THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

BOUNDING INSTRUCTION CACHE PERFORMANCE

By

ROBERT D. ARNOLD

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 1996

The members of the Committee approve the thesis of Robert D. Arnold defended on August 16, 1996.

David B. Whalley
Professor Directing Thesis

Theodore P. Baker
Committee Member

Susan I. Hruska
Committee Member

Approved:

R. C. Lacher, Chair, Department of Computer Science

CHAPTER 1

INTRODUCTION

Developers of real-time systems face a difficult challenge in developing software. For a real-time system to function correctly, the system must not only be logically correct, but must also adhere to specified timing constraints expressed in terms of hard and soft deadlines. Violation of these timing constraints can result in system performance degradation or complete system failure. A significant amount of research has been expended into various static scheduling algorithms such as Rate Monotonic Analysis [1]. Much less effort has been spent in predicting execution times of individual tasks. However, in order to apply these static scheduling algorithms, the worst-case execution time (WCET) must be known for each scheduled task. Some scheduling algorithms may require knowing the best-case execution time (BCET) as well.

Measurement of task execution times has been very difficult due to the possible variance in execution times for each instruction, must less each task. A significant source in the execution time variance is due to caching effects. Whether or not an instruction is in cache each time it is referenced depends upon the control-flow used to reach the reference to that instruction. The difference between a cache hit or miss can result in an order of magnitude difference in execution time of an instruction [2].

Current techniques for detecting timing constraint violations usually involve some type of dynamic measurement, such as direct measurement. The bounded execution time results from dynamic measurements are questionable since it is quite difficult to determine the input data for a real-time application of any significant complexity that will cause the WCET to be executed. With instruction caching it is even more difficult. To illustrate this concept consider the code in Figure 1. Assume that the paths through statement 1 and statement 2 require approximately the same time when they are located in the cache and both statement 1 and statement 2 are in different memory lines, but map to the same cache line. The worst-case input data would require that statement 1 and statement 2 be alternately executed. This would not be obvious from an examination of the source code. Since many inputs are time-dependent, it may also be difficult to consistently generate an input at the required time to drive a specific control-flow path. The technique described in this thesis eliminates this problem by automatically considering all possible control-flow paths.

```
for  (i=0; i < 1000; i++)
   if (condition)
      statement 1;
   else
      statement 2;
```

Figure 1: Example for Conflicting Cache Lines

Using direct measurements is also considered inadequate since it requires a functional software system on an existing hardware platform. Hence, most testing for timing constraint violations is done during the final phase of system testing. Resolving timing constraint violations at such a late stage in the development life-cycle significantly increases the effort to resolve the violation.

In order to reduce this variance, and its associated problems, a common solution is to disable all caching [3]. This extreme solution is successful in reducing the execution time variance. However, it reintroduces the problem solved by caching, which is a dramatic decline in performance. With the continual increase in cache sizes and the ratio between main memory and cache access times, disabling caches results in continually increasing sacrifice of system performance improvements. Recent hardware development has shown a CPU performance increase versus memory access time exceeding 10:1 per year [2]. This implies that at some point caching will be needed to provide the performance for the required system functionality. This research describes a technique to statically determine the BCET/WCET during compilation. This technique exploits a concept of *bounding* the execution times with the BCET/WCET rather than seeking the average execution time, which significantly reduces the complexity. This *static* approach does not require knowing the input data to drive the best/worst case paths, negates the difficulties in testing a specific path, and identifies timing constraint violations much earlier in the development process. This technique considers a non-preemptive system containing only an instruction cache. Due to the additional complexity, other architectural features were eliminated from consideration.

Figure 2 shows an overview of the timing analysis process. This technique is implemented by using an optimizing compiler called *vpo* [4], which produces control-flow information as a side-effect of compiling one or more *C* source code files. This control-flow information is analyzed by the static cache simulator to produce a control-flow graph consisting of the call graph and the control flow of each function. Using the specified cache configuration, the control-flow graph is analyzed to produce a categorization for each instruction's potential caching behavior. Each *instruction cache* *category* will describe the caching behavior of the instruction at different loop levels of the program execution. Details of the static cache simulator can be found elsewhere [5], [6], [7], [8]. The timing analysis technique described in this thesis uses these instruction caching categorizations along with the control-flow information from the compiler to estimate the best and worst-case instruction caching performance for each loop and function contained in the program. Once the timing analyzer has evaluated all functions within the program, a user-interface [9], [10] is invoked to allow the user to request timing bounds for specific code segments within the program.

