPCG will be 1. Next, we apply the $iters$ function on π_1. The BBCG will have two vertices, corresponding to $v1$ and $v2$, and since there does not exist a path between them that does not pass through the loop entry, an edge will be added, so that the size of the maximum clique in the BBCG will be 2. Hence, the worst-case profile <1, 2> will be added for $v3$. π_1 is the only clique in the MPCG; hence, Num_misses will be set to 0, and we will exit the repeat-until loop. Finally, the profile <0, 1> will also be added.

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 *Iters* can be called for every clique. The complexity of the *Iters* function is exponential in the size of the BBCG, but since the maximum length of a miss path and 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 [Li et al. 1995] that 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 $p = <Num\_misses_p, iters_p>$ of basic block v, let $W_{v,p}$ be the WCET of v assuming Num_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 as follows:

$$\text{Maximize}$$

$$\sum_{v \in V} \sum_{\text{WC profile } p \text{ of } v} W_{v,p} y_{v,p} \tag{1}$$

$$\text{Subject to}$$

$$\forall v \in V, \quad \sum_{(x,v) \in E} z_{x,v} = y_v = \sum_{(v,u) \in E} z_{v,u} \tag{2}$$

$$\forall v \in V, \quad \begin{cases} y_v = \sum_{\text{profile } p \text{ of } v} y_{v,p} & (3) \\ \sum_{\text{profile } p \text{ of } v} iters_p y_{v,p} \le C_v. & (4) \end{cases}$$

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 of the basic block on the WC path (Equation (3)). Every occurrence of the worst-case profile p will consume $iters_p$ number of iterations, and the maximum number of iterations is upper bounded by C_v. This establishes an upper bound on $y_{v,p}$, which is the maximum number of times v can cause Num_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, this ILP formulation allows the possibility of following different paths across iterations and uses the appropriate WCET for basic blocks in such cases.

Note that the meaning of a worst-case profile $<N, I>$ of basic block v is that there must be I consecutive iterations, in $I - 1$ of which v is not executed, and only then in the Ith iteration, v will suffer N cache misses. This places a coarse upper bound on the number of times v can suffer N misses, since if C is the total number of iterations, then v can suffer N misses at most $\frac{C}{I}$ times. If v has two worst-case profiles $<N, I>$ and $<N', I'>$ such that $I' < I$, then $N' < N$, because otherwise, the profile $<N, I>$ does not give a valid upper bound on the number of times v can suffer N misses.

Hence, if v has k profiles $<N_1, I_1>$, $<N_k, I_k>$ such that $I_1 > I_2 > \cdots > I_k$, and if y_i is the execution count of v in profile $<N_i, I_i>$ (for all i), then the iteration space can be broken down into disjoint sets of iterations of sizes I_1y_1, I_2y_2, I_ky_k. In the iteration set I_iy_i, v is executed only y_i times, but this consumes I_iy_i iterations. This gives rise to Equation (4) in the ILP.

5. ILP-BASED APPROACH

While the algorithms in the previous section can tackle the precision issues illustrated in Figures 1 to 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 behavior, and hence one cannot directly use the WCEP obtained using the IPET formulation. In this section, we show how to directly 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 to 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 $Acc_f : V \rightarrow \mathcal{B}$, and $Acc_f(v)$ gives the set of cache blocks accessed in v, such that the first access to these cache blocks in v is nonclassified. For each basic block v, let $MP_v : Acc_f(v) \rightarrow \mathbb{P}(\mathbb{P}(V))$ 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. Note that we also enforce an upper bound N on the number of miss paths of an access. Hence, if $|MP_v(m)| > N$ for some $m \in Acc_f(v)$, then m is classified as a Miss, and is removed from $Acc_f(v)$. 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 W_v be the estimated WCET of basic block v obtained by assuming that all NC accesses hit the cache.

