Bounding Pipeline and Instruction Cache Performance*

Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, Marion G. Harmon†

Abstract

Predicting the execution time of code segments in real-time systems is challenging. Most recently designed machines contain pipelines and caches. Pipeline hazards may result in multicycle delays. Instruction or data memory references may not be found in cache and these misses typically require several cycles to resolve. Whether an instruction will stall due to a pipeline hazard or a cache miss depends on the dynamic sequence of previous instructions executed and memory references performed. Furthermore, these penalties are not independent since delays due to pipeline stalls and cache miss penalties may overlap. This paper describes an approach for bounding the worst and best-case performance of large code segments on machines that exploit both pipelining and instruction caching. First, a method is used to analyze a program's control flow to statically categorize the caching behavior of each instruction. Next, these categorizations are used in the pipeline analysis of sequences of instructions representing paths within the program. A timing analyzer uses the pipeline path analysis to estimate the worst and best-case execution performance of each loop and function in the program. Finally, a graphical user interface is invoked that allows a user to request timing predictions on portions of the program. The results indicate that the timing analyzer efficiently produces tight predictions of worst and best-case performance for pipelining and instruction caching.

Index terms: real-time systems, worst-case execution time, best-case execution time, timing analysis, instruction cache, pipelining

1. Introduction

Many architectural features, such as pipelines and caches, present a dilemma for architects of real-time systems. Use of these architectural features can result in significant performance improvements. In order to exploit these performance improvements in a real-time system, the WCET (Worst Case Execution Time) must be predicted statically. In addition, sometimes the BCET (Best Case Execution Time) is also needed. However, the aforementioned performance enhancing features introduce a potentially high level of unpredictability. Dependencies between instructions can cause pipeline hazards that may delay the completion of instructions. While there has been much work accomplished

*This work was supported in part by the Office of Naval Research under contract number N00014-94-1-0006 and the National Science Foundation under the cooperative agreement HRD-9707076. Preliminary versions of this work were described in the 1994 *Real-Time Systems Symposium* under the title "Bounding Worst-Case Instruction Cache Performance" and the 1995 *Real-Time Systems Symposium* under the title "Integrating the Timing Analysis of Pipelining and Instruction Caching."

†C. A. Healy and D. B. Whalley are with the Department of Computer Science, Florida State University, Tallahassee, FL 32306-4530. R. D. Arnold is with Peek Traffic Systems Inc., 3000 Commonwealth Blvd., Tallahassee, FL 32303. F. Mueller is with the Institut für Informatik, Humboldt-Universität zu Berlin, Unter den Linden 6, D-10099 Berlin, Germany. M. G. Harmon is with the Department of Computer and Information Systems, Florida A & M University, Tallahassee, FL 32307-3101. The authors can be contacted at either [{whalley,healy}@cs.fsu.edu, (850) 644-3506, fax: -0058], [rarnold@transyt.peek-traffic.com, (850) 562-2253, ext. 272, fax: -4126], [mueller@informatik.hu-berlin.de, (+49) (30) 20181-276, fax: -280] or [harmon@cis.famu.edu, (850) 599-3042, fax: -3221].

on analyzing the execution performance of a sequence of instructions within a basic block, the analysis of pipeline performance across basic blocks is more problematic. Instruction or data cache misses further complicate the performance prediction problem since they require several more cycles to resolve than cache hits. Predicting the caching behavior of an instruction is even more difficult since it may be affected by memory references that occurred long before the instruction was executed.

