Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns *

Harini Ramaprasad, Frank Mueller
Dept. of Computer Science, Center for Embedded Systems Research
North Carolina State University
North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu

Abstract

While caches have become invaluable for higher-end architectures due to their ability to hide, in part, the gap between processor speed and memory access times, caches (and particularly data caches) limit the timing predictability for data accesses that may reside in memory or in cache. This is a significant problem for real-time systems.

The objective our work is to provide accurate predictions of data cache behavior of scalar and non-scalar references whose reference patterns are known at compile time. Such knowledge about cache behavior provides the basis for significant improvements in bounding the worst-case execution time (WCET) of real-time programs, particularly for hard-to-analyze data caches.

We exploit the power of the Cache Miss Equations (CME) framework but lift a number of limitations of traditional CME to generalize the analysis to more arbitrary programs. We further devised a transformation, coined "forced" loop fusion, which facilitates the analysis across sequential loops. Our contributions result in exact data cache reference patterns — in contrast to approximate cache miss behavior of prior work. Experimental results indicate improvements on the accuracy of worst-case data cache behavior up to two orders of magnitude over the original approach. In fact, our results closely bound and sometimes even exactly match those obtained by trace-driven simulation for worst-case inputs. The resulting WCET bounds of timing analysis confirm these findings in terms of providing tight bounds. Overall, our contributions lift analytical approaches to predict data cache behavior to a level suitable for efficient static timing analysis and, subsequently, real-time schedulability of tasks with predictable WCET.

1. Introduction

A data cache is an invaluable architectural feature in today's higher-end processors. The savings it provides in terms of memory latency are immense. Hence, data caches have become indispensable. Nonetheless, caching has one inherent complexity, *i.e.*, the latency of data reference becomes unpredictable. While instruction caches are more predictable [13], a memory reference cannot easily be predicted as a hit or a miss in the data cache [17]. This problem is compounded by the fact that references within a program may contend for the same line. Given these problems, precise static characterization of data cache behavior, while being a challenging task, can be extremely useful in making program memory behavior more predictable, specifically for real-time systems.

Several approaches have been and are being proposed to statically model data cache behavior. These approaches share the aim of predicting the behavior of a data cache as accurately as possible given information about memory access patterns. One such approach is the well known Cache Miss Equation (CME) framework proposed by Ghosh *et al.* [6]. This work proposes a method to generate a set of linear Diophantine equations to characterize the behavior of a data cache in loop-oriented code. Solving these equations is a computationally very complex problem and, hence, is generally considered impractical. However, statistics-based approximations and constrained methods that exploit certain properties of these equations have been proposed and are known to reduce the complexity.

These methods produce a slightly pessimistic estimate of the number of misses in loop nests and work only for perfectly nested, rectangular loops. They also impose restrictive assumptions on the code. Array subscript expressions and loop bounds must be affine combinations of the loop induction variables and must be known at compile time. Furthermore, no data-dependent conditionals are allowed in the code. We have used one such implementation of CMEs and have built upon it. Recent work [15] relaxes the assumption that loop nests have to be perfectly nested. Instead, entire programs with arbitrarily nested loops are transformed into sequential loop nests of equal depth. The disadvantage of this approach is that it introduces changes in the representation of data reuse and the analysis of cache behavior.

In our current work, we retain the assumptions about array subscripts and loop bounds. For loop nests, however, we go one step beyond previous work. We transform arbitrarily nested loops via "forced" fusion into a *single* loop nest (as opposed to a sequence of loop nests as described

* This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695.

above) with conditionals that only depend on the loop induction variables.

Using this approach, while still covering entire programs, we use the original representation for reuse to subsequently analyze CMEs. At this stage, we perform more detailed analysis than previous work. This allows us to derive *exact* hit/miss patterns for every reference in the loop nest. This means that, in addition to giving the exact number of misses, we can indicate *where and when exactly* each one of those misses occurs. We further propose a technique to handle non-rectangular loop nests and provide an upper bound on the misses for programs with data-dependent conditionals, thereby increasing the spectrum of programs that can be analyzed.

Our work enhances the performance of static timing analysis frameworks, *i.e*, the process of deriving safe upper bounds on the worst-case execution time (WCET) of tasks. Such an estimation of the WCET is a prerequisite for performing schedulability analysis of real-time applications. The presence of data caches causes unpredictability and, thus, pessimism during these estimations. Our work is directly applicable in this scenario. We feed the exact number of misses as generated by our framework to the timing analyzer and, hence, ensure significantly tighter estimates. Furthermore, our work may be applied in any place where prediction of data cache behavior is required, *e.g.*, in compiler-generated memory layout optimizations, such as tiling and padding, which are beyond the scope of this work.

Experimental results with our framework indicate improvements in the tightness of worst-case cache behavior of one, sometimes even two orders of magnitude over the original CME approach. These results tightly and safely approximate results from trace-driven cache simulation under worst-case input. Subsequent bounds on the WCET by the timing analyzer underline the applicability of these results for end-to-end timing analysis.

