Bounding Worst-Case Instruction Cache Performance*

Robert Arnold, Frank Mueller, David Whalley
Computer Science Dept., Florida State Univ.
Tallahassee, FL 32306-4019
e-mail: whalley@cs.fsu.edu, phone: (904) 644-3506

Marion Harmon
Comp. & Info. Sys. Dept., Florida A&M Univ.
Tallahassee, FL 32307-3101
e-mail: harmon@cis.famu.edu, phone: (904) 599-3042

Abstract

The use of caches poses a difficult tradeoff for architects of real-time systems. While caches provide significant performance advantages, they have also been viewed as inherently unpredictable since the behavior of a cache reference depends upon the history of the previous references. The use of caches will only be suitable for real-time systems if a reasonably tight bound on the performance of programs using cache memory can be predicted. This paper describes an approach for bounding the worst-case instruction cache performance of large code segments. First, a new method called ***Static Cache Simulation*** *is used to analyze a program's control flow to statically categorize the caching behavior of each instruction. A timing analyzer, which uses the categorization information, then estimates the worst-case instruction cache performance for each loop and function in the program.*

1. Introduction

Caches present a dilemma for architects of real-time systems. The use of cache memory in the context of real-time systems introduces a potentially high level of unpredictability. An instruction's execution time can vary greatly depending on if the instruction causes a cache hit or miss. Whether or not a particular reference is in cache depends on the program's previous dynamic behavior. As a result, it has been common practice to simply disable the cache for sections of code where predictability is required [1]. Unfortunately, even the use of other architectural features, such as a prefetch buffer, cannot approach the effectiveness of using a cache. Furthermore, as processor speeds continue to increase faster than the speed of accessing memory, the performance advantage of using cache memory becomes more significant. Thus, the performance penalty for not using cache memory in real-time applications will continue to increase.

Bounding instruction cache performance for real-time applications may be quite beneficial. The use of instruction caches has a greater impact on performance than the use of data caches. In addition, code generated for RISC machines often results in four times more instruction references than data references [2]. There also tends to be greater locality for instruction references than data references, resulting in higher hit ratios for instruction cache performance. Unlike many data references, the address of each instruction remains the same during a program's execution. Thus, instruction caching behavior should be inherently more predictable than data caching behavior.

This paper shows that with certain restrictions it is possible to predict much of the instruction caching behavior of a program. Let a task be the portion of code executed between two scheduling points (context switches) in a system with a non-preemptive scheduling paradigm. When a task starts execution, the cache memory is assumed to be invalidated. During task execution, instructions are brought into cache and often result in many hits and misses that can be predicted statically.

Figure 1 depicts an overview of the approach described in this paper for bounding instruction cache performance of large code segments. Control-flow information, which could have also been obtained by analyzing assembly or object files, is stored as the side effect of the compilation of a file. The control-flow information is passed to a static cache simulator. It constructs the control-flow graph of the program that consists of the call graph and the control flow of each function. The program control-flow graph is then analyzed for a given cache configuration and a categorization of each instruction's potential caching behavior is produced. Next, a timing analyzer uses the instruction caching categorizations along with the control-flow information provided by the compiler to estimate the worst-case instruction caching performance for each loop within the program. A user is then allowed to request the instruction cache performance bounds for any function or loop within the program.

2. Related work

Several tools to predict the execution time of programs have been designed for real-time systems. The analysis has been performed at the level of source code [3], intermediate code [4], and machine code [5]. Only the last tool attempted to estimate the effect of instruction caching

*This work was supported in part by the Office of Naval Research under contract number N00014-94-1-0006.

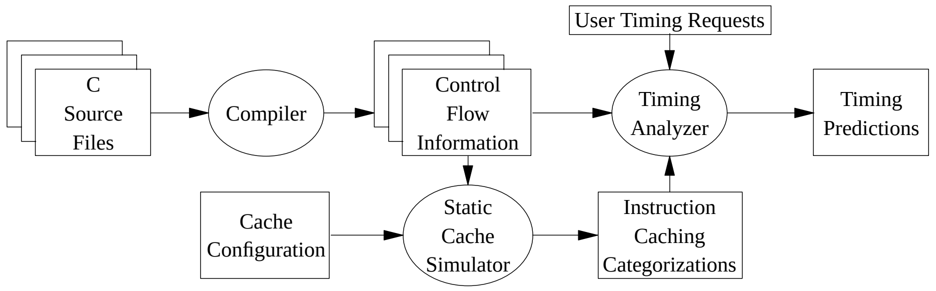


Figure 1: Overview of Bounding Instruction Cache Performance

and was only able to analyze code segments that contained no function calls and fit entirely into cache. Thus, this tool was able to assume that at most one miss will occur for each reference.

Niehaus outlined how the effects of caching on execution time can be estimated [6]. He suggested that caches be flushed on context switches to provide a predictable cache state at the beginning of each task's execution. He provided a rough estimate of the speedup benefit of caches and tried to determine the percentage of instruction cache references that can be predicted as hits. The suggested level of analysis was very abstract since it only recognized spatial locality for sequential execution and some temporal locality for simple loops. No general method to analyze the call graph of a program and the control flow within each function was described.

Lin and Liou suggested that more frequently executed tasks be placed entirely in cache and other tasks be denied any cache access [7]. While this approach may have some slight benefit for a few tasks, the performance of the remaining tasks will be significantly decreased. Part of their rationale was that if a task could not entirely fit in cache, then the worst-case execution would be the same as an uncached system since cache hits could not be guaranteed. It will be shown later that a high percentage of instruction cache hits for such programs can be guaranteed and that the worst-case performance is significantly better than a comparable system with a disabled cache.