The timing analysis of these features is further exacerbated since pipelining and caching behavior are not independent. For instance, consider the code segment and pipeline diagram in Figure 1 consisting of three SPARC instructions. The pipeline cycles and stages represent the execution of these instructions on a MicroSPARC I processor [1]. Each number within the pipeline diagram denotes that the specified instruction is currently in the pipeline stage shown on the left and is in that stage during the cycle indicated above. The first instruction performs a floating-point addition and requires a total of 20 cycles. Fetching the second instruction results in a cache miss, which is assumed to have a miss penalty of nine additional cycles in this paper. The third instruction has a data dependency with the first instruction and the execution of its MEM stage is delayed until the floating-point addition is

SPARC Instructions

```
inst 1: faddd  %f2,%f0,%f2
inst 2: sub    %o4,%g1,%i2
inst 3: std    %f2,[%o0+8]
```

Pipeline Diagram

		cycle																	
		1	2	3	4	5	...	11	12	13	14	15	16	17	18	19	20	21	22
stage	IF	**1**	2	2	2	2	...	2	**3**										
	ID		**1**						2	**3**									
	EX									**2**	3	3	3	3	3	**3**			
	FEX			**1**	1	1	...	1	1	1	1	1	1	1	1	**1**			
	MEM										**2**						3	3	**3**
	WB											**2**							
	FWB																**1**		

Figure 1. Example of Overlapping Pipeline Stages with a Cache Miss

completed.[1] The miss penalty associated with the access to main memory to fetch the second instruction is completely overlapped with the execution of the floating-point addition in the first instruction. If pipeline stalls and cache misses were treated independently, then the number of estimated cycles associated with these instructions would be increased from 22 to 31 (*i.e.* by the cache miss penalty).

Unfortunately, the problem of overestimating WCET and underestimating BCET may become more severe in the future. Cache miss penalties are increasing due to the growing gap between processor and main memory speeds. Delays due to pipeline stalls become more likely with the introduction of superscalar and superpipelined architectures. Thus, naive timing analysis of programs on machines with pipelines and caches will result in increased execution time prediction errors.

Let us define a task as 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. These caching predictions can be integrated with pipeline analysis to estimate tight WCET and BCET bounds.

Figure 2 depicts an overview of the approach described in this paper for bounding the worst and best-case performance of large code segments on machines with pipelines and instruction caches. Control-flow information, which could have been obtained by analyzing assembly or object files, is stored as the side effect of the compilation. This information identifies the loops that are in each function, the basic blocks that comprise each loop, the instructions that reside in each basic block, and the register operands associated with each instruction. The control-flow information is passed to a

[1] A `std` instruction has no write back stage since a store instruction only updates memory and not a register. The `std` instruction also requires three cycles to complete the MEM stage on the MicroSPARC I.

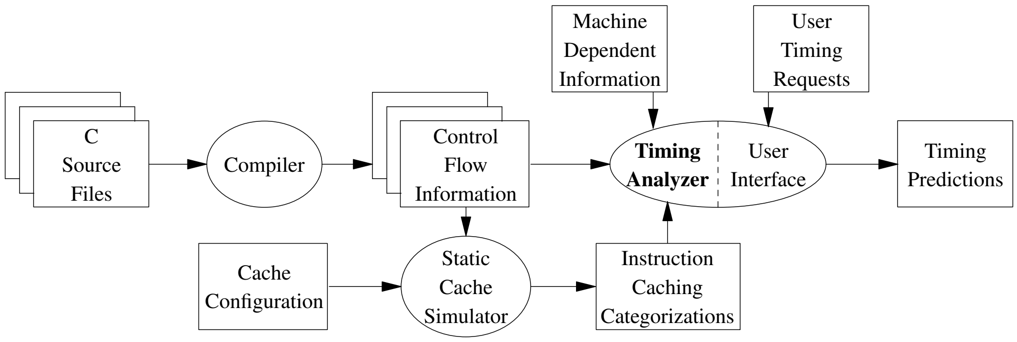

Figure 2. Overview of Bounding Pipelining and Instruction Caching Performance.

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's control-flow graph is then analyzed for a given cache configuration to produce a categorization of each instruction's potential caching behavior. The timing analyzer uses these categorizations to determine whether an instruction fetch should be treated as a hit or a miss during the pipeline analysis. It also reads machine-dependent and control-flow information to determine how each instruction proceeds through the pipeline. The timing analyzer produces a worst and best-case estimate of execution time for each loop and function within the program. Finally, a window-based interface is used to allow the user to request the timing bounds for portions of the program.