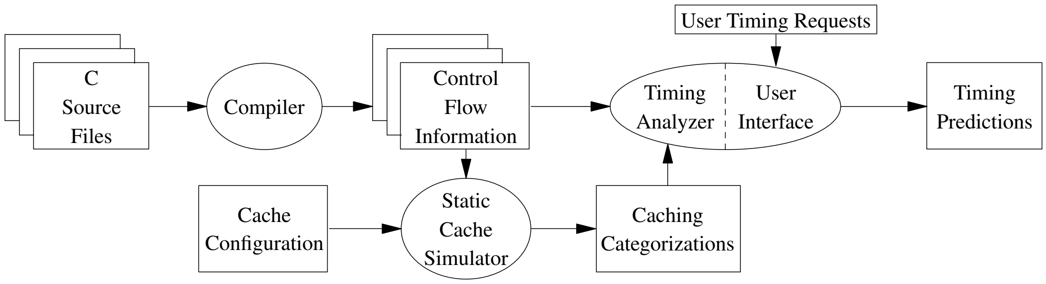

Figure 2: Overview of Bounding Instruction Cache Performance

CHAPTER 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 [11], intermediate code [12], and machine code [13]. Only the last tool attempted to estimate the effect of instruction caching 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 instruction reference.

Mueller [5] describes an approach called *static-cache simulation* that categorizes the caching behavior of each instruction into one of four categories, thereby allowing many instruction references to be treated as cache hits. The path analysis technique described in this thesis uses the categories produced by the static simulation to bound the best and worst-case execution times.

Lim *et. al.* [14] describe a method using a timing schema associated with each source-level language program construct. By executing a single bottom-up pass, each construct is analyzed in the context from which the construct is called. This approach works best when timing the entire program segment. Since the final execution time of a segment is not known until the surrounding context is considered, timing requests for a particular segment must assume a pessimistic worst-case bound. Additional pessimism is introduced from this technique's inability to analyze optimized code. Compiler optimizations can significantly reduce execution times. Also, the authors never demonstrated if their method is capable of recognizing spatial locality[1].

Li *et. al.* [15] uses an integer linear programming (ILP) technique to model instruction cache behavior. The authors automatically derive many constraints from a program's control-flow graph that can be solved using ILP. The user is required to express constraints regarding data dependencies within the control-flow and the number of iterations for each loop in the program. Each basic block is analyzed to determine the sets of instructions that mapped to the same cache line. Each set is referenced as a *line-block*. Three possible possible states are identified for each cache line. First, if only one line-block is mapped to it, then it will experience at most one miss penalty. Second, if two or more non-conflicting line-blocks map to a cache line, then these line-blocks will have at most one miss penalty among them. Finally, if two or more conflicting line-blocks map to it, then a cache conflict graph is constructed for this cache line. The edges between the line-blocks in this graph represent a possible path between the two conflicting line-blocks. Additional constraints are generated to represent the number of times these edges are traversed. Whenever a line-block is reached from a conflicting line-block, it is assumed that there is a miss penalty associated with its execution. A drawback of this approach is the exponential growth relationship between the program size and the number of ILP equations to be solved. Results from this research [15] indicate a significant increase in execution time required to perform a timing analysis

[1] The principle of locality holds that all programs favor a portion of their address space at any instant of time. Spatial locality implies that if an item is referenced, nearby items will tend to be referenced soon [2].

over the technique presented in this thesis.

CHAPTER 3

TIMING ANALYSIS

A timing analysis tree is constructed to simplify the process of determining the execution bounds of a program. The result of this analysis will be a tree structure in which each node of the tree represents a natural loop in the program[2]. In order to process loops and functions in a similar manner, each function is considered a loop that will iterate one time.

The creation of the timing tree requires the analysis of the program's code in order to determine information regarding the loops within each function. The optimizing compiler initiates this analysis by identifying for each loop: the nesting level, all the blocks contained within the loop, all exit blocks from the loop, the minimum number of loop iterations, and the maximum number of loop iterations. The timing tool extends this analysis by determining all possible paths through the loop.

As shown in Figure 3, a *path* is a sequence of unique blocks in the loop connected by control-flow transitions. A *basic block* is defined as a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the

[2] A natural loop is a loop with a single entry block. The timing analyzer is restricted to only analyzing natural loops since it would be difficult 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.

end without halt or possibility of branching except at the end [16]. A *loop header block* is defined as the unique entry block into the loop. Blocks outside the loop that are reached by control-flow transitions from blocks within the loop are defined as *loop exit blocks*. Each path in the loop must start with the loop header block and terminate with a block containing a transition to the header block (continue path) or to an exit block (exit path.) The path through a function is defined to start with the entry block in the function and end with the block containing a return instruction. If a path within a loop contains a nested loop, then the entire nested loop is represented in the path by only the header block of the nested loop. Associated with each loop is the set of exit blocks for that specific loop.

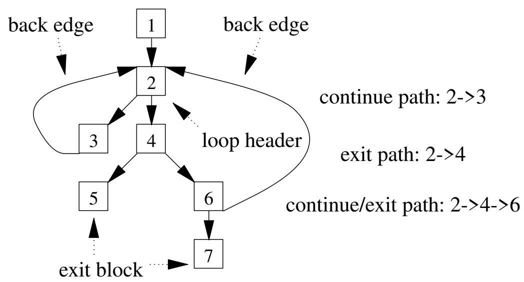

Figure 3: Example Introducing Loop Terminology

If the example given in Figure 3 is considered a function with one nested loop, the algorithm to determine all possible paths, depicted in Figure 4, will first identify all possible paths through the nested loop before determining all possible paths through the function. Figures 4 and 5 depict the algorithms to identify all possible paths, including nested loops, within a loop.