The remainder of this document is organized as follows. Section 2 discusses related work. Section 3 gives a brief introduction to static timing analysis, which is an important application for our work. Section 4 gives the necessary background information for the content in this document. Section 5 explains our contributions in the realm of programs that can be analyzed in more detail. Section 6 explains our approach to generating exact data cache reference patterns. In Section 8, we illustrate our method with an example. We discuss the implications that our work has on static timing analysis in Section 9 and show experimental results in Section 10. We then provide a conclusion in Section 11.

2. Related Work

A number of research groups have proposed several methods for making data caches more predictable. While some approaches trade off accuracy for speed, others trade off speed for flexibility and the detail of the output information provided.

Trace based simulators may be used to describe data cache behavior accurately for a given input, but they are very slow and do not provide any information about the cause for the misses for arbitrary inputs.

Several methods that bound data cache behavior have been proposed. Lim *et al.* [10] propose a method that takes data caching into account while computing the WCET for tasks for static memory references. Kim *et al*. [8] propose a method that classifies data references as static or dynamic. However, they do not deal with arrays or pointers.

Data flow analysis is used by Li *et al.* [9] to analyze data cache behavior. White *et al.* [18] propose a method for direct-mapped caches. This work is based on static cache simulation. These methods have high computational complexity due to the explosion of the data-flow state in the presence of arrays. Lundqvist *et al.* [12] present a study which shows to what extent data cache accesses are predictable and conclude that a majority of data cache accesses can be predicted.

Cache locking [11, 4] techniques have been used to make data cache behavior more predictable in real programs. In cache locking [11, 4], selected data is loaded into cache and locked in place so that it may not be replaced until the cache is explicitly unlocked. During the locked interval, since the cache contents are known, cache behavior is predictable. This approach has the disadvantage that locking and unlocking introduce some overheads. Furthermore, if the data is too large to fit into cache, it has to be completely unloaded from cache to make sure cache behavior is still predictable. This leads to performance loss.

Recently, some analytical methods for predicting data cache behavior have been proposed. They include the Cache Miss Equations by Ghosh *et al.* [6], which we have built upon, a probabilistic method of analysis as proposed by Fraguella *et al.* [5] and another analytical method by Chatterjee *et al.* [3]. The basic idea behind all these methods is the same – to characterize data cache behavior by means of a set of mathematical equations. On solving these equations, information about data cache behavior may be obtained. Vera *et al.* have proposed analytical methods based on the Cache Miss Equations to predict data cache behavior [14, 15].

The CME framework [6], in its original form, and the probabilistic methods [5] can be used only for perfect loop-nests with no data dependent conditionals.

The formula-based method [3] is a method that models cache behavior exactly using Presburger formulae to specify cache misses. This method can also deal with multiple loop nests and conditionals. However, it has been applied only for small programs. The applicability in real programs

has not been tested.

Vera et al [14, 15] have built upon the cache miss equations to efficiently produce cache misses in loop-nest-oriented code. Their focus is on analysis speed and, for this, accuracy is traded off to a certain extent. In our work, the main focus is accuracy in order to be able to supply the static timing analysis framework an accurate count of the data cache misses in a program.

3. Static Timing Analysis

Schedulability tests in real-time systems are generally based on the assumption that the WCET of every task in the task set being scheduled is known *a priori*. These estimates need to be a *safe* upper bound on the execution times of tasks. As previous work has demonstrated, dynamic analysis by actual execution of the task does not guarantee worst-case performance [16]. Nor is exhaustive testing of the entire input space practical, as shown in the same study. Hence, static timing analysis is a viable approach to obtain WCET of tasks. Static timing analysis traverses all execution paths in a program and, during this process, calculates a conservative (*i.e.*, safe) upper bound on the time for the longest path in the program.

The structure of a program may cause a hurdle in the path of the analyzer due to factors like data dependent control flow, pointer accesses, etc. Furthermore, *architectural features* also cause unpredictability for a timing analyzer. One such architectural feature, invaluable but, at the same time, particularly hard to model, is the *data cache*. If data cache behavior cannot be predicted sufficiently accurately, WCET estimates may become highly pessimistic. Such predictions may be counter-productive since it may deem task sets infeasible that would otherwise be schedulable.

In this work, we promote an analytical approach to obtain WCET bounds. Figure 1 depicts our framework for static timing analysis to derive WCET bounds. Currently,

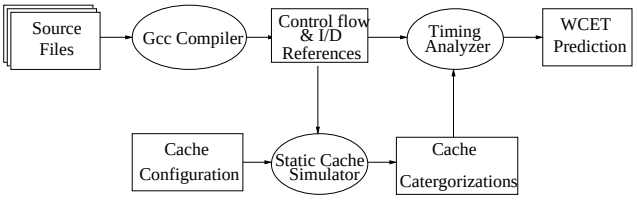


Figure 1. Static Timing Analysis Framework

the framework uses a static cache simulator that simulates the *instruction cache*. While the data-flow methodology utilized by this simulator is adequate for instruction caches, there is no equally effective framework that characterizes data caches. Even though initial work on data cache analysis followed a data-flow methodology as well, constraints on array references and on loop nests were rather restrictive, and analysis overhead was prohibitive for moderately sized caches [17]. The objective of predicting data cache behavior statically remains an area of research with much potential for applications. This motivates the work presented below.