There have been attempts to improve the performance and predictability of accessing memory for real-time systems by architectural modifications. For instance, Kirk described a system that relied on the ability to segment cache memory into a number of dedicated partitions, each of which can only be accessed by a dedicated task [8]. But this approach introduced new problems that included lower hit ratios due to the partitioning and an increased complexity of scheduling analysis by introducing another resource (cache partitioning) in the allocation process. Lee *et. al.* suggested to prefetch instructions in the direction that improves the worst-case execution time [9]. The justification for using their approach was that "it is very difficult, if not impossible, to determine the worst-case execution path and, therefore, the worst-case execution time of a task" when instruction caching is employed. Their analysis measured a 45% improvement of the predicted worst-case time as compared to no prefetching (and no instruction cache). This improvement is probably quite optimistic since bus contention was not taken into consideration (contention between instruction prefetching, data access, and thread prefetching). Furthermore, mispredicted branches may result in an uninterruptible block fetch along the wrong path that cannot be aborted. This misprediction penalty may now cause worst-case behavior along the (previously) shorter path. It will be shown later in this paper that much better worst-case performance predictions can be made in the presence of instruction caching than with just a prefetch buffer.

3. Static cache simulation

The method of static cache simulation is used to statically categorize the caching behavior of each instruction for a given program/task with a specific cache configuration. The static simulation consists of three phases. First, a program control-flow graph is constructed. Next, this graph is analyzed to determine the possible program lines that can be in cache at the entry and exit of each basic block within the program. Finally, the control-flow analysis information is used to categorize the caching behavior of each instruction. The following subsections give a brief overview of the static simulator. A more formal approach can be found elsewhere [10], [11].

3.1. Constructing the control-flow graph

Information is obtained from the compiler that describes the control flow for each function within the program. This control-flow information includes the number of instructions in each basic block, the successors of each block, and an identification of the blocks with function calls within the program. From this information a call graph between functions is constructed. In addition, a control-flow graph is constructed for each function, where the nodes are basic blocks and the edges denote control-flow transitions between basic blocks.

To statically estimate the caching behavior of a program as accurately as possible, functions are distinguished by function instances. An instance depends on the calling sequence, that is, it depends on the immediate call site within its caller as well as the caller's call site, etc. The instance *i* of a function corresponds to the *i*th occurrence of the function within a depth-first traversal of the call graph. Thus, a directed acyclic call graph (without recursion) is transformed into a tree of function instances.

3.2. Instruction categorization

The static cache simulator calculates abstract cache states associated with basic blocks. This calculation is performed by a repeated traversal of the call graph's function instances and the basic blocks within each function instance's control-flow graph. The notion of an abstract cache state is a concession from the choice of an exhaustive set of all possible cache states that may occur at execution time and the exponential growth of such an exhaustive set during simulation.

Definition 1: *A program line can* ***potentially*** *be cached if there exists a sequence of transitions in the combined control-flow graphs and call graph (with function instances) such that the program line is cached when the basic block is entered.*

Definition 2: *An* ***abstract cache state*** *of a basic block in a function instance is the subset of all program lines that can potentially be cached prior to the execution of the basic block.*

An obvious goal of the static cache simulation is to determine whether each instruction reference will always result in a cache hit or always result in a cache miss during program execution. An instruction in a basic block within a function instance can be classified as an ***always miss*** if the instruction is guaranteed to never be in cache when it is referenced. An ***always miss*** will occur when the instruction is the first reference to a specific program line in the basic block and the program line is not in the abstract cache state associated with the block. An instruction in a basic block within a function instance can be classified as an ***always hit*** if the instruction is guaranteed to always be in cache when it is referenced. An ***always hit*** is guaranteed when other instructions in the basic block have already referenced the same program line or the program line is in the abstract cache state associated with the block and no other program line that maps to the same cache line is in the abstract cache state.

Unfortunately, some instructions cannot be guaranteed to be always in cache or always not in cache when referenced. The caching behavior of these remaining instructions can be viewed differently depending upon the loop being analyzed. For instance, consider the example in Figure 2. Instruction a is the first instruction that can be executed within the program line x in the outer loop. Instruction b is the first instruction that can be executed within the program line y in the inner loop. Assume program lines x and y are the only two lines that map to cache line c and there are no conditional transfers of control within the two loops. In other words, instructions a and b will always be executed on each iteration of the outer and inner loops, respectively. How should instruction b be classified? With respect to the inner loop, instruction b will not be in cache when referenced on the first iteration, but will be in cache when referenced on the remaining iterations. This situation can be ascertained by the static cache simulator since it can determine that there are no other program lines within the inner loop that conflict with program line y and the abstract cache state at the exit point of the basic block preceding the inner loop does not contain program line y. With respect to the outer loop, instruction b will always cause a miss on each iteration since it will not be in cache as the outer loop initially enters the inner loop.

