CHALMERS

Data Cache Timing Analysis with Unknown Data Placement

Thomas Lundqvist

Technical Report 02-11

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Göteborg, February 2002

Data Cache Timing Analysis with Unknown Data Placement

Thomas Lundqvist

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
thomasl@ce.chalmers.se

Abstract

This paper presents a method to estimate the worst-case number of data cache misses for a single subroutine where data objects have an unknown placement in memory. This is motivated by the need for a worst-case execution time estimate for parts of a program, e.g., a single library subroutine. The first step is to identify all data objects that are accessed in a predictable manner but have an unknown placement. Then, a conflict analysis is made and we show how this analysis can be made arbitrarily accurate. Our experimental evaluation of two benchmarks and different cache configurations shows that it is indeed possible to get tight estimates of the worst-case number of misses for a single subroutine with an acceptable analysis effort.

1 Introduction

A common technique in current processor architectures is to improve the average memory access time by using cache memories. However, they make it more difficult to obtain tight estimates of the worst-case execution time (WCET) of programs. This estimate is often needed in real-time systems in order to guarantee timing requirements. Ideally, we want tight estimates of the WCET without sacrificing the average performance. For example, we do not want to turn off the caches.

In this paper, we study the problem of data cache analysis in the context of estimating the WCET for a single subroutine. This is useful when WCET analysis is needed before a program is fully written. For example, the subroutine could be a library function that will be used by several other programs. Furthermore, the analysis of a single subroutine can also be useful to reduce the complexity when analyzing a complete program.

When considering a single subroutine it is a common case that the target address of memory accesses will be unknown due to its dependency on unknown input. Typically, the initial stack pointer and input parameters like pointers to data objects in memory will be unknown. This could easily make all memory accesses unknown, leading to over-estimation of the WCET.

In Sections 3 and 4, we present an extension to a previously published WCET analysis method [7, 8] that makes it possible to reduce the over-estimation of the WCET due to the unknown memory accesses. The method analyzes the interaction of different memory accesses, such as global data accesses, interacting and conflicting with stack accesses. It exploits the fact that although the base address used for a sequence of accesses may depend on unknown input data, the offsets relative to this base address may be known, making it possible to better predict the cache behavior.

Our experimental results, which can be found in Section 5, show that the method in many cases finds tight estimates of the worst-case number of data cache misses. Also, the time complexity of the analysis is found to be reasonably low. The results show that the use of a traditional data cache can in many cases lead to a predictable estimate of the WCET even when analyzing a single subroutine. We relate our method to work by others [2, 4, 6, 10, 11] in Section 6.

```
matmult(matrix A, B, R)
{
  int x, y, z;

  for (x = 0; x < 10; x++)
    for (y = 0; y < 10; y++) {
      R[x][y] = 0;
      for (z = 0; z < 10; z++)
        R[x][y] += A[x][z] * B[z][y];
    }
}
```

Figure 1: Matrix multiplication subroutine

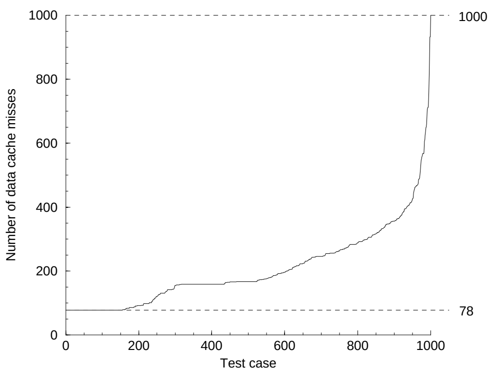

Figure 2: Measured number of cache misses.

2 The Problem

To better understand the problem with conflicting accesses in a data cache, we will start by looking at a small example. In the example and in the rest of this paper we will use the following system model.

For the purpose of this paper, we focus only on the effect of a data cache on the execution time. We treat all memory reads and writes as equivalent. The data cache is assumed to be a traditional cache where placement is managed by hardware. The least recently used block in the set (true LRU) is replaced when a new block is inserted. Furthermore, it is assumed that one or several regions of memory can be marked as being non-cacheable. To simplify the presentation we require that all initial pointer values are aligned with the data cache block boundaries.

Unless otherwise stated, we assume a 2048 bytes, direct-mapped cache with a block size of 16 bytes.

Let us assume we want to find an upper bound on the number of data cache misses occurring when the matrix multiplication subroutine in Figure 1 is run. The upper bound must take into account that the matrices can be placed at arbitrary locations in memory and that the value of the initial stack pointer is unknown. The subroutine multiplies two 10 x 10 matrices, A and B, and puts the result in matrix R. The `matrix` type is defined as a pointer to an array of integers. When compiled (without optimization), the local variables `x`, `y`, and `z` are allocated on the stack. This means that we will have a mix of accesses going either to the stack area or to the different matrices allocated somewhere in memory.