4. CME: Background and Overview

The Cache Miss Equation (CME) framework proposed by Ghosh *et al.* [6] is a method to generate a set of linear Diophantine equations to characterize the behavior of a data cache in loop-oriented code. In the following, a brief overview of CME is given to clearly distinguish prior work from our contribution.

4.1. Terminology

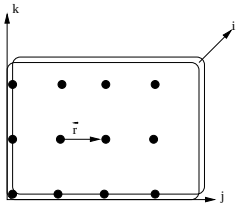
Before describing the details of CME, let us establish a common terminology to reason about data references of non-scalars in loops.

4.1.1. Iteration Space Every iteration of a loop nest is represented as an entity known as an *iteration point*. For example, in a loop nest of depth 3, the iteration where the values of the induction variables are 1, 2 and 3, respectively, for each loop starting from the outermost one, would be represented as the iteration point $\vec{i} = (1, 2, 3)$. The set of all iteration points for a given loop nest is known as its *iteration space*.

4.1.2. Reuse Vectors In order to summarize data reuse among references in loop-nest oriented code, the framework uses the concept of reuse vectors as defined by Wolf and Lam [19]. If a reference accesses the same memory line in two iterations $\vec{i}_1$ and $\vec{i}_2$, where $\vec{i}_2 > \vec{i}_1$, $\vec{r} = \vec{i}_2 - \vec{i}_1$ is called a reuse vector. For example, consider the matrix multiplication code shown in Figure 2(a). Here, the reference

```
for(i = 0; i < N; i++)
  for(k = 0; k < N; k++)
    for(j = 0; j < N; j++)
      C[j][i] += A[k][i] * B[j][k] ;
```

(a) Matrix Multiplication code



(b) Iteration space for matrix multiplication code

Figure 2. Loop Nest and Iteration Space with one Reuse Vector $\vec{r} = (0, 0, 1)$ for $A[k][i]$

$b(j, k)$ has a reuse, which is represented by the reuse vector $(0, 1, 0)$. Reuse vectors are classified into four types — self-temporal reuse, self-spatial reuse, group-spatial reuse and group-temporal reuse. Temporal reuse is a special case of spatial reuse where the two references under consideration access the *same* element in a cache line. Group reuse

occurs when two *different* references access the same memory line.

Consider the matrix multiplication code in Figure 2(a) again. The iteration space for this piece of code is shown in Figure 2(b). An example reuse vector $\vec{r} = (0, 1, 0)$ is shown in the iteration space.

4.2. Cache Miss Equations Overview

CMEs are a set of equations that represent all the *potential* cache misses for references in a loop nest. They relate the iteration space, base addresses of arrays, array sizes and the cache parameters in a precise fashion. They are, in their current form, able to analyze loop nests with no data dependent conditionals. Furthermore, the array subscript expressions and loop bounds are assumed to be affine combinations of the loop induction variables. They also need to be known at compile-time. For every reference and along every reuse vector for that reference, two kinds of CMEs are generated — *cold miss equations* and *replacement miss equations*.

4.2.1. Cold Miss Equations Solutions to cold miss equations represent potential cold or compulsory misses. These are misses that occur on the first access to a memory line. Cold misses may occur in two cases: when the reference under consideration reuses from an iteration point that is *outside* the iteration space and when a reference reuses from data that is mapped to a *different* cache line.

4.2.2. Replacement Miss Equations Solutions to replacement equations account for the remaining misses, namely capacity and conflict misses. For a given reference, replacement equations along a particular reuse vector represent interference with any other reference, including itself (self-conflict). Such misses occur when two references R_A and R_B map to the same cache set, which is also the intuition behind generation of replacement miss equations.

Solving CMEs directly is computationally complex. However, mathematical techniques for manipulating these equations are employed to make the process tractable [1, 14]. Solutions to each CME only represent *potential* cache misses. The effects of multiple CMEs are composed to find the *actual* miss points. A detailed description of the generation of CMEs and the algorithm to compute the actual misses may be found in [6].

4.3. CME Implementation Overview

In our work, we build upon an existing implementation framework for CMEs. This framework, named *Coyote*, is derived from work by Bermudo *et al.* [1]. This framework utilizes the basic reuse vectors as suggested in [19]. In contrast to prior work, our framework extends these reuse vectors to take into account the precise shape of the iteration space. We then consider the impact of statically generating miss patterns for references in the context of bounding the WCET for our static timing analysis framework.

5. Conceptual Enhancements

Our framework provides three fundamental enhancements to the CME framework in the form of relaxing some of its assumptions. This widens the range of programs that we are able to analyze. A detailed block diagram of the enhanced Coyote framework highlighting our contributions is shown in Figure 3. The original CME framework imposes several restrictions on programs that it can analyze. Fundamental among these are as follows. First, the loop bounds must be known at compile-time. Second, array subscript expressions must be affine functions of the loop induction variables. Third, the program can contain only perfectly nested, rectangular loops. Fourth, the program cannot contain data-dependent conditionals.