Figure 6 illustrates the path analysis algorithm in Figure 4 by depicting the creation of paths on each iteration of the algorithm while analyzing the simple loop

(1) Set a pointer at the current loop node structure. This structure will be used to reference the list of all possible paths through this loop.

(2) Move to the end of the list of blocks in the current path.

(3) Determine the list of successor blocks to the current basic block.

  (A) If the current block is not a loop header block then the list will contain all successor blocks to the current block.

  (B) If the current block is a loop header block to a *nested loop* then the list will contain all exit blocks out of this nested loop.

(4) If the current block contains an outgoing transition that represents a back edge to the current loop:

  (A) If multiple successor blocks exist with one block being a loop exit block and a second block being the loop header block then the current path represents both a *continue* path and an *exit* path. If the successor list contains additional blocks then more branches exist.

    (a) If additional branches exist then copy the current path to a new path, set the new path's type *CONTINUE/EXIT* and move the loop header block and loop exit block from the successor list to the new path's exit list. Insert the new path into the list of all paths through this loop.

    (b) If additional branches do not exist then set the type this path to *CONTINUE/EXIT* and move the loop header block and loop exit block from the successor list to this path's exit list.

  (B) If multiple successor blocks exist with the loop header block and no loop exit blocks present then the current path contains control-flow branches with one representing a *back edge*.

    (a) Copy the current path to the new path and move the loop header block from the successor list to the new path's exit list.

    (b) Set the type of the new path to *CONTINUE*.

    (c) Insert the new path into the list of all possible paths through this loop.

  (C) If only the loop header block exists in the successor list then the current path is complete.

    (a) Move the loop header block to the exit list.

    (b) Set the type of the current path to *CONTINUE*.

    (c) Go to step #8.

(5) If the current block represents an exit out of the loop:

  (A) If multiple successor blocks exist then the current path contains a branch at this point. Copy the current path to a new path and move the loop exit block to the exit list of the new path.

    (a) Set the type of the new path to *EXIT*.

    (b) Insert the new path into the list of all possible paths through this loop.

  (B) A successor list containing only one block which represents a loop exit block can only occur for a path through a function. Note that this occurs because the loop path through a function does not contain a *back edge* path. Set the type of this path to *EXIT*.

(6) While multiple blocks exist in the successor list branches exist in the control-flow.

  (A) Copy the current path to a new path.

  (B) Append the successor block to the new path's list of basic blocks through the loop.

  (C) Insert the new path into the list of all possible paths through this loop.

(7) If the successor list has a block in it then append this successor block to the end of the list of blocks in this current path.

(8) Go the next incomplete path in the list of all possible paths through this loop. If an incomplete path is found then go back to step #2.

(9) If no incomplete paths are found then exit.

Figure 4: Algorithm to Determine All Paths Through a Loop

(I) Find the maximum loop nesting level for the function being analyzed.

(II) From the maximum nesting level down to 0 find each loop with a nesting level equal to the current nesting level and do the following:

  (1) Create a structure (Loop Node) to hold general information about the loop (number, header block, nesting level, min and max iterations) and a pointer to all the possible paths through the loop.

  (2) Making the assumption that at least one path must exit through the loop, create a structure (loop_path_node) that will list information about one specific path through the loop. This specific information will include:

    (A) list of basic blocks to traverse through the loop

    (B) list of exit blocks out of the loop

  (3) Determine all Possible Paths Through the Loop

Figure 5: Algorithm to Find all Paths Through All Loops

given in Figure 3. Step 1 in Figure 6 shows the identification of the first block in the nested loop path. In analyzing the multiple transitions leaving block #2, the algorithm recognizes the branch in control-flow. As shown in step 2 the current path is duplicated and block #3 appended to the new path. In step 3 the algorithm is extending the current path by appending block #4. Block #4 also has two transitions leaving it. The first transition exits the loop to block #5. The second transition is to block #6 which is still in the loop. In step 4 the algorithm will recognize the exit to block #5 so it will duplicate the current path and append the loop exit block, block #5, to that loop path's exit list. In step 5 the algorithm is following the current path by appending block #6 onto the path. Two transitions leave block #6, the first is an exit to block #7, the second is a back edge transition to the loop header. The algorithm, in step 6 recognizes this path as a *CONTINUE/EXIT* path so it labels the path as such and moves the loop header block and loop exit block to the loop path's exit list. In step 7 the algorithm completes the remaining incomplete path from block #2 to block #3. Since block #3 has only a

(1) First Iteration Through Algorithm, Steps 1 and 2

2	<== Current Path

(2) First Iteration Through Algorithm, Steps 3 and 6

2	<== Current Path
2->3	<== New Path

(3) First Iteration Through Algorithm, Step 7

2->4	<== Current Path
2->3	<== New Path

(4) Second Iteration Through Algorithm, Step 5

2->4	<== Current Path
2->4	<== Completed EXIT Path with Exit Block #5
2->3	<== Incomplete Path

(5) Second Iteration Through Algorithm, Step 7

2->4->6	<== Current Path with Block #6 Appended
2->4	<== Completed EXIT Path with Exit Block #5
2->3	<== Incomplete Path

(6) Third Iteration Through Algorithm, Step 4

2->4->6	<== Completed CONTINUE/EXIT Path with Exit Block #7
2->4	<== Completed EXIT Path with Exit Block #5
2->3	<== Incomplete Path

(7) Fourth Iteration Through Algorithm, Step 4

2->4->6	<== Completed CONTINUE/EXIT Path with Exit Block #7
2->4	<== Completed EXIT Path with Exit Block #5
2->3	<== Completed CONTINUE Path

Figure 6: Example of the Nested Loop Path Analysis Sequence
(Reference the Path Analysis Algorithm in Figure 4)

transition to the loop header block, this path is marked as a *CONTINUE* path.