Figure 2 shows the behavior of this subroutine regarding data cache misses. The number of data cache misses has been measured for 1000 random placements of data, and then sorted. The minimum number of data cache misses occurring is 78. This represents the number of cold misses and matches approximately the working set of the subroutine, which is $1200/16 = 75$ blocks for the three matrices. Since the cache capacity is higher than 75 blocks we have no capacity misses and the extra cache misses we see for the majority of all test samples are due to conflicting mapping of data into the data cache. The worst observed combination of input pointer values resulted in 1000 cache misses.

This example shows the importance of testing all input values in order to find the worst case. However, an exhaustive test of all different input pointer values is often not practical. A pointer to a data object determines in which cache set the first block of the data object will be cached. Thus, for block aligned pointers, the number of different cache mappings is equal to the number of cache sets, $numsets$. If a subroutine depends on s unknown pointers, then the number of combinations to test is:

$$combinations = numsets^s$$

In the example, $numsets = 2048/16 = 128$. Thus, four pointers give us $128^4 \approx 268$ million combinations to test. The interesting question is if it is possible to find the worst case without doing exhaustive testing. As we will demonstrate in the following sections this is indeed possible and in Section 5, we find that the actual worst-case for this subroutine is not 1000 but 1252.

3 WCET Analysis Framework

Before we present our new method and how it can estimate the worst-case number of cache misses, we will first describe the WCET analysis framework used. For a more complete description, see [7, 8].

The WCET of a program is the maximum possible execution time for a given range of unknown input data. A general approach to estimate the WCET of a program is to find the worst-case execution time of the longest path in the program. To find the WCET of a single path, we need an accurate timing analysis method, and to avoid examining infeasible (non-executable) paths, we need an accurate path analysis method.

Our previously presented WCET estimation method [7] include both path and timing analysis using cycle-level symbolic execution. An instruction-level simulator, connected to a cycle-accurate timing model, is used to give an accurate timing analysis. Furthermore, the simulator has been extended to also handle unknown values. This makes it possible to distinguish between branch conditions that depend on the input data and branch conditions that are independent of input data, leading to an exploration of all feasible paths in the program and the automatic elimination of many infeasible paths.

In order to reduce the number of paths to explore, we also apply a path merging strategy. Typically, if a loop body contains n feasible paths, the number of paths will grow with a factor of n in each iteration. To cope with this, we merge all paths into one in each iteration before continuing with the next iteration of the loop. The merging of two paths means simply that we pick the longest path and discard the shortest path. To be safe, we add an extra penalty to the estimated execution time to account for the possible mistake we are doing. The short path might actually lead to a longer total execution time in the future. For example, the content of the caches in the short path might lead to a greater number of future cache misses when compared with the long path.

For data cache analysis, the method has been demonstrated to be quite powerful [8]. When doing a WCET estimation, it automatically identifies all accesses that have an unknown reference address. We refer to these accesses as *unpredictable accesses*. These accesses are then connected to the corresponding data structure. Each data structure that is accessed by an unknown data access is marked as being an *unpredictable data structure*. In the next step, these data structures are allocated to a non-cacheable area of the memory, and finally, another WCET estimation can be done to obtain a safe and tight estimate.

In our previous data cache study, data structures with an unknown storage location were always classified as unpredictable. In the next section, we will show how to extend the capability of our method to identify such data structures as predictable despite the storage location being unknown.

4 Approach

In this section, we will first explain how to identify data structures that are predictable but have an unknown placement. This is done in Section 4.1. Second, in Sections 4.2, 4.3, and 4.4, we will explain how to use this information in a data cache analysis without resorting to exhaustive testing of all possible combinations of different placements. Finally, in Section 4.5, we will show how our new data cache analysis handles multiple paths and how it affects the merge operation.

4.1 Data structure identification

The first step is to identify data structures that are accessed in a predictable manner but depends on an unknown base pointer that points to the location in memory where the data structure is allocated.

When analyzing a single subroutine, like the one in the example in Figure 1, we first assign unknown values to all unknown input values at the start of the analysis. Using our simulator tool we then obtain a memory access trace for each path in the program, identifying all accesses going to data structures with unknown placement as unknown. The data structures we are looking for will be classified as unpredictable.

Among the unpredictable data structures we then use a testing approach to identify the ones that can be turned into predictable ones. Our approach is to assign arbitrary values to the input data that corresponds to a base pointer of a data structure. This is done by simply linking the subroutine with a small test program that allocates the data structures to some place in memory and then calls the subroutine. Then, we run our simulator tool again to see if any data structure previously classified as unpredictable gets classified as predictable. These data structures are marked as being *predictable with an unknown placement*.

After the identification is done, all data structures have been classified as either *predictable* if the access pattern is independent of input data, *predictable with an unknown placement* if the access pattern is independent of input but the storage location depends on input data, or *unpredictable* if the access pattern may depend on input data regardless of the storage location being known or unknown.