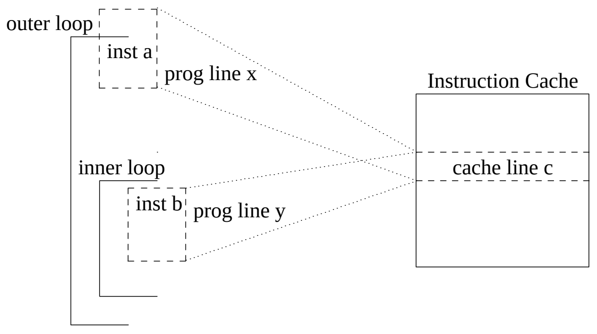


Figure 2: Example of a First Miss and First Hit

The caching behavior of instruction a in Figure 2 can also be predicted, assuming that the instruction that physically precedes the outer loop has to be executed immediately before the loop is entered. In this situation the first reference to instruction a will be a hit. All subsequent references to instruction a will be misses. This situation can be ascertained by the static simulator since it can determine that no other conflicting program lines can be accessed before instruction a is referenced for the first time and program line x will never be in cache on transitions back to the loop header.

The static cache simulator will produce a classification for each loop level in which an instruction is contained. The classifications indicate how references to an instruction should be treated in the worst-case estimation given that an always hit and always miss cannot be guaranteed and the current loop is the most deeply nested loop containing the instruction. These conditions and classifications are depicted in Table 1. A *first miss* simply indicates that the first reference to the instruction should be treated as a cache miss and all remaining references during the execution of the loop should be considered cache hits. Likewise, a *first hit* indicates that the first reference to the instruction will be a hit and all remaining references during the execution of the loop will be misses. When processing an outer loop, the timing analyzer can adjust the value obtained from the timing associated with an inner loop by examining the transitions between classifications from one loop level to the next. These adjustments will be described in the Timing Analysis section.

Other program lines in the loop that map to the same cache line?	The instruction is always executed in the loop and is in cache initially?	In the worst case treat the instruction as:
no	no	first miss
no	yes	always hit
yes	no	always miss
yes	yes	first hit

Table 1: Categorizations for the Remaining Instructions

3.3. Implementation of static cache simulation

The iterative algorithm in Figure 3 was used to calculate the abstract cache states. Each basic block has an input and output state of program lines that can potentially be in cache at that point. Initially, the top block's input state (the entry block of the `main` function) is set to all invalid lines. The input state of a block is calculated by taking the union of the output states of its immediate predecessors. The output state of a block is calculated by taking the union of its input state and the program lines accessed by the block and subtracting the program lines with which the block conflicts. The calculation of these abstract cache states requires a time overhead comparable to that of data-flow analysis used in optimizing compilers and a space overhead of *O(pl * bb * fi)*, where *pl* is the number of program lines, *bb* is the number of basic blocks, and *fi* is the number of function instances. The correctness of iterative data-flow analysis has been discussed elsewhere [12].

```
input_state(top) := all invalid lines
WHILE any change DO
  FOR each basic block instance B DO
    input_state(B) := NULL
    FOR each immed pred P of B DO
      input_state(B) += output_state(P)
    output_state(B) :=
      (input_state(B) + prog_lines(B))
      - conf_lines(B)
```

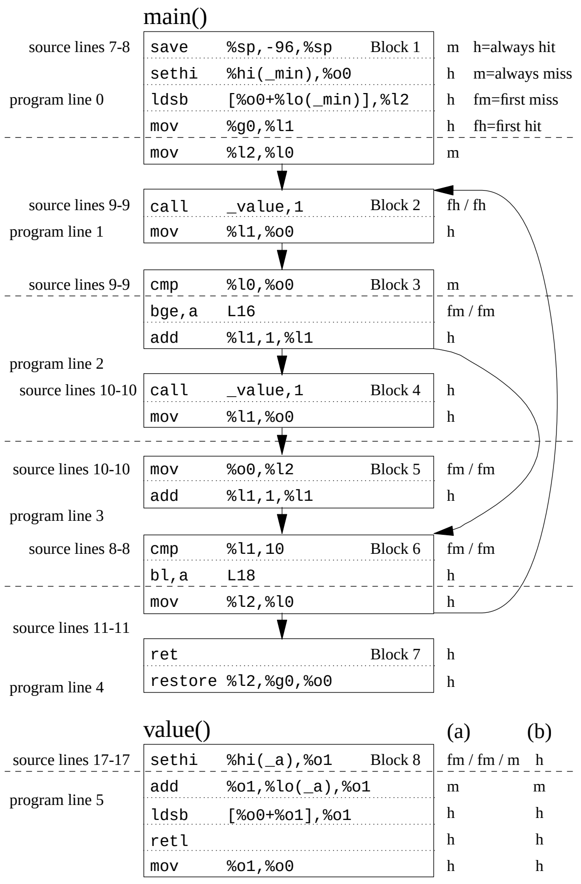
Figure 3: Algorithm to Calculate Cache States

A simple example will be used to illustrate the approach for bounding instruction cache performance. Figure 4 contains C code for a simple toy program that finds the largest value in an array. Figure 5 shows the actual SPARC assembly instructions generated for this program within a control-flow graph of basic blocks. Note the immediate successor of a block with a call is the first block in that instance of the called function. Assume there are 4 cache lines and the line size is 16 bytes (4 SPARC instructions). Block 8a corresponds to the first instance of `value()` called from block 2 and block 8b corresponds to the second instance of `value()` called from block 4. The instruction categorizations are given to the right of each instruction. For instructions that were not categorized as always being a hit or always a miss for each loop level, a categorization of each loop level is given, proceeding left to right from the innermost to the outermost loop. Note that a function is considered a loop with a single iteration. Two passes are required to calculate the input and output states of the blocks, given that the blocks are processed in the order shown in Figure 6. Pass 3 results in no more changes.

```
extern char min,a[10];

main()
{
  int i, high;

  high = min;
  for (i = 0;i < 10;i++)
    if (high < value(i))
      high = value(i);
  return high;
}

int value(index)
int index;
{
  return a[index];
}
```