Now that all the paths through the nested loop have been identified, the algorithm begins identifying all paths through the function. Reference Figure 7 for the path analysis sequence. In step 1 the algorithm has identified block #1 as the function start block. In step 2 the algorithm has recognized block #2 as the loop header block for the nested loop and represents it by appending block #2 to the current path. In step 3 the algorithm has determined all loop exit blocks from the nested loop so it duplicates the current path and appends block #5 to it. The current path is extended by appending block #7 to the end of it in step 4. The algorithm then recognizes recognizes block #7 as the exit block from the function so it labels the current path as an *EXIT* path in step 5. The last step is to consider the last incomplete loop path. In step 6 this path is also recognized as an *EXIT* path and labeled as such.

```
(1) First Iteration Through Algorithm, Steps 1 and 2
    1
(2) First Iteration Through Algorithm, Steps 3 and 7
    1->2          <== Current Path
(3) Second Iteration Through Algorithm, Steps 3 and 6
    1->2          <== Current Path
    1->2->5       <== New Path
(4) Second Iteration Through Algorithm, Step 7
    1->2->7       <== Current Path Extended By Appending Block #7
    1->2->5       <== Incomplete Path
(5) Third Iteration Through Algorithm, Step 5
    1->2->7       <== Completed EXIT Path with Exit Block #7
    1->2->5       <== Incomplete Path
(6) Fourth Iteration Through Algorithm, Step 5
    1->2->7       <== Completed EXIT Path with Exit Block #7
    1->2->5       <== Completed EXIT Path with Exit Block #5
```

Figure 7: Example of the Function Loop Path Analysis Sequence
(Reference the Path Analysis Algorithm in Figure 4)

Once the static analysis is complete, the timing tool must consider the caching behavior for each instruction in each path. However, the caching behavior is dependent on the context from which the instruction is referenced. The static cache simulator recognizes this dependency and produces caching behavior for each function instance. A *function instance* is dependent upon the immediate call site within its caller as well as the caller’s call site etc.

Figure 8 illustrates that the static cache simulator categorizes the worst-case instruction caching behavior into one of four categories for every instruction within every function instance. This example assumes the cache consists of four lines, each containing four instructions. For the worst-case an instruction is classified as an *always* *miss* if the instruction is not guaranteed to be in the cache when referenced. An instruction is classified as an *always hit* if the instruction is guaranteed to always be in the cache when referenced. Instructions categorized as *first miss* are not guaranteed to be in the cache on the first reference to the instruction, but are guaranteed to be in cache for all remaining iterations of the loop. Instructions categorized as *first hit* indicate the opposite. In other words, for all references to the instruction during execution of the current loop, the instruction can be guaranteed to be in cache for the first reference but can not be guaranteed to be in cache for all subsequent references.

The definition for each instruction caching category differs for the best-case analysis. An instruction is classified as an *always miss* if the instruction is guaranteed to not be in cache when referenced. An instruction is classified as an *always hit* if the instruction may be in cache when referenced. Instructions categorized as *first miss* are guaranteed to not be in the cache on the first iteration of the loop but may be in the cache on all remaining iterations of the loop. Instructions categorized as *first hit* may be in cache on the first iteration of the loop but is guaranteed to not be in the cache on all remaining iterations. Since consideration for nested loops affects whether or not an instruction is in the cache, the static cache simulator generates a classification for each loop level in which the instruction is contained.

```
1      extern char min, a[10];
2
3      main()
4      {
5        int i, high;
6
7        high = min;
8        for (i=0; i < 10; i++)
9          if (high < value(i))
10             high = value(i);
11       return high;
12     }
13
14     int value(index)
15     int index;
16     {
17       return a[index];
18      }
```