Finally, to make a WCET estimation, all predictable data structures are allocated to some arbitrary memory location. To make a correct analysis, the normal data cache analysis is enhanced with a conflict analysis to take the unknown placement into account.

4.2 Data cache analysis

After the identification step, the data cache analysis is responsible for analyzing the memory accesses produced by the simulator. Whenever a load or store instruction is executed, we must determine if it hits or misses in the cache. The added complexity in this case is that we want to find out if the access can result in a cache hit regardless of the placement of data structures.

To be able to analyze the interaction of different memory accesses we keep a history of all memory accesses that have occurred. Whenever a new access occurs we add it to the end of the history list and also do a local analysis to determine if it hits or misses in the cache. An example of such a history list can be seen in Table 1. The first column in the table shows the accesses made by the program expressed as offsets from the initial stack pointer or pointer to data structure A. During analysis, fixed values are set to these pointers ($sp = 100000, p_A = 3800$) and column 2 and 3 in the table shows the content of the list of accesses used during analysis. In the list, each access is labeled according to the data structure it targets. The memory accesses belonging to a certain data structure is defined as being part of the same *memory access sequence*.

To decide if an access is a hit or a miss, we first imagine that each access sequence uses its own separate cache. This makes it possible to identify compulsory misses like cold misses or misses due to internal conflicts, i.e., conflicts with other accesses in the same sequence. If an access is not identified as a miss in this step, we continue with an analysis where we take into account possible conflicts from other sequences. Accesses from other sequences are then assumed to be interfering in the most pessimistic manner.

An example of the result of an analysis, for both a direct-mapped and a 2-way set associative cache, can be seen in Table 1. For the first 6 accesses, each access can be classified by only looking at accesses belonging to the same access sequence. Thus, the analysis is similar to a traditional cache analysis. However, for accesses 7,8, and 9, each access can miss due to conflicts with accesses belonging to other sequences. When analyzing access 7, we find that this access must be assumed to be a miss since p_A can be set to a value that makes accesses 4,5, and 6 map to the same cache set as access 7. This is true for both the direct-mapped and the 2-way cache since accesses 4,5, and 6 will replace two blocks in a cache set. Access 8 is analyzed in the same manner and must also be assumed to be a miss. The last access, access 9, is more interesting. Here, the stack pointer can be set to a value that makes accesses 7 and 8 map to the same cache set as access 9. However, the stack accesses can only replace at most one block in a cache set, so we get a possible conflict miss for the direct-mapped cache but a guaranteed hit for the 2-way cache.

An interesting detail in the example concerns access 7 and 8. If we study the example carefully, we actually find that only one of the accesses 7 and 8 will miss. The stack access cannot replace both accesses simultaneously, only one of them. In the next sections, we will first present a basic algorithm that cannot discover this case. Then, we will show how to extend the basic algorithm in order to improve the accuracy of the analysis. Using the extended version of the algorithm we can indeed identify one of the accesses 7 and 8 as a hit.

4.3 Basic algorithm

The basic algorithm used can be seen in Figure 3. It analyzes a new access A with target address r. First, the list containing all previous accesses is searched, starting from the end, to find an access targeting the same sequence and memory block, $memblock$, as access A. If no such access is found, the access is a cold miss.

If $memblock$ is found at a position pos, we must analyze all intermediate memory accesses and see if they can

		History of accesses		Analysis, direct-mapped		Analysis, 2-way assoc.	
	Address	Address	Sequence	Set #	Hit/Miss analysis	Set #	Hit/Miss analysis
1	sp - 109c	fef64	stack	118	Miss (cold)	54	Miss (cold)
2	sp - 10a0	fef60	stack	118	Hit, same block as 1	54	Hit
3	sp - 10a4	fef5c	stack	117	Miss (cold)	53	Miss (cold)
4	p_A + 850	4050	A	5	Miss (cold)	5	Miss (cold)
5	p_A + 50	3850	A	5	Miss (cold)	5	Miss (cold)
6	p_A + 450	4050	A	5	Miss conflict from 5	5	Hit
7	sp - 10a0	fef60	stack	118	Miss if conflict from 4,5,6	54	Miss if conflict
8	sp - 10a4	fef5c	stack	117	Miss if conflict from 4,5,6	53	Miss if conflict
9	p_A + 450	4050	A	5	Miss if conflict from 7,8	5	Hit

Initial stack pointer: $sp = 100000$, initial pointer to data structure A: $p_A = 3800$

Size of data cache: 2048 bytes. Block size: 16 bytes.

Table 1: Example of basic analysis for a direct-mapped and a 2-way cache.

replace $memblock$. In this step, we sort all intermediately accessed memory blocks into different sets according to the sequence and cache set they map to. This sorting can be seen as a kind of backwards cache simulation where each sequence is stored in a separate cache. Starting with the last access, the number of the accessed memory block is put into a set $blocks(q, i)$, where q is the sequence that the access belongs to and i is the cache set it maps to. This is done for all accesses until pos is reached.