Recent work [15] relaxes the assumption about perfectly nested loops and allows sequential loop nests of equal depth by transforming arbitrary loop nests. This part of the loop transformation is done in the blocks (d) and (e) of Figure 3. A disadvantage of this scheme is that it leads to changes in representation of reuse and iteration spaces. However, our work goes one step further and uses "forced" loop fusion, represented by block (f) in Figure 3, to get a single loop nest with conditionals, based on loop induction variables, introduced to maintain correctness of the program. In contrast to traditional loop fusion where loop bodies of a common iteration space are combined in one loop of the same iteration space, such as in [20], forced fusion concatenates loop bodies by extending the first iteration space with the second one. Loop bodies are conditionally executed depending on the iteration point in the fused space. The conditionals thus introduced are used to specify whether or nor a certain reference is executed at a certain iteration point. Section 7 generalizes forced loop fusion.

Our work presents a technique to deal with non-rectangular loops in programs. Conceptually, a non-rectangular loop is simply a condition on the upper bound of an inner loop that is based on the current value of an outer loop. We represent a non-rectangular loop accordingly by introducing conditionals dependent solely on loop induction variables. These are then treated exactly in the same way as the conditionals introduced due to forced loop fusion.

Furthermore, we allow programs with data-dependent conditionals where, if the condition is not satisfied, one part of the code is simply skipped (*i.e.*, there is no explicit "else" part to the condition). For such programs, we give an upper bound on the number of misses that the program will incur in the data cache. Most sorting and counting algorithms have such conditionals and, therefore, supporting

them greatly increases the applicability of our analysis!

Since we reuse the original CME framework, shown in block (g) of Figure 3, up to the CME generation stage, the equations are generated assuming that all references in the loop nest are executed at every point in the iteration space. However, we introduce several conditionals during forced loop fusion and in dealing with non-rectangular loop nests. The implications of this are that the reuse vectors generated may be overly optimistic, *i.e.*, could result in unsafe timing predictions that are lower than actual worst-case times. To prevent timing violations and ensure timing safety, an extra analysis step is added to the actual miss calculation stage, which is feasible since conditionals are dependent solely on the loop induction variables. This step is represented by block (h) of Figure 3.

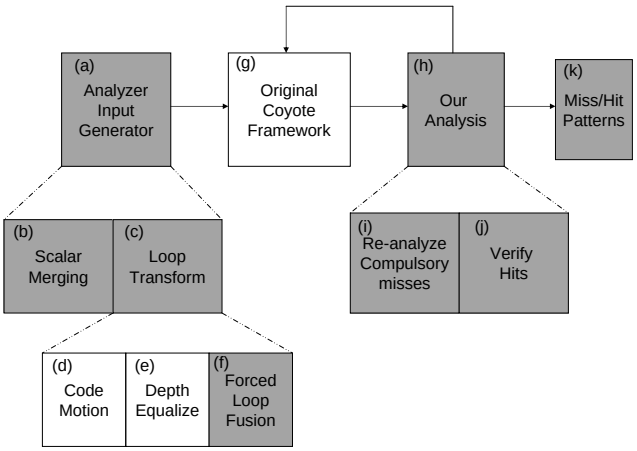


Figure 3. Data Cache Analyzer: Enhanced Coyote Framework

The second problem stems from the layout of array elements in cache lines. In the original CME framework by Ghosh *et al.* [6], all arrays are assumed to be *aligned* in memory lines and, hence, cache lines. This assumption might not always be true — the first element of an array may have a non-zero offset from the start of the cache line. The Coyote framework, a CME implementation, relaxes this assumption and takes *exact base addresses* into consideration during its analysis [1, 14]. However, even Coyote does not take arbitrary reuses into account.

The pitfalls of the CME framework include the following problems: If arrays are not cache-aligned and elements are accessed in non-sequential order, they may have reuses that the original CME framework does not detect. As an example, consider a *two-dimensional* array $a[1..10][1..10]$ that has a *column-major* layout. Consider a data cache that is large enough to hold this array, for the sake of simplicity. Let the cache line size be 32 bytes and the array element size be 4 bytes. Let us assume that the base address of the array causes it to have the mapping shown in Figure 4 when it is brought into cache. Now, consider an iteration space