Figure 4: C Program to Find the Largest Array Value

After determining the input states of all the blocks, each instruction is categorized according to the criteria specified in the previous section. By examining the input states of each block, one can make observations that may not be detected by a naive inspection of only physically contiguous sequences of references. For instance, the static cache simulator determined that the last instruction in block 6 will always be in cache (an always hit) due to spatial locality. It also determined that the first instruction in block 8b will always be in cache (an always hit) due to



```
main()
source lines 7-8    save     %sp,-96,%sp         Block 1   m    h=always hit
                    sethi    %hi(_min),%o0                 h    m=always miss
program line 0      ldsb     [%o0+%lo(_min)],%l2           h    fm=first miss
                    mov      %g0,%l1                       h    fh=first hit
                    mov      %l2,%l0                       m

source lines 9-9    call     _value,1            Block 2   fh / fh
program line 1      mov      %l1,%o0                       h

source lines 9-9    cmp      %l0,%o0             Block 3   m
                    bge,a    L16                           fm / fm
                    add      %l1,1,%l1                     h
program line 2
source lines 10-10  call     _value,1            Block 4   h
                    mov      %l1,%o0                       h

source lines 10-10  mov      %o0,%l2             Block 5   fm / fm
                    add      %l1,1,%l1                     h
program line 3
source lines 8-8    cmp      %l1,10              Block 6   fm / fm
                    bl,a     L18                           h
                    mov      %l2,%l0                       h
source lines 11-11
                    ret                          Block 7   h
program line 4      restore  %l2,%g0,%o0                   h

value()                                                    (a)          (b)
source lines 17-17  sethi    %hi(_a),%o1         Block 8   fm / fm / m  h
program line 5      add      %o1,%lo(_a),%o1               m            m
                    ldsb     [%o0+%o1],%o1                 h            h
                    retl                                   h            h
                    mov      %o1,%o0                       h            h
```

Figure 5: SPARC Insts with Categorizations for Figure 4

```
"I" = invalid
cache   0123012301   cache   0123012301
program IIII012345   program IIII012345
PASS 1
 in(1)=[IIII      ]  out(1)=[  II01    ]
 in(2)=[  II01    ]  out(2)=[  II01    ]
in(8a)=[  II01    ] out(8a)=[  II    45]
 in(3)=[  II    45]  out(3)=[   I 12 4 ]
 in(4)=[   I 12 4 ]  out(4)=[   I 12 4 ]
in(8b)=[   I 12 4 ] out(8b)=[   I  2 45]
 in(5)=[   I  2 45]  out(5)=[      2345]
 in(6)=[   I 12345]  out(6)=[     12345]
 in(7)=[     12345]  out(7)=[     12345]
PASS 2
 in(1)=[IIII      ]  out(1)=[  II01    ]
 in(2)=[  II012345]  out(2)=[  II01234 ]
in(8a)=[  II01234 ] out(8a)=[  II  2345]
 in(3)=[  II  2345]  out(3)=[   I 1234 ]
 in(4)=[   I 1234 ]  out(4)=[   I 1234 ]
in(8b)=[   I 1234 ] out(8b)=[   I  2345]
 in(5)=[   I  2345]  out(5)=[      2345]
 in(6)=[   I 12345]  out(6)=[     12345]
 in(7)=[     12345]  out(7)=[     12345]
```

Figure 6: Calculation of States for Blocks in Figure 5

temporal locality. It detected that the first instruction of block 3 and the second instruction of block 8 will never be in cache (always misses) since the program lines associated with the two instructions map to the same cache line and the execution of block 8 always precedes block 3. The static cache simulator was also able to predict the caching behavior of instructions that could not be classified as always being a hit or always a miss. It determined that the second instruction in block 3 will miss on its first reference and all subsequent references will be hits. Since the first instruction in block 5 and first instruction in block 6 are both classified as first misses and they are in the same program line, then only one miss will occur associated with both instructions during the program execution. Finally, the first instruction in block 2 will always be in cache on its first reference and may or may not be in cache on subsequent references depending on whether the second call to `value()` is executed. Thus, in the worst case the instruction is viewed as a first hit.

The current implementation of the static simulator imposes some restrictions. First, only direct-mapped cache configurations are allowed.[1] Second, recursive programs are not allowed since cycles in the call graph would complicate the generation of unique function instances.[2] Finally, indirect calls are not handled since an explicit call graph must be generated statically.

3.4. Timing analysis

The goal of this research is to allow a user to acquire the most accurate bounds on instruction caching performance of code segments that can be obtained in a reasonable amount of time. After the static cache simulator has produced the instruction categorizations, the user will be queried for a maximum number of iterations for each loop that the compiler could not determine statically. Next, a timing analysis tree is constructed and the worst-case instruction cache performance is estimated for each loop in the tree. Once this initial timing analysis has been completed, the timing analyzer accepts timing requests for either functions or loops.[3]

[1] Recent studies have shown that direct-mapped caches typically have a faster access time for hits, which outweighs the benefit of a higher hit ratio in set-associative organizations for large caches [13].

[2] While cycles in a call graph can be detected, they are also difficult to describe to a user and it is difficult for the user to estimate the maximum number of recursive iterations that will be performed.