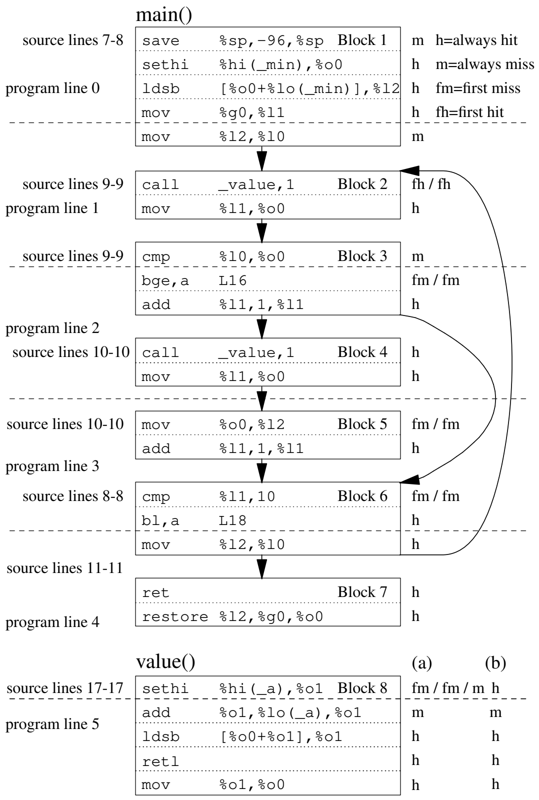

Figure 8: Simple Program Instructions with Categorizations

The timing tool will use the results from the static analysis and the caching categorization to build the timing tree. The algorithm to build this timing tree is depicted in Figure 9. The timing tree for the simple program used earlier in this paper is shown in Figure 10. In step 1 the algorithm is parsing the information from the static cache simulator to create nodes in the timing tree for each function instance. Stored in this node are all relevant information to uniquely identify the function instance, the list of instructions contained in the function, and the caching behavior for each instruction.

The process of creating nodes in the structure is completed in step 2 by using the loop information from the static analysis to insert a node for each loop in every function instance. Each of these loop nodes will be used in the timing analysis to generate execution times. Therefore, the instructions referenced in the nested loop must be moved from the function node to the loop node's instruction list. This is accomplished in step 2B of the algorithm. The final tree structure is then created in step 2C by parsing the instruction list at each node to identify calling sequences and creating links between *parent* and *child* loops.

(I) Read in the static cache simulation results.
  (1) For each function instance
    (A) Create a node in the linked list, indicating the node represents a function and saving the function name, instance, and parent node name.
    (B) Read in each instruction, saving the instruction number, the loop to which the instruction belongs, the invoked function and instance if the instruction invokes a function, and the caching behavior for the instruction.

(II) Build the timing tree.
  (1) Parse the linked list and locate the main function.
  (2) For each function node in the list, and referring the static analysis for that specific function:
    (A) If the function contains nested loops, then for each loop in the function create a node in the list, indicating the node represents a loop. Set the corresponding pointers for the parent node to reference the child node. Set the corresponding pointer in the newly created node to reference the static analysis information.
    (B) Parse the linked list of nodes, and for each "function node" move instructions from its instruction list to the node for the loop which contains the instruction.
    (C) For each node in the tree parse the instruction list and for each function invocation, set a pointer from the current node to the correct instance of the invoked function.

Figure 9: Algorithm to Create the Timing Tree

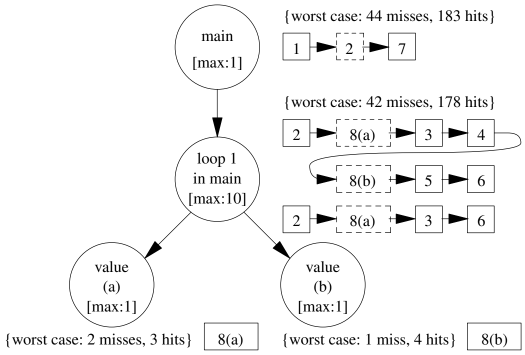

Figure 10: Timing Analysis Tree for the Simple Program

CHAPTER 4

LOOP ANALYSIS

The loops in the timing analysis tree are processed in a bottom-up manner. The execution time for a loop is not calculated until the times for all of its immediate child loops are known. Note that the timing analysis does not support recursive programs. There will be a worst-case and best-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 nested loop in Figure 3 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.

The worst-case loop algorithm depicted in Figure 11 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.

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. This step calculates a base time that is used when first misses and first hits have already been encountered. 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 which first misses will now be counted as hits on subsequent references. 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.

Figure 11: Algorithm to Determine Worst-case Path Through a Loop

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 hits. 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), 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 algorithm for estimating the best-case time for a loop is somewhat simpler. Let n be the minimum number of iterations associated with a loop. The best-case loop estimation algorithm is as follows:

(1) If n is equal to 1, then set the total best-case time of the loop to 0 and go to step 5.
(2) Calculate the minimum time required to execute any continue path assuming that all first misses are counted as misses and all first hits are counted as hits.
(3) Calculate the minimum time required to execute any continue path assuming that all first misses are counted as hits and all first hits are counted as misses.
(4) Multiply the value calculated in step 3 by (n-2) and add it to the value calculated in step 2. Set the total best-case time of the loop to this value.
(5) Calculate the times for all exit paths for the last iteration. For each set of exit paths that have a transition to a unique exit block, add the total best-case time for the loop to the shortest time within that set to produce the total best-case time associated with that exit block of the loop.