To see if $memblock$ can be replaced, the algorithm calculates the maximum possible age, age_{max}, of $memblock$. The age of a block is used by the least-recently used replacement scheme in order to replace the oldest block when a new block needs to be inserted. If age_{max} is greater than the associativity of the cache then $memblock$ may have been replaced. The maximum age is found by counting the number of unique intermediate accesses mapping to the same cache set as access A. For accesses belonging to the same sequence as A, this is easily found as the number of elements in $blocks(seq, cacheset)$, where seq and $cacheset$ are the sequence that A belongs to and the cache set that A maps to, respectively. For other sequences, any cache set can interfere. Therefore, for each other sequence, the maximum number of elements found in any cache set is picked. Finally, if $age_{max} > assoc$, access A is a possible miss, otherwise it is a guaranteed hit.

4.4 Improving accuracy

When the basic algorithm analyzes each access of a sequence seq, it uses the pessimistic simplification that each sequence other than seq causes the maximum possible interference. However, by doing this analysis locally for each access the algorithm derives quite a pessimistic view of the conditions necessary for conflicts to occur.

Ideally, we want to calculate the maximum number of misses among all accesses that can occur considering all possible placements of data structures (an exhaustive algorithm). To express this more formally we first need some definitions. The placement of a data structure is determined by its corresponding initial pointer value. However, the cache behavior is only dependent on where the data is mapped in the cache memory. Therefore, to consider all placements of data we need only consider all offsets, o_i, of a sequence i in the cache. Since all pointers are assumed to be aligned with data cache block boundaries, the offset can be defined as the index of the cache set to which the initial pointer maps, i.e., $o_i \in S$ where $S = [0, numsets-1]$ and $numsets$ is the number of cache sets. Let $\mathbf{o} = (o_1, o_2, \ldots, o_{s-1}) \in S^s$ be a vector of dimension s containing the offsets for all sequences where s is the number of sequences. The set $\mathbf{S} = S^s$ contains all possible combinations of placement for all data structures. We can now express the maximum number of misses in the ideal case as:

```
# $r$ is the target address of the memory access $A$.
# $seq$ is the sequence that the access belongs to.
# $blocksize$ is the cache block size.
# $assoc$ is the cache associativity.
# $size$ is the cache size.
# $numsets = size/blocksize/assoc$ is the
#                 number of cache sets.
# $|set|$ = the number of elements in set $set$.

function $Analyze(A)$
  $memblock = r/blocksize$
  $pos = ListSearchBackwards(memblock, seq)$
  if $memblock$ not found then
    MISS (cold)
  else
    $blocks = SortAccesses(pos)$
    $cacheset = (r/blocksize) \bmod numsets$
    $age_{max} = 1$
    $age_{max} = age_{max} + |blocks(seq, cacheset)|$
    for all sequences $q \neq seq$ do
      $age_{max} = age_{max} + \max_{0 \leq i < numsets} |blocks(q, i)|$
    end for
    if $age_{max} > assoc$ then
      MISS (possible conflict)
    else
      HIT (guaranteed)
    end if
  end if
end function

function $SortAccesses(pos)$
  return $blocks$ where
    $blocks(q, i)$ = set containing all different memory
      blocks from end of history list to position $pos$,
      of sequence $q$ mapping to cache set $i$.
end function
```

Figure 3: Hit/Miss analysis algorithm

$$ideal = \max_{\mathbf{o} \in \mathbf{S}} \left(\sum_{0 \leq A < n_a} miss(A, \mathbf{o}) \right)$$

The sum is made over all accesses A in the program (along a single path, in Section 4.5 we show how multiple paths are handled) and n_a is the total number of accesses. The value of the function $miss(A, \mathbf{o})$ is 1 if access A may miss when data structures are placed according to offsets $\mathbf{o}$, otherwise the value is 0. This means that for each combination of placements we do a traditional cache analysis of all accesses and pick the combination that gives the maximum number of misses.

The basic algorithm, described in the previous section, go through all accesses and does a local analysis for each access. The local analysis includes the examination of all possible data structure placement offsets. It calculates:

$$\begin{aligned} basic &= \sum_{0 \leq A < n_a} \begin{cases} 1 & \text{if } Analyze(A) = \text{MISS} \\ 0 & \text{otherwise} \end{cases} \\ &= \sum_{0 \leq A < n_a} \left(\max_{\mathbf{o} \in \mathbf{S}} miss(A, \mathbf{o}) \right) \end{aligned}$$

The difference is that the $\max$ function has been moved inside the summation, making the analysis more pessimistic. Our approach to get improved accuracy is to do something in between the ideal and the basic algorithm. By splitting the basic analysis into different cases, we can do an exhaustive test of all cases and for each case do a basic analysis using a restricted set of offsets.