[3] Work is currently progressing on processing timing requests for ranges of source lines within a single iteration of a loop.

3.5. Constructing the timing analysis tree

A timing analysis tree is constructed to simplify the process of predicting the worst-case times. Each node within the tree is considered a natural loop.[4] The outer level of each function instance is treated as a loop that will iterate only once when entered.

The timing analyzer next determines the set of possible paths through each loop. A path is a sequence of unique blocks in the loop connected by control-flow transitions. Each path starts with the loop header and is terminated by a block with a backedge or a transition to an exit block outside the loop. Figure 7 shows a simple example that identifies a loop header, backedges, exit blocks, continue paths, and exit paths. Each path is designated as either a continue path (the last block is the head of a backedge transition), an exit path (the last block has a transition to an exit block outside the loop), or both. Thus, each path corresponds to a possible sequence of blocks that could be executed during a single loop iteration. The number of loop iterations indicates the number of times the header of the loop is executed once the loop is entered.

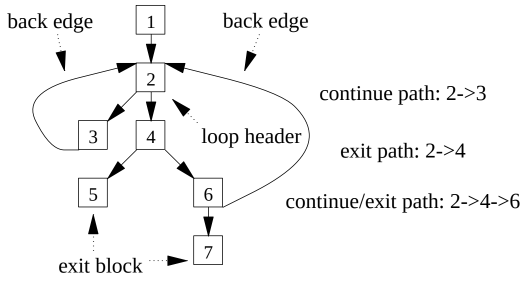


Figure 7: Example Introducing Loop Terminology

If a path within a loop enters a child loop, then the entire child loop is represented as a single block along that path. Associated with each loop is a set of exit blocks, which indicates the possible blocks outside the loop that can be reached from the last block in each exit path. Thus, the possible paths within non-leaf loops that contain child loops can also be calculated.

Figure 8 shows some of the information in the timing analysis tree for the program in Figure 5. Within each loop node the maximum number of iterations is indicated. To the right of each loop node are the possible paths through the loop. Blocks representing a child loop in a path are denoted by having a dashed line boundary. In this example all paths can both continue and exit. The worst-case instruction cache performance is given adjacent to each loop node. The calculation of these results is described in the next section.

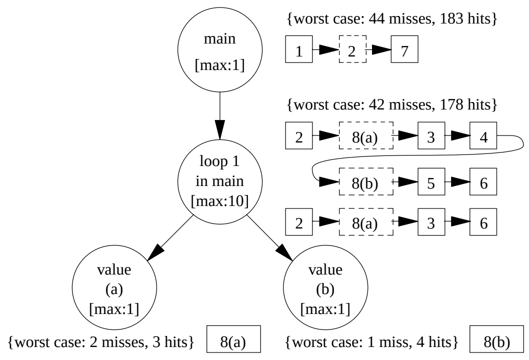

{worst case: 44 misses, 183 hits}
main
[max:1]
1
2
7
{worst case: 42 misses, 178 hits}
2
8(a)
3
4
loop 1
in main
[max:10]
8(b)
5
6
2
8(a)
3
6
value
(a)
[max:1]
value
(b)
[max:1]
{worst case: 2 misses, 3 hits}
8(a)
{worst case: 1 miss, 4 hits}
8(b)

Figure 8: Timing Analysis Tree for Program in Figure 5

3.6. Loop analysis

The loops in the timing analysis tree are processed in a bottom-up manner. In other words, the worst-case time for a loop is not calculated until the times for all of its immediate child loops are known. There will be a worst-case time calculated that corresponds to each exit block. Thus, when the timing analyzer is calculating the worst-case time for a path containing a child loop, it uses the child loop times associated with the exit block of the child loop that is the next block along the path. For instance, the time associated with the loop in Figure 7 exiting to block 5 would be less than the time exiting to block 7 since block 6 would not be executed on the last iteration.

Let n be the maximum number of iterations associated with a loop. The algorithm for estimating the worst-case time for the loop is as follows:

(1) Calculate the maximum time required to execute any continue path assuming that all first misses are counted as hits and first hits are counted as misses. Set the number of calculated iterations to 0.

(2) Go to step 6 if the number of calculated iterations is n - 1.

(3) Calculate the maximum time required to execute any continue path in the current iteration, where each instruction classified as a first miss and not yet encountered is counted as a miss and all first hits are counted as misses.

(4) Go to step 6 if the time calculated in step 3 is equal to the time calculated in step 1.

(5) Add the maximum time calculated in step 3 to the total worst-case time for the loop. If this is the first iteration, subtract the difference between a miss and a hit from the total worst-case time for each first hit in the loop. Denote

[4] A natural loop is a loop with a single entry block. While the static simulator can process unnatural loops, the timing analyzer is restricted to only analyzing natural loops since it would be difficult for both the timing analyzer and the user to determine the set of possible blocks associated with a single iteration in an unnatural loop. It should be noted that unnatural loops occur quite infrequently.

which first misses will now be counted as hits. Add one to the number of calculated iterations. Go to step 2.

(6) Add (n - 1 - number of calculated iterations) * (time from step 1) to the total worst-case time for the loop.

(7) Calculate the times for all exit paths within the loop for the last iteration. For each set of exit paths that have a transition to a unique exit block, add the longest time within that set to the time calculated in step 6 to produce a total worst-case time associated with that exit block for the loop.