Figure 12: Algorithm to Determine Best-case Path Through a Loop

The best-case algorithm selects the shortest path on each iteration of the loop. In order to demonstrate the correctness of the algorithm, one must show that no other path for a given iteration will produce a shorter best-case time than that calculated by the algorithm. The time for the first iteration is typically calculated in step 2 (i.e. when the loop iterates more than once). The first time program lines are referenced in a loop, first misses will be misses and first hits will be hits. Thus, step 2 will calculate the shortest path for the first iteration. Step 3 calculates the shortest continue path given that first misses will be hits and first hits will be misses. All the first hits within the loop will be encountered on the first iteration according to the definition of first hits that was used by the authors. Thus, they can be safely treated as misses on subsequent iterations. A first miss will be a hit if it has been encountered previously. Even if a first miss had not been encountered in the first iteration, treating the reference as a hit in the second iteration will only cause a slight underestimation. Step 4 adds the time for the first iteration to the time calculated for the next n-2 iterations. Step 5 examines the last iteration separately since paths associated with the exit blocks may be shorter than the shortest continue path.

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 *base time* associated with that child loop will be used in the calculation of the path time. This base time is determined from the longest execution time for the loop path in which first misses are considered hits and first hits are considered misses. An *adjust value* is calculated as the sum of the differences between each path's time and the base time for each positive difference.

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 1. The *fm=>fm* adjustment is necessary since there should be only one miss associated

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 1: Use of Child Loop Times

with the instruction and a miss should only occur the first time the child loop is entered. The *m=>fh* adjustment is necessary since the first reference will be a hit at the outer loop level.

The bottom-up approach used to time the loops creates the potential to underestimate the WCET. In calculating the child loop's base time the assumption is made that the child loop will encounter all the first misses used in this base path. This assumption fails for child loops containing a greater number of loop paths than the maximum number of iterations. This failure generates the potential to underestimate the WCET, which is unacceptable. The *adjust value* is added to the *base time* only for the first iteration of the child node. Timing values for subsequent iterations of the child node use only the base time.

To illustrate this situation consider the code segment in Figure 13. The function *fun* contains 3 possible paths as shown in Table 2. The instruction cache is structured to contain 32 cache lines with each cache line containing 4 bytes. This results in the entire function fitting into cache and with each instruction categorized as a first miss. The actual execution time for the function *fun* iterated 5 times is 132 cycles. This value is derived from summing the time for each iteration, 50+43+25+7+7.

In analyzing this function the timing tool will identify path 1 as the worst-case path with a WCET of 70 cycles. Path 2 contains two instructions not encountered in path 1 and categorized as first misses. This results in a WCET of 25 cycles for path 2. Path 3 contains 2 instructions not encountered in path 1 or path 2 and also categorized as first misses so its WCET is calculated to be 23 cycles. In timing the function from the context of the outer loop the timing tool will calculate the WCET for the first iteration

```
main()
{
   int count, sum=0;

   for (count=0; count < 5; count++)
      sum += fun(count)
}

      C Source Code              Inst      Assembly Code
-------------------------        ----      -------------------------
int fun (i)                        0       cmp    %o0,%g0
int i;                             1       bne,a  L22
{                                  2       cmp    %o0,1
  if (i == 0)                      3       retl
    return 1;                      4       mov    1,%o0
  else if (i == 1)                 5 L22:  bne    L23
    return 2;                      6       nop
  else                             7       retl
    return 3;                      8       mov    2,%o0
}                                  9 L23:  retl
                                  10       mov    3,%o0
```

Figure 13: Example for Loop Adjust Values

as 70 cycles. The WCET for the base path is calculated to be 7 cycles since it is assumed that all first misses have been encountered. Without the use of adjust values the total WCET for the function will be calculated as the first iteration WCET + (base path X 4 remaining iterations) for a total of 98 cycles. This results in an underestimation for the WCET.

Path Number	List of Instructions	WCET if newly encountered *fm = miss*	WCET if newly encountered *fm = hit*
1	0,1,2,5,6,9,10	70	7
2	0,1,2,5,6,7,8	25	7
3	0,1,2,3,4	23	5

Table 2: Path Information Pertaining to Function `fun()` in Fig. 13

The use of an adjust value avoided this underestimation. The adjust value is calculated as the difference between the base time and initial path times, e.g. (70 - 7) + (25 - 7) + (23 - 7) resulting in 97 cycles. The first iteration WCET is calculated as the base time (7 cycles) + the adjust value (97 cycles) for a total of 104 cycles. For each subsequent iteration only the base value is used. The resulting WCET for the loop is now 132 cycles; which is an exact prediction for this example. This adjust value technique results in a slight overestimation if the number of loop iterations is fewer than the number of paths. However, this overestimation is preferable to an underestimation.

To illustrate the use of the worst-case algorithm, the calculation of the worst-case instruction cache performance for the example shown in Figure 8 will be described. The worst-case performance results for each loop in the timing analysis tree are shown in Figure 10. 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 Figure 11 for 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 1 (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.