Let K be the number of cases to restrict each offset to. Then, for a given case $c_i \in C$, where $C = [0, K-1]$, we restrict offset o_i to:

$$o_i = Kh_i + c_i$$

where: $h_i \in H$, and $H = [0, numsets/K - 1]$. For example, if $K = 2$ we restrict the offsets to even or odd values when setting $c_i = 0$ or $c_i = 1$, respectively. The number of cache sets, $numsets$, must be a multiple of K. Thus, K is typically a power of 2.

We can now formulate an expression for the estimated maximum number of misses that the improved algorithm calculates. First, we introduce the vector notations: $\mathbf{c} = (c_0, c_1, \ldots, c_{s-1}) \in \mathbf{C} = C^s$, and $\mathbf{h} =$

	History of accesses			Possible cache set mappings for restricted offsets			
				$c_{stack} = 0$	$c_{stack} = 0$	$c_{stack} = 1$	$c_{stack} = 1$
	Address	Sequence	Set #	$c_A = 0$	$c_A = 1$	$c_A = 0$	$c_A = 1$
6	4050	A	5	1, 3, 5, ...	0, 2, 4, ...	1, 3, 5, ...	0, 2, 4, ...
7	fef60	stack	118	0, 2, 4, ... **h**	0, 2, 4, ... **m**	1, 3, 5, ... **m**	1, 3, 5, ... **h**
8	fef5c	stack	117	1, 3, 5, ... **m**	1, 3, 5, ... **h**	0, 2, 4, ... **h**	0, 2, 4, ... **m**

m= possible cache miss, **h**= cache hit.

Table 2: Example of improved analysis ($K = 2$) for a direct-mapped cache.

$(h_0, h_1, \ldots, h_{s-1}) \in \mathbf{H} = H^s$. Then, the result from the improved algorithm can be expressed as:

$$\begin{aligned} improved &= \max_{\mathbf{c}\in\mathbf{C}} \sum_{0\le A<n_a} \left(\max_{\mathbf{h}\in\mathbf{H}} miss(A, \mathbf{o}) \right) \\ \mathbf{o} &= K\mathbf{h} + \mathbf{c} \end{aligned}$$

Thus, for a given case $\mathbf{c}$ we do a basic analysis but only include conflicts from other accesses if they occur for the restricted set of offsets. By setting $K = 1$ or $K = numsets$ the improved algorithm reduces to the basic or ideal algorithm, respectively.

In our implementation, we analyze all cases simultaneously by keeping a table, $miss\_table(\mathbf{c})$, that holds the possible number of misses discovered so far in the analysis for each case $\mathbf{c}$. Then, in the end, the result is obtained from:

$$improved = \max_{\mathbf{c}\in\mathbf{C}}(miss\_table(\mathbf{c}))$$

To better understand how the improved algorithm works, we will again study the example in Table 1 and see how accesses 7 and 8 is classified by the improved algorithm. The result from an improved analysis for a direct-mapped cache can be seen in Table 2. Each offset is split into 2 cases ($K = 2$) for a total of 4 cases since we have 2 sequences. Each case is given by the variables $c_{stack}, c_A \in [0, 1]$. As an example, for $c_{stack} = 0$ and $c_A = 1$ we restrict the offsets to:

$$o_{stack} = 2h_{stack} + 0$$

$$o_A = 2h_A + 1$$

Even offsets for the stack pointer means that access 7 maps to even cache sets and access 8 to odd cache sets. Odd offsets for the pointer to data structure A means that access 6 maps to even cache sets. If access 6 may map to the same cache set as accesses 7 or 8 we may get a conflict miss. Thus, when the possible cache set mappings overlap we may get conflict misses. The improved analysis manages, in this case, to find the fact that only one of the accesses 7 and 8 can miss due to conflicts with access 6 and we get at most one miss for these accesses for each case.

4.5 Path analysis and merging

Up until now, we have only dealt with the problem of analyzing memory accesses from a single path in the program. To extend the analysis to multiple paths we must define what happens when paths are split into two and how to handle the merge operation (see Section 3 for a description of our WCET analysis framework).

To handle multiple paths, we make each path contain its own private timing model. This means that each path will have a list of accesses and a table of the number of possible misses, $miss\_table$, unique to that path. When a path is split into two, the list and the table must be duplicated.

When merging two paths, we only keep the list and the table that belongs to the path that we believe will be the longest one. To compensate for the possible mistake we are doing we must compare the two paths and find out if the short path possibly can cause a longer execution time in the future (see [7] for further discussion on this issue). We call this possible execution time difference the *worst case execution time penalty*, $WCET_p$. If $WCET_p > 0$ we must add $WCET_p$ additional data cache misses to the final WCET in order to guarantee a safe estimation.