The algorithm terminates when the number of calculated iterations reaches n - 1. The algorithm can terminate earlier if the maximum time required to execute any continue path is equal to the maximum time required to execute a continue path where all first misses are treated as hits. In fact, the upper bound on the number of times that step 3 has to be processed is $m+1$, where m is the number of paths in the loop. Each path will have its first misses treated as misses at most once. After all first misses are eliminated, the next maximum path found would be equal to the value calculated in step 1.

The algorithm selects the longest path on each iteration of the loop. In order to demonstrate the correctness of the algorithm, one must show that no other other path for a given iteration of the loop will produce a longer worst-case time than that calculated by the algorithm. The calculation of a worst-case time associated with a path simply requires summing the times associated with each of the instructions in the path. The time used for each instruction depends on whether it is assumed to be a hit or miss, which depends on its categorization. The cache hit time is one cycle on most machines. The cache miss time is the cache hit time plus the miss penalty, which is the time required to access main memory. All categorizations are treated identically on repeated references, except for first misses and first hits. Assuming that the instructions have been categorized correctly for each loop, it remains to be shown that first misses and first hits are interpreted appropriately for a given iteration of the loop.

A first hit implies that the instruction will be a hit on its first reference after the loop is entered and all subsequent references to the instruction during the execution of the loop will be misses. The definition the authors used for a first hit requires that the instruction be within every path of the loop. Thus, the first path chosen for step 3 will encounter every first hit in the loop. After the first iteration, first hits are treated as misses.

A first miss implies that the instruction will be a miss on its first reference after the loop is entered and all subsequent references will be misses. Step 3 indicates that an instruction classified as a first miss will be counted as a miss only the first time it is encountered.

Once the maximum time of the current iteration is equal to the time calculated in step 1 (where all first misses are treated as hits), then this value is replicated for all remaining iterations, except for the last one. Once there are no more first misses encountered for the first time (and the first iteration has encountered all first hits), then the worst-case cache performance for a path will not change since the instructions within a path will always be treated the same. The last iteration is treated separately in step 7. The longest exit path for a loop may be shorter than the longest continue path. By examining the exit paths separately, a tighter estimate can be obtained. Thus, the algorithm estimates a bound that is at least as great as the actual worst-case bound.

The timing of a non-leaf loop is accomplished using this algorithm and the times from its immediate child loops. Whenever a path in a non-leaf loop contains a child loop, then the time associated with that child loop will be used in the calculation of the path time. The transition of a categorization from the child loop level to the current loop level will be used to determine if any adjustment to the the child loop time is required. These transitions between categorizations and appropriate adjustments are given in Table 2. The *fm=>fm* adjustment is necessary since there should be only one miss associated with the instruction and a miss should only occur the first time the child loop is entered.[5] The *m=>fh* adjustment is necessary since the first reference will be a hit.

Child => Parent	Action to Adjust Child Loop Time
fm => fm	Use the child loop time for the first iteration. For all remaining iterations subtract the miss penalty from the child loop time.
fm => m	Use the child loop time directly.
fh => fh	Use the child loop time directly.
m => fh	For the first iteration subtract the miss penalty from the child loop time. For all remaining iterations use the child loop time directly.
m => m	Use the child loop time directly.

Table 2: Use of Child Loop Times

To illustrate the use of the worst-case algorithm, the calculation of the worst-case instruction cache performance for the example shown in Figures 4, 5, 6, and 8

[5] Note that additional work was required when the number of distinct paths containing first misses to different program lines exceeds the number of loop iterations. This situation can commonly occur within functions. A maximum adjustment value was used to compensate in an efficient manner for the remaining loop iterations.

will be described. The worst-case performance results for each loop in the timing analysis tree are shown in Figure 8. Since a loop cannot be timed until its immediate child loops are processed, the two function instances of `value` will be processed first, followed by loop 1 in `main`, and finally the function `main`. For loops with just a single iteration, only step 7 in the worst-case algorithm contributes to the calculated performance of that loop.

The worst-case performance for the example is calculated in the following manner. The leaf loops of the timing analysis tree are the two instances of the function `value` and are processed first. The worst-case instruction cache performances of `value(a)` and `value(b)` are {2 misses, 3 hits} and {1 miss, 4 hits}, respectively. For loop 1 in `main`, step 1 of the algorithm calculates a cache performance of {4 misses, 18 hits} given that all first misses are treated as hits and first hits are treated as misses. This result was obtained from {2 misses, 10 hits} from instructions directly in loop 1 and {1 miss, 4 hits} from both of the invoked function instances of `value`. Note that the time obtained from the first function instance of `value` was adjusted as described in Table 2 (fm => fm). The result found for the first iteration in step 3 is {6 misses, 16 hits}, which was obtained by adding {3 misses, 9 hits} from instructions directly in loop 1, {2 misses, 3 hits} from `value(a)`, and {1 miss, 4 hits} from `value(b)`. The next result calculated in step 3 is equal to the result from step 1. By applying step 6, 8*{4 misses, 18 hits} will be used to represent the performance of the next 8 iterations. Since both paths through the loop are exit paths, the worst-case time for the exit paths calculated in step 7 is the same as the result in step 1. Thus, the total worst-case performance for loop 1 in `main` is {42 misses, 178 hits} ({6+9*4 misses, 16+9*18 hits}). The loop representing the entire function `main` only iterates once and is calculated in step 7. The worst-case instruction cache performance for the entire program is {44 misses, 183 hits}. This result was obtained by {2 misses, 5 hits} from instructions directly in the outer level of `main` and {42 misses, 178 hits} from loop 1 in `main`. The worst-case performance result of loop 1 did not have to be adjusted in the calculation of the performance of the function `main` since the function `main` only iterates once. The implementation of the algorithm calculates the exact worst-case instruction cache performance for this example. This analysis requires a complexity of $O(p*l)$, where p is the number of paths in each loop and l is the number of loops in the timing tree.