2. Instruction Caching Categorization

Static cache simulation[2] is used to statically categorize each instruction according to its caching behavior using a specific cache configuration in a given program. The static simulation consists of three phases. First, the control-flow graph of the entire program is constructed. This graph includes the control-flow information of each function and a function instance graph, which is simply a call

[2] Static cache simulation is only briefly introduced in this section. It is described in more detail elsewhere [2, 3, 4, 5, 6].

graph where each function instance is uniquely identified by the sequence of call sites required for its invocation. Thus, a directed acyclic call graph (without recursion) is transformed into a tree of function instances.

Next, this program control-flow graph is analyzed to determine the program lines that may be in cache at the entry and exit of each basic block within the program. The iterative algorithm in Figure 3 is used to calculate an input and output cache state for each basic block in the function instance graph. A cache state is simply the subset of all program lines that can potentially be cached at that point in the control flow. 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 above steps are repeated until no more changes occur.

```
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.

The input state for each basic block is used to categorize the caching behavior of each instruction within the block. The categorization for each loop level is obtained by examining the cache state for that instruction with a mask representing the program lines that are accessed by the loop. An instruction's caching behavior is assigned to one of four categories for each loop level in which an instruction is contained. Note that each function is treated as a loop that executes for a single iteration. The categorizations of worst and best-case instruction cache behavior are given in Tables 1 and 2. When processing an outer loop that contains an inner loop, the timing analyzer can adjust the value obtained from the timing associated with an inner loop by examining the transitions between categorizations of an instruction from one loop level to the next. These adjustments will be described in Section 5.

Informally, an instruction's worst-case cache categorization for a particular loop level is determined as follows. Let L be the program line that contains an instruction within a basic block. The instruction is categorized as an *always hit* if it is not the first instruction encountered in L in the block, or if L is in the abstract cache state and it does not conflict with any other program line in the same abstract cache state. The instruction is categorized as a *first hit* if it was a *first hit* for the previous (deeper)

Instruction Category	Definition According to Behavior in the Instruction Cache
always miss	The instruction is not *guaranteed* to be in cache when it is referenced.
always hit	The instruction is *guaranteed* to always be in cache when it is referenced.
first miss	The instruction is *not guaranteed* to be in cache on its first reference each time the loop is executed, but is *guaranteed* to be in cache on subsequent references.
first hit	The instruction is *guaranteed* to be in cache on its first reference each time the loop is executed, but is *not guaranteed* to be in cache on subsequent references.

Table 1. Definitions of Worst-Case Instruction Caching Categorizations

Instruction Category	Definition According to Behavior in the Instruction Cache
always miss	The instruction is *guaranteed* to *not* be in cache when it is referenced.
always hit	It is *possible* that the instruction is in cache every time it is referenced.
first miss	The instruction is *guaranteed* to *not* be in cache on its first reference each time the loop is executed, but *may be* in cache on subsequent references.
first hit	The instruction *may be* in cache on its first reference each time the loop is executed, but is *guaranteed* to *not* be in cache on subsequent references.

Table 2. Definitions of Best-Case Instruction Caching Categorizations

loop nesting levels or if all of the following conditions (1)-(6) hold:

(1) The instruction is the first reference to L in the block, and L is in the abstract cache state.

(2) There exists a program line in the abstract cache state for this loop that conflicts with L.

(3) L is in the abstract output cache state of all preheaders[3] of this loop.

(4) None of the conflicting lines is in the abstract output cache state of the preheaders of this loop. The purpose of this stipulation is to guarantee that the instruction will be a hit in cache on the first iteration of the loop, in accord with the definition of *first hit* in Table 1.

(5) L is in the post dominator of the loop's headers, *i.e.* the current line will be referenced during each loop iteration.[4]

(6) None of the conflicting lines is in the linear cache state of the current block, *i.e.* for each loop iteration, the current line will be referenced before any conflicting line. This requirement guarantees that L can only be replaced by a conflicting line after the instruction has been referenced at least once.[5]