The penalty $WCET_p$ is found by comparing the list of accesses and the table of misses between the two paths, p_s and p_l, that are being merged. p_s and p_l is the short path and the long path respectively. The $WCET_p$ is given by:

$$\begin{aligned} & WCET_p(p_s, p_l) = \\ & \quad WCET_p(list_s, list_l) + \\ & \quad + WCET_p(miss\_table_s, miss\_table_l) \end{aligned}$$

The definition of $WCET_p$ for the table of misses shown above is simply the maximum difference found between the short and the long path for any case (entry) in the table. For example, assume that for one case the number of possible misses in $miss\_table_s$ is greater than the the number of possible misses for the same case in $miss\_table_l$. At the end of the analysis, this case may prove to be the maximum one and thus represents the final WCET. This means that the final WCET could get underestimated if we do not add the penalty. Thus, we get:

$$\begin{aligned} & WCET_p(miss\_table_s, miss\_table_l) = \\ & \quad \max_{c_0, c_1, \ldots, c_{s-1}} (miss\_table_s(c_0, c_1, \ldots) \\ & \qquad - miss\_table_l(c_0, c_1, \ldots)) \end{aligned}$$

The penalty $WCET_p(list_s, list_l)$ for the list of accesses is calculated by the algorithm in Figure 4. In principle, one cache miss penalty must be added to $WCET_p$ for each possible access in the future that would be classified as a hit according to $list_l$ and a miss according to $list_s$. However, to accurately determine this difference between $list_l$ and $list_s$ is very complex and the algorithm is based on a more simple assumption. To begin with, it is assumed that all accesses in $list_l$ are unique to the long path and do not exist in $list_s$. Then, $WCET_p$ will simply be the number of accesses in $list_l$. This is, however, overly pessimistic. To get a useful algorithm, some additional operations are used. First, each access list is cleaned by removing all accesses that refer to a memory block that is later in the list referred to (superseded accesses). It is only the last access that occurred to a memory block that is used when determining hit or misses.

```
# $list_s$ = list of accesses in short path.
# $list_l$ = list of accesses in long path.
# $list(last)$ = the last access in list
# $access_s = access_l \Rightarrow$ accesses refer to same memory block
#                                 and belongs to same sequence.

**function** $WCET_p(list_s, list_l)$
  remove superseded accesses in $list_l$ and $list_s$

  **while** $list_s(last) = list_l(last)$
    remove last element in both lists
  **end while**

  **return** number of accesses left in $list_l$
**end function**
```

Figure 4: $WCET_p$ algorithm for access lists

Second, if the last element (or elements) in both lists are identical, i.e., they refer to the same memory block and belong to the same sequence, then they can be removed since these accesses can never cause a difference in the future.

It is worth noting that the algorithm presented in Figure 4 is quite pessimistic. In the next section, we evaluate this pessimism experimentally.

5 Experimental results

To evaluate the accuracy and time complexity of the presented algorithms, we have implemented the improved conflict analysis and the merging algorithm into our WCET analysis tool [7]. Then, memory accesses from two different subroutines, *matmult* and *compress*, have been analyzed.

5.1 Experimental setup

The WCET tool used is based on the PowerPC architecture. In this evaluation, the pipeline and instruction cache analysis were turned off. The data cache was set to 2048 bytes and the block size to 16 bytes. All pointers to objects are kept aligned with cache block boundaries.

	matmult	*compress*
	Multiplies two 10x10 matrices	Compresses 50 bytes of data
Identified sequences	4	3
Multiple paths	no	yes
Path merges	0	3965
Executed instructions	47994	46170
Data accesses made	8207	8005

Table 3: Benchmark properties.

The improved algorithm described in Section 4.4 has been implemented. However, an extra optimization has been added. By exploiting symmetry, it is possible to set the offset of one sequence to a fixed value and remove this sequence from the case variables. So if each offset of a sequence gets split into K cases, the total number of cases to examine will be K^{s-1} where s is the number of sequences.

The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the benchmarks. No optimization was enabled. The simulated run-time environment contains no operating system; consequently, we disabled all calls to system functions like disk I/O and the `printf` function in the benchmarks.

Table 3 shows some properties of the benchmarks used. For *compress*, the instruction count and data access count is along the worst-case path found.

5.2 Data structure identification

The subroutine *matmult* has three incoming parameters being pointers to the three matrices R, A, and B, where R is the result matrix, and A and B are the source matrices for the multiplication. These matrices were identified as predictable data structures but with an unknown placement (see Section 4.1). Also, the stack area is treated as a predictable data structure but with an unknown placement since the initial stack pointer is considered to be unknown input data. In total, we identified 4 different memory access sequences.

In *compress*, the incoming parameters are pointers to the source and destination text buffers. The source text buffer was identified as predictable but with an unknown placement. However, the destination buffer was identified as unpredictable. This buffer must therefore be allocated to a non-cacheable memory area. Furthermore, *compress* was found to access the stack area as well as many global variables. Again, the stack area is treated as a predictable data structure with an unknown placement. The global accesses target both predictable and unpredictable data structures. The unpredictable structures were allocated to a non-cacheable memory area, while the predictable data structures were assumed to have an unknown placement since they could get allocated to a different place if the program is relinked. Also, the global variables were treated as one compound object, i.e., the linker is assumed to allocate these variables in the same order regardless of the final placement. In total, we identified 3 different memory access sequences targeting the source text buffer, the stack area, and the global variables.