CHAPTER 5

RESULTS

To assess the effectiveness of the timing analyzer, six simple programs were selected. A description of these programs is given in Table 3. For each program a direct-mapped cache configuration containing 8 lines of 16 bytes was used. Thus, the cache contains 128 bytes. A very small cache size was chosen because the test programs were relatively small themselves. The instruction cache performance of each entire program was predicted. The sizes of these test programs may be comparable to the size of typical code segments containing timing constraints in real-time applications. In addition, the code executed between two scheduling points (context switches) in a non-preemptive system may often be smaller than the code of a typical program. Using a small cache also provided a more realistic simulation of a typical ratio of program to cache size. The programs were 4 to 17 times larger than the cache as shown in column

Name	Num Bytes	Num Funcs	Description or Emphasis
Des	2,240	5	Encrypts and Decrypts 64 Bits
Matcnt	800	8	Counts and Sums Values in a 100x100 Matrix
Matmul	788	7	Multiplies 2 50x50 Matrices
Matsum	632	7	Sums Nonnegative Values in a 100x100 Matrix
Sort	572	5	Bubblesort of 500 Numbers in Ascending Order
Stats	1,488	8	Calcs Sum, Mean, Var., StdDev., & Linear Corr. Coeff.

Table 3: Test Programs

2 of Table 3. The analysis of test cases with smaller ratios, where test programs fit into the instruction cache, could be accomplished quite easily and would not represent a significant challenge. Using a smaller cache demonstrates the ability of the timing analyzer to predict tight bounds under a more difficult setting. Column 3 shows that each program was highly modularized to illustrate the handling of timing predictions across functions.

A distribution of the worst and best-case instruction categorizations is shown in Table 4. These numbers indicate 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.

Name	Always Hit		Always Miss		First Miss		First Hit	
	Worst	Best	Worst	Best	Worst	Best	Worst	Best
Des	70.00%	70.61%	27.28%	17.50%	1.93%	4.26%	0.79%	0.18%
Matcnt	70.64%	71.81%	25.48%	22.48%	2.65%	1.87%	1.22%	0.21%
Matmul	71.15%	71.75%	24.51%	20.00%	3.57%	2.65%	0.77%	0.17%
Matsum	69.89%	69.89%	26.24%	23.30%	3.87%	2.28%	0.00%	0.00%
Sort	67.70%	68.12%	28.42%	23.60%	3.26%	4.40%	0.62%	0.21%
Stats	71.76%	72.03%	24.30%	22.16%	3.55%	2.16%	0.39%	0.13%
Average	70.19%	70.70%	26.04%	21.51%	3.14%	2.94%	0.63%	0.15%

Table 4: Static Categorization Distribution for the Test Programs

Table 5 shows the dynamic results associated with these test programs. Only *Matmul* consistently 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 matrix multiplication. The *Observed Cycles* shows the cycles spent for an execution with worst-case and best-case input data. The number of cycles was measured using a traditional cache simulator [17], where a hit required one cycle and a miss required ten cycles (a miss penalty of nine cycles). The *Estimated Cycles* represents the number of cycles estimated by the timing analyzer. The *Estim. Ratio* depicts the ratio of the predicted instruction cache performance using the timing analyzer (*Estimated Cycles*) to the observed worst-case performance (*Observed Cycles*). The *Naive Ratio* shows a similar ratio assuming there was no cache analysis. This worst-case 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). Likewise, the best-case naive prediction determines the minimum number of instructions that could be executed and assumes that each instruction reference requires one cycle (hit time).

Name	Worst Case					Best Case				
	Hit Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Hit Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	81.41%	142,956	163,159	1.14	3.86	86.38%	59,998	19,399	0.32	0.21
Matcnt	85.32%	959,064	1,049,064	1.09	4.31	88.49%	719,082	719,082	1.00	0.49
Matmul	99.05%	2,917,887	2,917,887	1.00	9.21	99.05%	2,917,887	2,917,887	1.00	0.92
Matsum	87.09%	677,210	677,210	1.00	4.63	86.21%	657,210	657,210	1.00	0.45
Sort	83.99%	7,640,132	15,222,440	1.99	8.16	99.63%	10,951	4,466	0.41	0.38
Stats	88.59%	357,432	357,432	1.00	4.93	88.59%	357,432	357,432	1.00	0.49

Table 5: Dynamic Results for the Test Programs

The example programs were used to 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 and best-case performance as compared to observed performance can be estimated very tightly. In fact, the timing analyzer is able to calculate exact predictions of instruction cache performance when there is no conditional control flow other than iterating through loops.

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 intentionally 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. 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 best-case estimation was not affected since the best-case algorithm assumed the shorter path would always be taken.

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 and best-case paths in a function within *Des* primarily due to data dependencies. A longer or shorter 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 and minimum loop iterations from the control-flow information produced by the compiler or requests a maximum and minimum 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 for the worst-case estimation on average by a factor of two for this specific loop. The inner loop contributed much less to the total executed instructions for best-case since the outer loop was aborted after the first iteration when it was found that the array was sorted. However, the number of iterations performed for the single execution of this inner loop for the best-case estimation was still underrepresented by a factor of *N-1*, where *N* is the number of elements in the array. This inaccuracy accounted for the error in both the estimated and naive ratios since much of the cycles for the program were produced within this loop. Note that both of these problems are encountered by other timing tools and are not related to cache predictability.