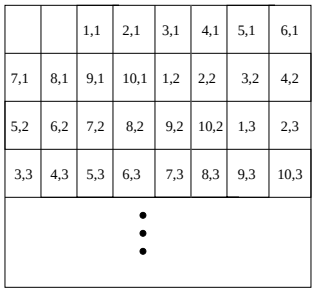


Figure 4. Sample Mapping of 2-D Column-Major Array in Cache

6. Deriving Exact Cache Reference Patterns

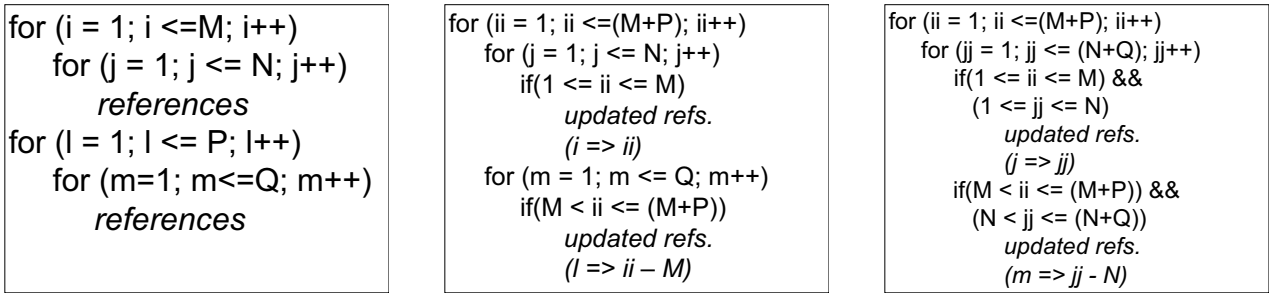
The original CME work occasionally provides imprecise results for certain programs (see below). In this section, we develop a novel approach to overcome this limitation.

6.1. Cause for Pessimism in CME Framework

Each individual CME only represents potential miss points. These iteration points are then analyzed considering the combined effect of all reuse vectors for the reference. This analysis categorizes the iteration points into misses or hits. The CME framework produces slightly pessimistic estimates for the number of misses for each reference in a loop nest. There are two reasons for this. First, the implementation of CMEs (*e.g.*, as provided by Coyote) does not analyze all iteration points due to the complexity involved. Instead, a representative sample of the iteration space is considered for analysis and a *confidence* value is given as a feedback.

of depth two that traverses the array in row-major order. The elements $a[1][1], a[1][2]$ and $a[1][3]$ are correctly categorized by the CME framework as cold misses since it is the first time those memory lines are accessed. Next, elements like $a[3][3]$ are also classified as cold misses since they are on a different memory line than previously accessed data. On similar assumptions, the CME framework also classifies access $a[5][2]$ as a cold miss. However, in reality, since $a[5][2]$ is on the same memory line as $a[1][2]$, it has already been brought into the cache and should actually be classified as a *hit*. Ignoring such reuse leads to pessimism in the miss count.

A third reason for pessimism in the CME framework is that it only captures reuse between uniformly generated references. In contrast, reuse across variables is not captured. While this impacts array references only for layouts where one array ends and another starts in the same cache line, it severely impacts programs frequent references to scalar



(a) Original Loop Nests (b) After Fusing Outermost Levels (c) Final Fused Loop Nest

Figure 5. Example Illustrating Forced Loop Fusion

variables that share cache lines due to their layout.

6.2. Our Generalization

In our work, we stress on deriving *exact* data cache reference patterns. This makes it mandatory for us to consider *all* iteration points while computing actual miss points. This increases the complexity of computation. However, since ours is a *static* approach that *pre-computes* all data cache reference patterns by code analysis, this *one-time* overhead is acceptable. Moreover, the improvement an exact pattern promises in accuracy of static timing analysis is a significant motivation for this approach in spite of the overhead.

The reuse with offsets described in the second problem is not easily captured in terms of reuse vectors. Hence, we take a different approach and *perform further analysis on the iteration points that are classified as compulsory misses by the CME framework*. This is represented by the block (i) in Figure 3. The analysis is as follows. We check if there exists any prior iteration that references an element in the same cache line as that of the reference under consideration and if this reference has not been replaced since. In order to avoid traversing the iteration space to find such iterations, we use a back-tracking approach. We consider each element that would map to the same cache line and simply map them back to the iteration space. This gives us an iteration point that refers to these elements. Afterwards, the check to ensure that this iteration point is *earlier* in the iteration space is a simple process. Since the number of elements that map to any cache line is a constant, the complexity involved in this process is generally affordable.

If a program has many scalars, the first access to each of them is treated as a miss by the original framework. Here, each variable would be considered separately and, hence, reuse vectors do not capture reuse between them. In order to overcome this limitation, we "merge" scalars of equal size that are adjacent in memory and treat them as an array for the purpose of analysis. This is done as a pre-analysis phase as shown in block (b) of Figure 3. Hence, we can capture reuse between different scalars that map to the same cache line by treating them as elements of a single array.

7. Forced Loop Fusion

Forced loop fusion is a technique that we have introduced to concatenate iteration spaces of several loop nests into one loop nest. The algorithm for performing forced loop fusion is described in Figure 6. Consider input loop nests of the form shown in Figure 5(a). We start fusing from the outermost loop and proceed to the inner levels.At every level, we add the number of iterations in every loop at that level and make it one fused loop level. In order to maintain the correct order of memory accesses in the original loops, we introduce one conditional for each reference in the innermost loop which specifies when that reference is to be executed with respect to the recently fused loop level. Further, the subscript of the references corresponding to the loop being altered are modified accordingly. The result after one level of this addition is shown in Figure 5(b). Now, the same process is repeated for the subsequent levels in the loop nests being transformed until we finally have one single loop nest with conditions for execution of the references within the loop nest. The final transformed loop nest in shown in Figure 5(c).

```
n iter = num iters in curr level
lb new = lower bound for new loop at curr level
ub new = upper bound for new loop at curr level
for each loop level in loop nests
    n iter = 0
    for each loop at curr level
        n iter += num iters in curr loop
    lb new = 1
    ub new = n iter
    remove loops at curr level from loop nests
    add new loop at curr level to loop nests with new index
    update subscripts of all references in loop nests as follows:
    for each reference
        for each subscript referring to curr loop level
            replace old index by (new index for level –
            num iters. in orig. loop represented by old index)
```