5.3 Metrics

The worst-case number of data cache misses has been estimated for different data cache associativity (direct-mapped, 2-way, and 4-way), and using different number of cases (see Section 4.4), splitting each offset of a sequence into $K = 1$ case (basic analysis) or $K = 2, 4, 8, 16,$ or 32 cases, giving a total number of cases of K^{s-1} where $s = 4$ for *matmult* and $s = 3$ for *compress*. The observed WCET has been included as a reference value. Ideally, we would have liked to include the actual WCET but this requires an exhaustive analysis that was too expensive to perform. Instead, we used guided testing to find the WCET. This was done by doing a traditional cache analysis for each tested case of initial pointer values. By using the result from the most accurate conflict analysis, we could speed up the testing by focusing on promising ranges of input values.

As a comparison, two more traditional data cache analysis methods have been used, *cache nothing* and *cache a single sequence*. The *cache nothing* method is simply to assume that all data accesses cause a cache miss or to turn off data caching. The *cache a single sequence* method is to only let one sequence be cached. Then, we have used a traditional data cache analysis using an arbitrary placement of the target data structure.

		Matmult		Compress				
		Total misses	Ratio	Cache misses	Ratio	Merge penalty	Accesses not cached	Total misses
Direct-mapped	K=1	4204	3.36	3311	1.16	7	3001	6319
	K=2	3828	3.06	3260	1.14	30	3001	6291
	K=4	2963	2.37	3143	1.10	45	3001	6189
	K=8	2143	1.71	2861	1.00	85	3001	5947
	K=16	1567	1.25	2859	1.00	86	3001	5946
	K=32	1252	1.00	2859	1.00	86	3001	5946
Observed worst		1252		2852		0	3001	5853
2-way set assoc.	K=1	4029	10.38	563	3.61	54	3001	3618
	K=2	2566	6.61	325	2.08	73	3001	3399
	K=4	1299	3.35	189	1.21	86	3001	3276
	K=8	730	1.88	156	1.00	86	3001	3243
	K=16	480	1.24	156	1.00	86	3001	3243
	K=32	389	1.00	156	1.00	86	3001	3243
Observed worst		388		156		0	3001	3157
4-way set assoc.	K=1	78	1.00	19	1.00	86	3001	3106
Observed worst		78		19		0	3001	3020
Cache nothing		8207				0	8005	8005
Cache stack only		4103		6		0	5182	5188

Table 4: Conflict analysis results using improved algorithm with K cases.

5.4 Conflict analysis results

Table 4 shows the results from the WCET analysis of the *matmult* and the *compress* subroutine. The observed worst-case tells us that when using the direct-mapped or the 2-way set associative cache, misses occur due to conflicts between accesses in the different sequences. The 4-way associative cache eliminates all conflict misses and only cold (compulsory) misses are left.

The basic analysis ($K = 1$) over-estimates the worst case for both the direct-mapped and the 2-way cache by a factor ranging from 1.16 (*compress*, direct-mapped) to 10.38 (*matmult*, 2-way). However, for the 4-way cache it finds the exact number of misses. To get tighter estimates for the direct-mapped and the 2-way cache, we need to split the analysis into cases. The algorithm managed to reach within 1 % of the exact worst-case number of misses when using $K = 32$ for *matmult* and $K = 8$ for *compress*. This is interesting since an exhaustive analysis corresponds to $K = 128$ for the direct-mapped cache. Thus, the exact estimate was found with considerable less effort than what an exhaustive analysis would require.

For *compress*, the result from the basic analysis is considerably lower for the 2-way cache when compared with the direct-mapped cache. This is expected since the 2-way cache, compared to the direct-mapped cache, allows the maximum age of a block to be higher before it gets replaced, leading to fewer cache misses. Somewhat surprising, the same is not true for *matmult*. In *matmult*, the basic analysis gives almost the same result for the direct-mapped and the 2-way cache. The reason for this can be found by studying the memory access pattern in *matmult*. The typical data structure access pattern is: $A, B, R, stack, A, B, R, stack, \ldots$. This means that the possible maximum age of a previous access to the same data structure is 4. Thus, for *matmult*, the associativity of the cache must be greater than 3 to avoid counting many accesses as possible conflict misses.

In both subroutines, stack accesses are the most common type of access and have therefore been chosen as the access type to cache for the *cache single sequence* method. The results from this method reveals an inter-

	Matmult		Compress	
	Total #	Analysis	Total #	Analysis
K	cases	time [sec]	cases	time [sec]
1	1	1	1	15
2	8	1	4	15
4	64	2	16	15
8	512	4	64	16
16	4096	25	256	20
32	32768	191	1024	37
Traditional		0.6		3.5

Table 5: WCET analysis times

esting fact. For *compress* and the direct-mapped cache, it is better to only cache stack accesses than to cache all sequences. By caching only stack accesses, the number of conflict misses is reduced more than the increase of the number of misses from the data that is not cached. Furthermore, for *matmult* and the direct-mapped cache, the *cache stack* method performs better than the basic analysis does, showing the need to split the analysis into cases.