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

$$\text{Maximize} \sum_{v \in V}(W_v y_v + \sum_{m \in Acc_f(v)} CMP\ x_{v,m}) \tag{5}$$

$$\text{subject to}$$

$$\forall v \in V, \quad y_v = \sum_{(x,v)\in E} z_{x,v} = \sum_{(v,u)\in E} z_{v,u} \tag{6}$$

$$\forall v \in V,\ \forall m \in Acc_f(v), \begin{cases} x_{v,m} \le \sum_{\pi \in MP_v(m)} x^{\pi}_{v,m} & (7) \\ \forall w \in BB_{v,m}, \sum_{\pi \in MP_v(m): w \in \pi} x^{\pi}_{v,m} \le y_w. & (8) \end{cases}$$

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 *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 π of the access to m in v, the variable $x^{\pi}_{v,m}$ counts the number of misses suffered by the access along π. 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 (e.g., 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 $x^{\pi}_{v,m}$ variables (Equation (7)). A miss path is present on the WCEP if the execution counts of all basic blocks on the miss path are nonzero. 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 of these requirements.

In addition to these 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 previous ILP, we can guarantee not only 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.

This formulation is more precise than the ILP formulation proposed in Nagar and Srikant [2015]. The major difference between the two formulations is in Equation (8), where, in Nagar and Srikant [2015], 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 miss path π_1 is completely contained inside another miss path π_2, and if the WCEP contains π_2, then the formulation in Nagar and Srikant [2015] 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 previous 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 [Li et al. 2007]. 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. We note that the proposed approaches in their current formulation will only work for instruction caches, since we assume unique cache blocks referenced by every access. Using our approach for data caches will require a nontrivial extension. Further, we assume a single-level hierarchy, although the proposed techniques can also be applied for analysis of the L1 cache in a multilevel noninclusive cache hierarchy.

For cache analysis, Chronos uses Abstract Interpretation-based *must* and *may* cache analysis [Ferdinand and Wilhelm 1999] and the safe version of Persistence analysis [Huynh et al. 2011] to provide a hit-miss classification to every memory access. For pipeline analysis, Chronos builds a separate execution graph [Li et al. 2006] for every basic block, modeling all the pipeline stages of every instruction in the basic block, their execution time estimates, and their interdependencies. It also includes prologues and epilogues, that is, instructions occurring before and after the basic block in the program, to take into account pipeline effects that cross basic block boundaries. Note that construction of the execution graph 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 that remain not classified 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 W_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 overestimate the WCET by ignoring the effect of instruction parallelism.

To make matters worse, some architectures may exhibit strong impact timing anomalies [Wenzel et al. 2005], 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. Wenzel et al. [2005] list the two main factors that may cause strong impact timing anomalies: (1) an out-of-order functional unit that allows instructions to be dispatched in an order different from the program order or (2) multiple, in-order, nonuniform 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, five-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 W_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 that 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 number of misses possible in basic block v for the profile p is k $(= Num\_misses_p)$ and the number of NC accesses is n, then we create $\binom{n}{n-k}$ versions of v, corresponding to every choice of $n-k$ accesses. For a given version corresponding to a particular choice of $n-k$ accesses, we change the hit-miss classification of these $n-k$ accesses from NC to AH, and 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 $W_{v,p}$ in the modified IPET ILP formulation described toward 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 that 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 the 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*[2] library for finding maximal cliques in graphs. We experimented on benchmarks chosen from Mälardalen, MiBench, StreamIt, and DEBIE suites (obtained from the TACLeBench collection[3]). For each benchmark, we use a different L1 instruction cache, with cache size approximately equal to 10% to 20% of the code size and minimum and maximum cache sizes being 128 bytes and 16KB, respectively. Table VII in Online Appendix B lists the exact code size (in bytes) and cache configuration for every benchmark. The L1 cache hit latency is one cycle, while the main memory latency is 30 cycles.

While determining the cache miss paths of accesses, the maximum miss path length (T) is restricted to 16, and the maximum number of miss paths per access (N) is restricted to 100. Larger miss paths increase the likelihood of finding conflicting basic blocks or basic blocks that are not on the worst-case path. On the other hand, 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

[2]http://users.aalto.fi/ pat/cliquer.html.
[3]http://www.tacle.eu/index.php/activities/taclebench.