CHAPTER 6

FURTHER WORK

This research has been extended significantly in three different areas after the programming was completed by this author. The first extension was the inclusion of a graphical user interface [9], [10] to allow the user to request timing results on specific sections of the program. Through the presentation of the source code, assembly code, and all possible subpaths, the user is able to quickly identify a particular program segment for which timing predictions can be requested. This was possible since the timing tool can accept requests to analyze individual loop subpaths. Timing requests for program source code segments was possible by tracking the source code lines down to the basic block level.

The second extension includes consideration of pipelining issues for the MicroSPARC 1. The technique described in this thesis did not consider pipelining issues. The code structure for the timing analysis tool was designed to allow the pipelining extension at a later date [19], [20].

Research is ongoing that addresses data caching [21]. Due to the possibility for data addresses to change during a program's execution, determining bounds for the worst-case and best-case data cache performance is much more complex. However, reasonable bounds can be calculated after determining the range of addresses for many data references.

Another area of work recently begun is directed to resolve the possibility for exponential growth in the number of possible paths through a loop. If a loop contains *n* consecutive IF statements, the total number of paths through this loop is 2^n. Work is proceeding for partitioning a loop or function into sections to reduce the number of paths at a given level in the program.

CHAPTER 7

CONCLUSION

Predicting the execution time of a program on a processor that uses cache memory has long been considered an intractable problem [3], [22], [23]. This research has developed a technique for predicting a bounds on the worst-case and best-case instruction cache performance. By statically analyzing the program structure and using caching behavior classifications from the static cache simulator, this timing analyzer is able to exactly predict instruction cache performance when there is no conditional control-flow other than iterating through loops. Tight predictions can be obtained for many programs with conditional control-flows, as demonstrated in this thesis.

This research demonstrates that instruction cache behavior is sufficiently predicatable for real-time applications. Thus, instruction caches should be enabled, generating a significant speedup for the predicted performance as compared to disabled caches (depending upon the hit ratio and miss penalty). 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. Thus, methods to predict caching behavior will become an essential part of timing analysis techniques.

REFERENCES

[1] C.L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *ACM* **20**(1) pp. 46-61 (January 1973).

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

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

[4] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

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

[6] 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).

[7] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding Instruction Cache Performance," *ACM Transactions on Computer Systems*, (submitted June 1995).

[8] F. Mueller and D. B. Whalley, "Fast Instruction Cache Analysis via Static Cache Simulation," *Proceedings of the 28th Annual Simulation Symposium*, (April 1995).

[9] L. Ko, D. B. Whalley, and M. G. Harmon, "Supporting User-Friendly Analysis of Timing Constraints," *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp. 107-115 (June 1995).

[10] L. Ko, C. Healy, E. Ratliff, M. Harmon, R. Arnold, and D.B. Whalley, "Supporting the Specification and Analysis of Timing Constraints," *IEEE Real-Time Technology and Applications Symposium* **2**(1) pp. 170-178 (June 1996).

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

[12] 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).

[13] 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).

[14] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim, "An Accurate Worst Case Timing Analysis Technique for RISC Processors," *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, pp. 97-108 (December 1994).

[15] Y. S. Li, S. Malik, and A. Wolfe, "Performance Estimation of Embedded Software with Instruction Cache Modeling," *International Conference on Computer-Aided Design*, (November 1995).

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

[17] 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).

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

[19] Christopher A. Healy, *Predicting Pipeline and Instruction Cache Performance,* Masters Thesis, Florida State University, Tallahassee, FL (1995).

[20] C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, (December 1995).

[21] R. White, *Bounding Worst-Case Data Cache Performance,* PhD Prospectus, Florida State University, Tallahassee, FL (July 1996).

[22] 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).

[23] 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).

BIOGRAPHICAL SKETCH

Robert Arnold received a Bachelor of Science and Engineering degree in Computer Information Sciences from the University of Florida in 1989. Upon graduation Mr. Arnold was employed by Motorola Inc. to develop embedded systems applications at its Plantation Florida facility. He is currently employed by Peek Traffic Transyt in Tallahassee Florida to develop embedded systems applications for its line of traffic signaling equipment.

ACKNOWLEDGEMENTS

I wish to thank my major professor, Dr. David Whalley for his guidance and support during this research. I am also grateful for the assistance Frank Mueller provided to interface the static cache simulator with this timing tool. I am also grateful for the continuous love and support given by my wife Debbie. The research upon which this thesis is based was supported by the Office of Naval Research under contract number N00014-94-1-0006.

TABLE OF CONTENTS

LIST OF TABLES

TABLE NUMBER AND DESCRIPTION **PAGE**

LIST OF FIGURES

FIGURE NUMBER AND DESCRIPTION **PAGE**

ABSTRACT

The use of cache poses a difficult tradeoff for real-time system developers. While caches provide significant performance advantages, they have also been considered inherently unpredictable since the behavior of a cache reference depends upon the previous references accessing the same cache line. The use of caches can only be suitable for real-time systems when caching behavior can be reliably predicted. This thesis describes an approach for bounding the instruction cache performance for large code segments. A timing analyzer was developed that uses caching behavior information generated from a static cache simulator to estimate the worst-case and best-case instruction cache performance for each loop and function in a program.