3.7. Effectiveness of the timing analyzer

To assess the effectiveness of the timing analyzer, six simple programs were selected. *Des* (Data Encryption Standard) encrypts and decrypts 64 bits. *Matmul* multiples 2 50x50 matrices. *Matsum* determines the sum of the nonnegative values in a 100x100 matrix. *Matcnt* is a variation from *Matsum* since it also counts the number of elements that were summed. *Sort* uses the bubblesort algorithm to sort 500 numbers into ascending order. The final program, *Stats*, calculates the sum, mean, variance, and standard deviation for two arrays of numbers and the linear correlation coefficient between the two arrays.

These programs and the results of evaluating these programs are shown in Tables 3 and 4. For each program a direct-mapped cache configuration containing 8 lines of 16 bytes was used. Thus, the cache contains 128 bytes. The programs were 4 to 17 times larger than the cache as shown in column 2 of Table 3. Column 3 shows that each program was highly modularized to illustrate the handling of timing predictions across functions. Columns 4-7 show the static percentage of each type of instruction categorization in the function instance tree. Each instruction within the tree was weighted equally. If an instruction receives different categorizations for each loop nesting level, then the ratio of the number of instances for a categorization to the number of loop nesting levels for the instruction will be used to calculate the percentage. For example, given that an instruction is classified as "fm/m/m/m" over 4 loop nesting levels, then 0.25 of the instruction is considered a first miss and 0.75 of the instruction is considered an always miss.

Table 4 shows the dynamics results associated with these test programs. Column 2 indicates the hit ratio for each program. Only *Matmul* had a very high hit ratio due to spending most of its cycles in 3 tightly nested loops containing no calls to perform the actual multiplication. Column 3 shows the cycles spent for an execution with worst-case input data. The number of cycles was measured using a traditional cache simulator [14], where a hit

Name	Num Bytes	Num Func	Always Hit	Always Miss	First Miss	First Hit
Des	2,232	5	70.62%	26.76%	1.83%	0.79%
Matmul	788	7	71.15%	24.51%	3.57%	0.77%
Matcnt	800	8	70.64%	25.48%	2.65%	1.22%
Matsum	632	7	69.89%	26.24%	3.87%	0.00%
Sort	536	5	68.18%	27.60%	4.22%	0.00%
Stats	1,488	8	71.76%	24.30%	3.55%	0.39%

Table 3: Static Results for the Test Programs

required one cycle and a miss required ten cycles (a miss penalty of nine cycles). These assumptions were described as realistic by other researchers [13], [2]. Column 4 shows the number of cycles estimated by the timing analyzer. Column 5 shows the ratio of the predicted worst-case instruction cache performance using the timing analyzer in column 5 to the observed worst-case performance in column 3. Column 6 shows a similar ratio assuming a disabled cache. This naive prediction simply determines the maximum number of instructions that could be executed and assumes that each instruction reference requires a memory fetch of ten cycles (miss time).

Name	Hit Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	81.59%	142,079	158,678	1.12	3.88
Matcnt	85.32%	959,064	1,049,064	1.09	4.31
Matmul	99.05%	2,917,887	2,917,887	1.00	9.21
Matsum	87.09%	677,210	677,210	1.00	4.63
Sort	84.05%	7,620,684	15,198,004	1.99	8.18
Stats	88.59%	357,432	357,432	1.00	4.93

Table 4: Dynamic Results for the Test Programs

The example programs illustrate various points. The *Matmul* and *Stats* programs have no conditional statements except to exit loops. The only conditional control statement besides loops in the *Matsum* program was an `if-then` statement to check if an array element was nonnegative. For such programs, predictions for worst-case performance as compared to observed worst-case performance can be estimated very tightly.

The *Matcnt* program not only determines the sum of the nonnegative elements (like the *Matsum* program), but also determines the number of nonnegative and negative elements in the matrix. Thus, there was an `if-then-else` construct used in the code to either add a nonnegative value to a sum and increment a counter for the number of nonnegative elements or just increment a counter for the negative elements. The adding of the nonnegative value to a sum was accomplished in a separate function. This function was placed in a location that would *conflict* with the program line containing the code to increment a counter for the negative elements. Multiple executions of the `then` path, which includes the call to the function to perform the addition, still required more cycles than alternating between the two paths. Yet, the algorithm for estimating the worst-case performance assumed that the first reference to a program line within a path would always be a miss if there were accesses to any other conflicting program lines within the same loop (see Table 1). This assumption simplified the algorithm since the effect of all combinations of paths does not have to be calculated and an exponential time complexity was avoided. Thus, one reference was counted repeatedly as a miss instead of a hit. This path was executed 10,000 times and this accounted for a 90,000 cycle [10,000*miss penalty] or 9% overestimation. Note that the execution of this single path accounted for 43.56% of the total instructions referenced during the execution of the program.