5.5 Merge penalty

Table 4 also shows that for *compress*, a merge penalty was needed to guarantee a safe estimate of the total worst-case number of misses. This means that differences in the list of accesses or the $miss\_table$ (see Section 4.5) was large enough to cause a penalty to be added. In our case, the difference is mainly due to the list of accesses and the use of the algorithm in Figure 4. This algorithm is quite simple and pessimistic. Nevertheless, the accuracy of the algorithm is quite sufficient for our study, since the added penalties are quite small.

Another contribution to the penalty is our use of a rather simple timing model. In our study, the execution time of a path is defined as being the number of data cache misses. If we would make the timing model more accurate and add the instruction execution times, the penalty would probably be reduced or eliminated since the execution time differences between paths to be merged would increase and this would in turn reduce the penalty needed.

5.6 Time complexity

An important question is how many cases the analysis can be split into without requiring too much analysis effort. The answer can be found in Table 5 which shows the analysis times for different number of cases and for the two benchmarks. The table also includes the times for doing a traditional data cache analysis (no conflict).

Compared to a traditional analysis, the analysis time needed for a basic conflict analysis ($K = 1$) is a factor of 2 and 4 for *matmult* and *compress*, respectively. When splitting the analysis into cases, the extra analysis time needed is proportional to the number of total cases. As can be seen in the table, for small number of cases the extra analysis time is not noticeable. Therefore, a small number of total cases (for example 64) can always be used without slowing down the algorithm noticeably. Thus, for *compress*, the worst case is found with hardly no extra time needed. This is not true for *matmult* where we must spend a lot of time if we really want to find the worst case.

6 Related Work

Previously published data cache analysis methods [2, 4, 6, 8, 10, 11] cannot analyze the interaction between different sequences of memory accesses. However, it is interesting to see to what extent they still can be used.

Previous methods are only able to handle a single access sequence. In order to handle multiple sequences, the problem must first be reduced in some way. There are several alternatives available and we have already used two methods in Section 5, the *cache nothing* and the *cache a single sequence* methods, that are both applicable. These two methods reduce the problem into analyzing at most a single sequence, which can be handled by all previously presented data cache analysis methods.

If the memory system lacks the possibility of not caching some parts of the memory, we will be forced to cache all data. In this case, the worst-case estimates will always become worse. We must count accesses to non-cached data as misses and if this data also becomes cached we must account for the possible additional conflict misses that may occur.

In this study, we have assumed a traditional cache architecture. However, there are several other cache architectures that could make the problem easier or maybe remove the need for any detailed conflict analysis. For example, hardware cache partitioning [5] could maybe be used to avoid conflicts by caching each data structure into a separate partition. Another way to avoid conflicts is to use a cache where the placement of data is software managed [3]. However, these methods introduce the need to manage the partitions or placement of data. Traditional caches may be easier to use.

Even with traditional data caches it is possible to apply different techniques to avoid conflict misses. For example, software cache partitioning [9] or other conflict avoidance techniques [1] could be used. However, these techniques often require that a complete program is analyzed and produces a fixed placement of data. The algorithm presented in this paper focuses only on a single subroutine. It is not clear how the different techniques can be combined.

7 Conclusions

A new method has been presented to estimate the worst-case number of data cache misses for a single subroutine. In the paper, we present first a basic algorithm and then an improved version that splits the analysis into different cases. An experimental evaluation has been performed using two different benchmark subroutines and using different data cache associativity. When no conflict misses occurred (4-way associative cache), the actual worst-case was found using the fast basic algorithm. On the other hand, when conflict misses occurred (direct mapped and 2-way associative cache) the improved algorithm was needed to reach the actual worst-case number of misses.

Acknowledgments

Professor Per Stenström has contributed with valuable comments on earlier versions of this paper. This research is supported by a grant from Swedish Research Council on Engineering Science under contract number 221-96-214.

References

[1] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceeding of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 139–149, October 1998.

[2] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 16–30, June 1998.

[3] B. Jacob. Hardware/software architectures for real-time caching. Presented at CASES'99: Workshop on Compiler and Architecture Support for Embedded Systems, Washington DC, October 1999.

[4] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 230–240, June 1996.

[5] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 229–237, December 1989.

[6] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[7] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–207, November 1999.

[8] T. Lundqvist and P. Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 255–262, December 1999.

[9] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 137–145, June 1995.

[10] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3nd IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.

[11] F. Wolf and R. Ernst. Data flow based cache prediction using local simulation. In *Proceedings of the IEEE High Level Design Validation and Test Workshop*, pages 155–160, November 2000.