Figure 6. Forced Loop Fusion Pseudocode

```
for(i = 1; i <=10; i++)
    for(j = 1; j <= 5; j++)
        A[i][j] = 19 ;
    for(k = 1; k <= 10; k++)
        D[i][k] = A[i][k] + 7 ;
for(l = 1; l <= 10; l++)
    for(m = 1; m <= 5; m++)
        D[l][m] = 13 ;
```

(a) Original Loop Nests

```
for(i = 1; i <=20; i++)
    for(j = 1; j <= 20; j++)
        if( (i >=1) && (i <= 10)
            && (j >=0) && (j <= 5) )
            A[i][j] = 19 ;
        if( (i >=1) && (i <= 10)
            && (j >=6) && (j <= 15) )
            D[i][j−5] = A[i][j−5] + 7 ;
        if( (i >=11) && (i <= 20)
            && (j >=16) && (j <= 20) )
            D[i−10][j−15] = 13 ;
```

(b) Transformed Loop Nest

Ref.	Dim Lb and Ub	Base Addr.	Elem. Size
A	1..10, 1..10	151944	4
B	1..10, 1..10	153000	4

(c) Details of Each Variable

Figure 7. Analyzing the Cache Behavior of References in a Loop Nest

Reference	Coyote Output	Output by Our Framework
1	50 misses	MMMMM.....M..... = 6 misses
2	100 misses	.....MMMMM.....M.....M..... = 7 misses
3	100 misses	MMMMMMMMMM.....M...M.....M..... = 13 misses
4	50 misses	..... = 0 misses

Table 1. Original framework vs. pattern by our framework (hits: dots, misses: M)

8. Example of Data Cache Analysis

In this section, we provide a simple example showing the generation of miss/hit patterns. Consider a direct-mapped, 1 KB data cache with a line size of 32 bytes. Since cache patterns are as large as the iteration space, we consider a small example here, which illustrates our approach.

Let us take as an input the loop nests in Figure 7(a). The details for each variable in the code are shown in Table 7(c). First, we pre-process the given loop nests to transform them into a single loop nest through forced loop fusion. Recall that forced fusion concatenates loop bodies by extending the first iteration space with the second one. Loop bodies are conditionally executed depending on the iteration point in the fused space. The resulting loop nest is shown in Figure 7(b). Using this transformed loop nest as input, we generate cache miss equations for this input using the Coyote framework [1]. The miss/hit patterns that we produce as a result are shown in Table 1. We are able to specify how many misses occur for each reference accurately and, in addition, we are able to say exactly where each miss occurs in the iteration space. We also show the results that the original Coyote framework produces in this case, but, since Coyote cannot deal with a loop nest such as the one used in the example, it only knows about the transformed, fused iteration space and has no knowledge of the conditionals we introduce. Hence, its estimates are way off since it assumes all the references are executed at every point in the fused iteration space.

9. Implications to the Static Timing Analyzer

In this section, we briefly discuss the changes imposed on static timing analysis by our novel data cache analyzer. The extended framework is shown in Figure 9. The shaded blocks represent novel as well as enhanced modules in the framework to incorporate the data cache analyzer.

Our data cache analysis framework readily produces the *actual miss/hit patterns*, which indicate the position of each miss in a sequence of references. Instead of exploiting actual miss/hit patterns with complex interleavings, we determined it to be sufficient for the static timing analyzer to obtain an *exact count* of the number of misses. We then cluster all misses at the beginning of a reference pattern (followed by all hits for the remaining references). This pattern

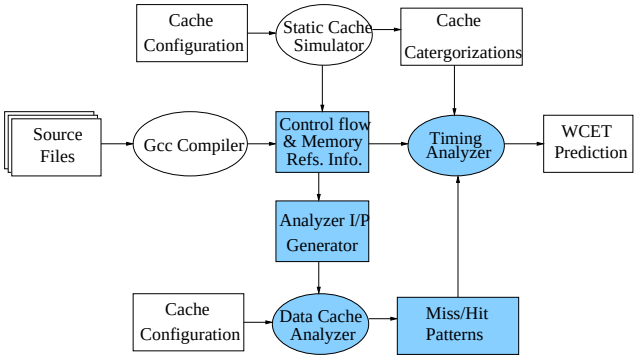


Figure 9. Static Timing Analysis Framework enhanced with Data Cache Analyzer

```
for(i = 1; i <= 10; ++i)
   for(j = 1; j <= 10; ++j)
      A[i][j] = 19 ;
```

(a) Sample Loop Nests

i	Iterations with "Miss time" for j loop	Iterations with "Hit time" for j loop
1	10	0
2	3	7
3..10	0	10

(b) After Fusing Outermost Levels

Figure 8. Miss Pattern Crossing Loop Nests

is fed to the static timing analyzer, which considers the impact of cache hits and misses in the context of pipeline analysis and path traversal to obtain bounds on the worst-case execution time of programs.