The analysis of the final two programs, *Des* and *Sort*, depicts problems faced by all timing analyzers. The timing analyzer did not accurately determine the worst-case paths in a function within *Des* primarily due to data dependencies. A longer path could not be taken in a function due to a variable's value in an if statement. The *Sort* program contains an inner loop whose number of iterations depends on the counter of an outer loop. At this point the timing tool either automatically receives the maximum loop iterations from the control-flow information produced by the compiler or requests a maximum number of iterations from the user. Yet, the tool would need a sequence of values representing the number of iterations for each invocation of the inner loop. The number of iterations performed was overrepresented on average by a factor of two for this specific loop. This inaccuracy accounted for the overestimation in both the estimated and naive ratios since most of the cycles for the program were produced within this loop. Note that both of these problems have nothing to do with cache predictability.

3.8. Processing user timing requests

Once the timing analyzer has calculated a worst-case time for each loop in the timing analysis tree, the user can request specific timing information about portions of the program. The user first specifies the name of a function. The user is then presented with the set of loops that are within the function. Each loop is identified by its loop nesting level within the function and the source line numbers it spans. The user can choose to obtain a worst-case performance for the entire function or select a loop. Since there may be more than one instance of a function within the timing analysis tree, the timing analyzer will determine the worst-case times from all function instances associated with the user request.

4. Future work

We have designed and partially implemented an algorithm to estimate the best-case instruction cache performance for each loop within a program. A naive best-case estimation, which assumes all instructions along the shortest paths will be hits, will be much closer to the observed best-case performance since locality within programs causes most instruction references to be hits. We expect that the estimated best-case performance can be as tightly predicted as the estimated worst-case performance.

We are exploring methods to predict the timing of other architectural features associated with RISC processors. Work is currently ongoing that uses a micro-analysis technique [5] to predict pipeline performance for the MicroSPARC I. The effect of data caching is also an area that we are pursuing. Unlike instruction caching, many of the addresses of references to data can change during the execution of a program. Thus, obtaining reasonably tight bounds for worst-case and best-case data cache performance is significantly more challenging. However, many of the data references are known. For instance, static or global data references retain the same addresses during the execution of a program. Due to the analysis of a function instance tree (no recursion allowed), addresses of run-time stack references can be statically determined as well. Compiler flow analysis can be used to detect the pattern of many calculated references, such as indexing through an array. While the benefits of using a data cache for real-time systems will probably not be as significant as using an instruction cache, its effect on performance should still be substantial.

5. Conclusions

Predicting the worst-case execution time of a program on a processor that uses cache memory has long been considered an intractable problem [1], [7], [9]. This paper has presented a technique for predicting worst-case instruction cache performance in two steps. First, a static cache simulator analyzes the control flow of a program to statically categorize the caching behavior of each instruction within the program. Second, a timing analyzer uses this instruction categorization information to estimate the worst-case instruction cache performance for each loop in the program. A user is allowed to query the timing analyzer for the worst-case performance of any function or loop within the program.

It has been demonstrated that instruction cache behavior is sufficiently predictable for real-time applications. Thus, instruction caches should be enabled, yielding a speedup of four to nine for the predicted worst case as compared to disabled caches (depending on the hit ratio and miss penalty). This speedup is a considerable improvement over prior work, such as requiring special architectural modifications for prefetching, which only results in a speedup factor of 2 [9]. As processor speeds continue to increase faster than the speed of accessing memory, the performance benefits for using cache memory in real-time systems will only increase.

6. References

[1] D. Simpson, “Real-Time RISCS,” *Systems Integration*, pp. 35-38 (July 1989).

[2] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA (1990).

[3] C. Y. Park, “Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths,” *Real-Time Systems* **5**(1) pp. 31-61 (March 1993).

[4] D. Niehaus, “Program Representation and Translation for Predictable Real-Time Systems,” *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, pp. 53-63 (December 1991).

[5] M. G. Harmon, T. P. Baker, and D. B. Whalley, “A Retargetable Technique for Predicting Execution Time,” *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, pp. 68-77 (December 1992).

[6] D. Niehaus, E. Nahum, and J. A. Stankovic, “Predictable Real-Time Caching in the Spring System,” *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software*, pp. 80-87 (April 1990).

[7] T. H. Lin and W. S. Liou, “Using Cache to Improve Task Scheduling in Hard Real-Time Systems,” *IEEE Workshop on Architecture Support for Real-Time Systems*, pp. 81-85 (December 1991).

[8] D. B. Kirk, “SMART (Strategic Memory Allocation for Real-Time) Cache Design,” *Proceedings of the Tenth IEEE Real-Time Systems Symposium*, pp. 229-237 (December 1989).

[9] M. Lee, S. L. Min, C. Y. Park, Y. H. Bae, H. Shin, and C. S. Kim, “A Dual-mode Instruction Prefetch Scheme for Improved Worst Case and Average Case Program Execution Times,” *Proceedings of the Fourteenth IEEE Real-Time Systems Symposium*, pp. 98-105 (December 1993).

[10] F. Mueller and D. Whalley, “Efficient On-the-fly Analysis of Program Behavior and Static Cache Simulation,” *Static Analysis Symposium*, pp. 101-115 (September 1994).

[11] F. Mueller, *Static Cache Simulation and Its Applications*, PhD Dissertation, Florida State University, Tallahassee, FL (August 1994).

[12] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

[13] M. D. Hill, “A Case for Direct-Mapped Caches,” *IEEE Computer* **21**(11) pp. 25-40 (December 1988).

[14] J. W. Davidson and D. B. Whalley, “A Design Environment for Addressing Architecture and Compiler Interactions,” *Microprocessors and Microsystems* **15**(9) pp. 459-472 (November 1991).