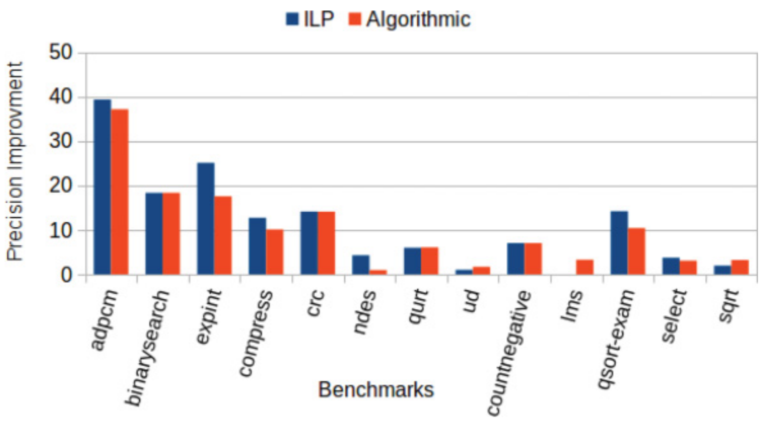

Fig. 6. Graph showing precision improvement (in %) in WCET of Mälardalen benchmarks.

of finding conflicting miss paths that 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 the AI-based approach. Figure 6 shows the precision improvement (in %) of the WCET computed using the ILP-based and algorithmic approaches, as compared to the WCET computed using the AI-based approach, for benchmarks from the Mälardalen suite. The precision improvement is computed as $\frac{WCET_{AI}-WCET_x}{WCET_{AI}}\%$, where $x \in \{Algorithmic, ILP\}$. 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 nonzero 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 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 four benchmarks (*qurt*, *ud*, *lms*, *sqrt*) where the algorithmic approach provides higher precision improvement than the ILP-based approach, three 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 of these benchmarks have multiple paths (i.e., if-then-else or switch-case statements), and in fact, these are the only multipath benchmarks in the Mälardalen suite, while all the other benchmarks for which the proposed approaches do not show any precision improvement contain only a single path. In all, we tested

Table III. Breakdown of Precision Improvement

Benchmark	No. of Accesses/BBs That Benefit from Approach of Section			
	4.1	4.2	4.3	4.4
adpcm	0	0	2	0
binarysearch	0	0	2	0
expint	1	0	0	3
compress	5	2	0	8
crc	0	2	0	4
ndes	2	0	1	0
qurt	6	0	0	0
ud	4	0	0	0
countnegative	0	1	2	2
lms	2	0	0	0
qsort-exam	0	0	0	2
select	0	0	0	3
sqrt	1	0	0	1

our approach on 25 benchmarks, out of which 13 showed precision improvement. The precision issues that are targeted by our 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. In general, programs with branching inside loops would benefit largely from the proposed approaches.

Table III shows the exact breakdown 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 that 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 that 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 was less than the number of NC accesses, determined using Algorithm 1. Finally, the fifth column contains the number of basic blocks that 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 Sections 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 that require more than one iteration of their enclosing loops. Again, this precision issue only requires loops that have multiple paths from the entry to the exit of the loop. On the other hand, the approaches of Sections 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 a 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 that 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

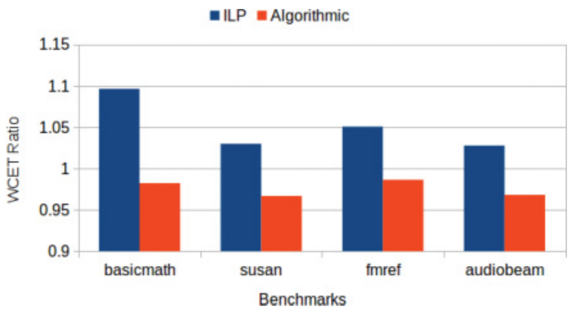

Fig. 7. Graph showing ratio of WCET obtained using ILP/algorithmic approach as compared with AI-based approach for benchmarks with floating-point operations.

subsume the precision improvement of all 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 are fairly small, the analysis time for all benchmarks was in the range of a 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 the 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 that contain plenty of floating-point operations. The selected benchmarks are *basicmath* and *susan* from the *MiBench* suite, and *audiobeam* and $fmref$ from the *StreamIt* suite. The code sizes of these benchmarks are 116KB, 48KB, 47KB, and 48KB, respectively. For this experiment, we assumed a four-way 4KB 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 four 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 four benchmarks being 2.4%.