To efficiently obtain WCET bounds by static analysis, the analysis approach considers one loop nest at a time starting with the inner-most nest. The times of the longest paths are then repeatedly determined while any cache or processor states are changing till a steady state (fixpoint) is reached, *i.e.*, two consecutive loop iterations result in the same WCET bound. The remaining loop iterations are then guaranteed to be bound by this fixpoint as well [7]. The overall bound for an inner loop can then be used directly in the context of the outer loop in conjunction with adjustments due to changing caching effects between loop nests. This method assumes a consistent pattern for the worst-case cache categorization, even across loops. However, this assumption may not hold for data cache.

Consider n data cache misses for a reference, where n may exceed the upper bound on the number of iterations for an inner loop. Hence, misses extend beyond the iterations of the inner loop. Furthermore, these misses may be scattered over a subset of iterations of the outer loop, not necessarily following any regular pattern, as was observed in the experiments. To handle such misses, the next iteration of the outer loop needs to be considered when finding a fixpoint for the inner loop. This increases the number of iterations of the inner loop that need to be considered before reaching a steady state. We have developed a method that solves this problem with a space and time complexity $O(r+1)$, where r is the number of references in a loop nest.

To illustrate our solution to the above problem, consider the code shown in Figure 8(a). Let us assume that the number of misses predicted by the data cache analyzer, for the sake of demonstration, is 13. We now time the inner "j" loop once considering the reference A[i][j] to be a miss. This is termed the "Miss time" for the loop. Next, we time the inner loop considering the same reference to be a hit. This time is the "Hit time" for the loop. During timing analysis, we propagate information about the inner loop's timing for a certain instance to the outer loop by retaining the "Miss time" and "Hit time" for later accumulation. Table 8(b) shows these values for the example being considered.

This concept, when extended to a loop nest with several references, leads to an algorithm with complexity $O(r+1)$ since we need to consider several permutations of miss/hit status for the references as opposed to just two timings in the above example.

10. Experimental Results

All but two of the programs tested stem from the DSP-Stone benchmark suite [21]. These benchmark programs were modified to replace pointer-based memory accesses with equivalent array accesses to make them statically analyzable. We also inlined functions due to current implementation constraints of our framework that we will lift in the future. These were the only changes. We further included a sorting benchmark, simple-srt-test, from the CLAB benchmark suite in our test set [2]. Lastly, we also constructed a synthetic benchmark to better assess the contributions of our work compared to the original CME/Coyote framework, as explained below. We were unable to use all the benchmarks in the DSPStone suite due to the fact that they have indirect memory accesses, which are currently not analyzable by our framework due to restrictions of the basic CME framework itself.

The first set of experiments compares the result of using the original CME framework and that of our extended framework. Table 2 shows the number of misses and hits produced by the original framework and our framework, respectively. For all except the last benchmark, we assume that arrays are aligned on cache line boundaries for simplicity. For all these benchmarks, we see that there is a mismatch in the total number of accesses (hits+misses) between the original CME framework and our framework. As explained in Section 8, this is due to the fact that the original CME framework cannot analyze these benchmarks as they are. Thus, the benchmarks are transformed as explained in the earlier sections to a form accepted by the original CME framework. However, during this process, we introduce several conditionals based on the loop induction variables, which the CME framework is not aware of and cannot take into account. Hence, it considers the entire fused iteration space with accesses in a unconditional fashion, *i.e.*, disregarding the conditionals. This accounts for the mismatch in the total number of accesses. In fact, Coyote does not catch even a single hit in reality in any except the last

Benchmark used	CME framework		**Our framework**		Simulator	
	Misses	Hits	Misses	Hits	Misses	Hits
convolution	400	400	**26**	374	26	374
dotproduct	8	0	**3**	5	1	7
fi r	599	1192	**26**	573	26	573
lms	1207	9449	**27**	1071	27	1071
matrix1	4600	779400	**39**	4561	38	4562
nrealupdates	1200	2400	**52**	1148	50	1150
simple-srt-test	14	59986	**14**	29686	14	29686
looptest	39	161	**26**	174	26	174

Table 2. Comparison: Orig. CME / Our Framework / Trace-Driven for 4KB Data Cache

two benchmarks shown in Table 2. Hence, for these benchmarks, the very fact that our framework can consider them is an advantage that the original framework does not possess. For the simple-srt-test benchmark, the loop nest is non-rectangular and Coyote does not recognize that. It assumes that the entire rectangular space is traversed.

For the sake of comparison on equal ground, we created a synthetic benchmark with a loop structure that is analyzable by Coyote. This is the last row in Table 2. Here, we see that our framework produces tighter estimates than the original Coyote framework even for programs analyzable by Coyote. This is due to 1) arrays not aligned on cache line boundary and 2) not recognizing adjacent scalars as sharing a cache line (see Section 6).