6.4. DEBIE Benchmarks

We also experimented with a real-world benchmark, DEBIE-1, to show the scalability of the proposed approaches. The DEBIE-1 software is used to control the DEBIE-1 instrument, which is placed on a satellite to observe micro-meteoroids and small space debris by detecting 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 the AI-based, ILP, and algorithmic approach. Table IV shows the root function of these

Table IV. DEBIE-1 Tasks

DEBIE-1 Task	Root Function	Code Size	Cache Size
Task - 1	TC_InterruptService	78KB	4KB
Task - 2	HandleTelecommand	420KB	16KB
Task - 3	HandleHealthMonitoring	321KB	16KB
Task - 4	TM_InterruptService	152KB	8KB
Task - 5	HandleHitTrigger	92KB	4KB
Task - 6	HandleAcquisition	466KB	16KB

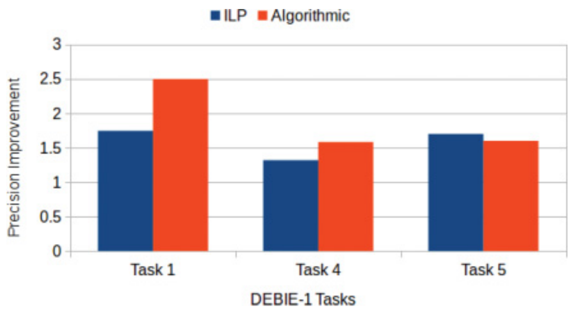

Fig. 8. Graph showing precision improvement (in %) in WCET of DEBIE-1 tasks.

Table V. Breakdown of Precision Improvement

Benchmark	No. of Accesses/BBs That Benefit from Approach of Section			
	4.1	4.2	4.3	4.4
basicmath	3	0	0	6
susan	7	13	1	10
fmref	63	0	0	0
audiobeam	9	2	4	48
DEBIE-1 T1	15	0	0	0
DEBIE-1 T4	10	0	0	0
DEBIE-1 T5	244	0	0	4

tasks, along with the code size and the L1 cache size that we used while performing the experiments.

Out of the six tasks, the proposed approaches show zero or negligible precision improvement for three tasks. The precision improvement of the other three 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 two out of the three 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 the majority of the Mälardalen benchmarks.

Table V shows the breakdown of the precision improvement in terms of the various techniques used by the algorithmic approach (similar to Table III) for the DEBIE-1

Table VI. Analysis Time (in Seconds)

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

tasks and the benchmarks of Section 6.3. Again, all four techniques are successful to varying degrees across benchmarks. Notably, all benchmarks contain accesses that should be classified as AH but are missed by *must* analysis. The majority of the benchmarks also have basic blocks whose worst-case profiles require multiple iterations of their enclosing loop.

Table VI 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 approaches need cache miss paths, the total time taken by the ILP-based approach would be the sum of columns 3 and 4, while the total time taken by the algorithmic approach would be the sum of columns 3 and 5.

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

7. RELATED WORK

AI-based approaches [Alt et al. 1996; Ferdinand and Wilhelm 1999; Huynh et al. 2011; Cullman 2013] 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. These approaches only classify the behavior of individual cache accesses into a small number of classes. However, there are various instances of cache behavior that can be safely used for WCET estimation but that cannot be specified in terms of the individual hit-miss classifications.

Data-Flow Analysis has also been used to perform cache analysis [Mueller 2000]. Conceptually, this approach is similar to AI-based cache analysis, as it defines a data-flow analysis framework to find cache contents that may enter the cache across all executions, or cache contents that 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 [Wilhelm and Wachter 2009; Dalsgaard et al. 2010; Gustavsson et al. 2010; Lv et al. 2011]. 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. Moreover, some of them only consider the impact of the 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 WCEP, most notably, the CSTG-based approach proposed by Li et al. [1996]. However, this approach can potentially introduce an exponentially large number of variables and constraints in the ILP and is considered nonpractical even for small programs [Wilhelm 2004].

Cerny et al. [2013] provide a general framework for finding any quantitative property of a program by representing program execution as a weighted transition system and finding extremal traces in this transition system. They also introduce state-based and segment-based abstractions to reduce the size of the transition system. The state-based abstraction, in particular, has a close resemblance to AI-based *must* cache analysis used for WCET estimation. Chu et al. [2016] use symbolic execution trees to represent program execution. They symbolically execute some program paths with complete precision to obtain their execution time estimates and reuse this information to make the analysis scalable. Their approach is orthogonal to our approach, since it is aimed at finding execution time estimates of entire program paths, while our approach targets WCETs of basic blocks.

Few works [Chattopadhyay and Roychoudhury 2011; Banerjee et al. 2013] have used SAT solvers to directly search for infeasible paths that can affect cache behavior. Chattopadhyay and Roychoudhury [2011] 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. Banerjee et al. [2013] modify 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 (using, for example, symbolic execution, SAT solvers, or model checking).

Andalam et al. [2013] make a tradeoff between the precision of the join in AI-based analysis 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. 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, their approach is specifically targeted toward analysis of direct-mapped caches and does not extend well for set-associative caches. Further, their approach is capable neither of using information about worst-case execution path nor of finding worst-case profiles of basic blocks inside loops.

8. CONCLUSION

In this work, we have proposed a new approach for cache analysis that does not provide simple 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.

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.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques*. Addison Wesley.

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

S. Andalam, R. Sinha, P. Roop, A. Girault, and J. Reineke. 2013. Precise timing analysis for direct-mapped caches. In *Proceedings of the 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC'13)*. 1–10.

Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2013. Precise micro-architectural modeling for WCET analysis via AI+SAT. In *RTAS*.

Pavol Cerny, Thomas A. Henzinger, and Arjun Radhakrishna. 2013. Quantitative abstraction refinement. In *POPL*.

Sudipta Chattopadhyay and Abhik Roychoudhury. 2011. Scalable and precise refinement of cache timing analysis via model checking. In *RTSS*.

Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. 2016. Precise cache timing analysis via symbolic execution. In *RTAS*.

Christoph Cullmann. 2013. Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 1s (2013), 40.

Andreas Engelbredt Dalsgaard, Mads C. Olesen, Martin Toft, Ren Rydhof Hansen, and Kim Guldstrand Larsen. 2010. METAMOC: Modular execution time analysis using model checking. In *Worst-Case Execution Time Analysis*. 113–123. DOI:http://dx.doi.org/10.4230/OASIcs.WCET.2010.113

Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems* 17, 2–3 (Dec. 1999), 131–181.

Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. 2010. Towards WCET analysis of multicore architectures using UPPAAL. In *WCET*.

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

Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudury. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming* 69, 1–3 (2007), 56–67.

Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. 2006. Modeling out-of-order processors for WCET analysis. *Real-Time Systems* 34, 3 (2006), 195–227.

Y.-T. S. Li, Sharad Malik, and Andrew Wolfe. 1995. Efficient microarchitecture modeling and path analysis for real-time software. In *RTSS*.

Y.-T. S. Li, Sharad Malik, and Andrew Wolfe. 1996. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *RTSS*.

Mingsong Lv, Nan Guan, Qingxu Deng, Ge Yu, and Wang Yi. 2011. McAiT A timing analyzer for multicore real-time software. In *Automated Technology for Verification and Analysis*.

Frank Mueller. 2000. Timing analysis for instruction caches. *Real-Time Systems* 18, 2–3 (2000), 217–247.

Kartik Nagar and Y. N. Srikant. 2015. Path sensitive cache analysis using cache miss paths. In *VMCAI*.

Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. 2005. Principles of timing anomalies in superscalar processors. In *Proceedings of the 5th International Conference on Quality Software, 2005 (QSIC'05)*. IEEE, 295–303.

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

Stephan Wilhelm and Björn Wachter. 2009. Symbolic state traversal for WCET analysis. In *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT'09)*. 137–146.

Received June 2016; revised December 2016; accepted January 2017