In order to further verify the safety of our results, we ran a cache simulation for each of the benchmarks using worst-case input. We can see these results also in Table 2. We never under-estimate the worst-case performance of the program being analyzed. Table 3 shows the per-reference breakdown of the same results for one of the benchmarks. The reason for the small disparity between the results of the data cache simulator and our framework is that the CME framework only considers reuse within a variable. Reuse across multiple variables is not considered. As explained in Section 5, we handle this problem in the case of scalars since the disparity would be much more significant there.

srt-test is a sorting benchmark taken from the CLAB suite. This contains data dependent conditionals and also non-rectangular loops. We see from the results that our framework produces an exact bound on the number of cache misses for this case.

The final set of experiments conducted demonstrate the fact that consideration of a data cache for purposes of timing analysis makes a significant difference to the WCET bound produced by the timing analyzer. The results in Table 4 show the worst case execution cycles (WCEC) when data references are considered as 1) always miss, 2) first N misses and 3) cold misses only. The second category, namely *first N misses*, uses the output produced by our data

Reference	Our estimate of misses	Simulator result
1	13	13
2	13	13
3	13	12
4..10	0	0

Table 3. Per-Reference Misses by Our Framework vs. Trace-Driven for matrix1

cache analyzer framework. The third category uses cold miss counts from the trace-driven simulator to verify results.

From the results in Table 4, we can see that considering *every* reference as a miss would severely overestimate the WCET bounds. On the other hand, we see that our estimate provides a tight upper bound on the number of data cache misses, thereby enabling tight WCET bounds. For a cache size of 4KB, which is large enough to fit the entire data set for all benchmarks, our estimate comes very close to the estimate considering only cold misses as provided by the trace-driven simulator. For a smaller cache size which results in additional misses, our estimate is tight.

Benchmark	Always Miss	**First N Misses**		Cold Misses
		1K Cache	4K Cache	
convolution	8791	**5051**	**5051**	5051
dotproduct	530	**480**	**480**	460
fir	12797	**7097**	**7097**	7097
lms	18544	**11814**	**11814**	11814
matrix1	96168	**52378**	**50558**	50548
nrealupdates	23338	**12658**	**11858**	11838
simple-srt-test	668894	**372034**	**372034**	372034
looptest	6482	**4742**	**4742**	4742

Table 4. Timing Analysis for Different Data Cache Categorizations in *Cycles*

11. Conclusion

This work demonstrates the benefits and the potential of Cache Miss Equations in characterizing data cache behavior in a *safe and accurate* manner. The contributions of this work are threefold. First, we provide exact data cache reference patterns for scalar and non-scalar references in loop nest oriented code as opposed to slightly pessimistic miss counts produced in previous work. Second, we apply a transformation we termed “forced” loop fusion to transform any arbitrary loop nests into a single loop nest. While we still consider whole programs with arbitrary loop nests, the transformation brings the loop nests to the form required by the original CME framework and, thus, avoids

major changes to the framework. Also, we handle more general data dependent conditionals and non-rectangular loop nests. Third, we integrate our outputs with the static timing analyzer. This enables us to provide exact miss counts to the analyzer, thus providing the potential to make WCET estimates significantly tighter. Experimental results with our framework indicate improvements in the tightness of worst-case cache behavior of one, sometimes even two orders of magnitude over the original CME approach. These results tightly and safely approximate results from trace-driven cache simulation under worst-case input. Subsequent bounds on the WCET by the timing analyzer underline the applicability of these results for end-to-end timing analysis.

12. Future Work

While our method itself is scalable in terms of cache size, considering larger caches is not useful due to the benchmarks that we analyze. The benchmarks have small data sets, which fit completely into a 4KB cache itself. Hence, using a cache larger than that would not demonstrate the full capabilities by our framework, such as modeling of conflict and capacity misses, not just cold misses. As part of future work, we shall explore larger benchmarks. Further, we intend to test our framework with set-associative caches and explore the applicability of our framework to L2 caches in addition to L1 data caches.

References

[1] N. Bermudo and X. Vera. Coyote project documentation. Technical report, Mlardalen University, 2001.

[2] C-Lab. Wcet benchmarks. Available from http://www.c-lab.de/home/en/download.html.

[3] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–297, June 2001.

[4] D. Decotigny and I. Puaut. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, page 114, dec 2002.

[5] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[7] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):121–148, May 2000.

[8] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Embedded Technology and Applications Symposium*, June 1996.

[9] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, Dec. 1996.

[10] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, Dec. 1994.

[11] B. Lisper and X. Vera. Data cache locking for higher program predictability. In *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, Mar. 06 2003.

[12] T. Lundqvist and P. Stenstrm. Empirical bounds on data caching in high-performance real-time systems. Technical report, Chalmers University of Technology, 1999.

[13] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[14] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior (research note). *Lecture Notes in Computer Science*, 1900:194–198, 2000.

[15] X. Vera and J. Xue. Let's study whole-program cache behavior analytically. In *International Symposium on High Performance Computer Architecture*. IEEE, Feb. 2002.

[16] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[17] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 192–202, June 1997.

[18] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, Nov. 1999.

[19] M. E. Wolf and M. S. Lam. A data locality optimizating algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[20] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[21] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. Dspstone: A dsp-oriented benchmarking methodology. In *Signal Processing Applications and Technology*, 1994.