An instruction is a *first miss* if it is not already categorized as an *always hit* or *first hit*, the instruction was a *first miss* at the next deeper loop nesting level (if that level exists), it is the first instruction encountered in L in the block and L is in the abstract cache state, and there exist conflicting program lines but only outside the current loop nesting level. In all other cases, the instruction is conservatively categorized as an *always miss*.

The instruction's best-case categorization is determined as follows. The instruction is categorized as an *always miss* if it is the first reference to L in the block and L is not in the abstract cache state. The instruction is categorized an a *first miss* if it was a *first miss* or *always hit* at the next deeper loop nesting level (if that level exists), this instruction is the first reference to L in the block, L is in the

[3] The loop header of a natural loop is the single basic block in which the loop is initially entered. The preheader is the basic block that precedes the header.

[4] Note that an instruction does not have to be referenced during each loop iteration to be classified as a *first miss*.

[5] The linear cache state of a block represents the hypothetical cache state in the absence of loops. During the static cache simulation, no linear abstract cache state information is propagated along back edges of a loop.

abstract cache state, and L is not in the linear cache state of the block.[4] The instruction is categorized as a *first hit* if it was a *first hit* for the previous (deeper) loop nesting levels or if the following conditions (1)-(5) hold:

(1) The instruction is the first reference to L in the block, and L is in the abstract cache state.

(2) There exists a program line in the abstract cache state for this loop that conflicts with L.

(3) L is in the abstract output cache state of all preheaders of this loop.

(4) L is in the post dominator of the loop's headers, *i.e.* the current line will be referenced during each loop iterations.

(5) L is not in the abstract cache state preceding any of the back edges, *i.e.* L is replaced by a conflicting line during each loop iteration. The purpose of this requirement is to guarantee that the program line conflicting with L will be encountered on every iteration after the first. Thus, the instruction will be a cache miss on these iterations, in agreement with the definition of *first hit* in Table 2.

In all other cases, the instruction is conservatively categorized as an *always hit*. Formal definitions of these instruction categorizations are given in the appendix.

The current implementation of the static simulator imposes some restrictions. First, only direct-mapped cache configurations are allowed.[6] Second, recursive programs are not allowed since cycles in the call graph would complicate the generation of unique function instances.[7] Finally, indirect calls are not handled since an explicit call graph must be generated.

---

[6] Recent studies have shown that direct-mapped caches often have a faster access time for hits, which sometimes outweigh the benefit of a higher hit ratio in set-associative organizations for large caches [7]. We are currently investigating the timing analysis of set-associative caches.

[7] 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. Pipeline Path Analysis

This section describes how the analysis of the pipeline performance of a sequence of instructions is accomplished. Information for all levels of timing analysis is stored in data structures as depicted in Figure 4. First, information about each type of instruction is read from a machine-dependent data file. This pipeline information for each type of instruction includes the worst and best-case number of cycles required by each stage of the pipeline for its execution.[8] The analyzer also reads from the machine-dependent data file other information for each instruction. This information includes the latest stage each source operand of an instruction can receive its value via hardware forwarding without causing a pipeline stall and the earliest stage in which the result of the instruction can be forwarded. Finally, information about the specific instructions in the sequence is obtained and stored in instances of `struct inst_node`. This information includes the actual registers associated with the source and destination operands, which is obtained from the control-flow information generated by the compiler, and the instruction caching categorization of each instruction, which is produced by the static cache simulator.

A path of instructions consists of all the instructions that can be executed during a single iteration of a loop (or in the case of a function, all the instructions that are executed in one invocation of the function). Thus, a path consists of a sequence of basic blocks connected by control-flow transitions. If a loop has no conditional control flow (e.g. `if` or `switch` statements), then there will be only one path associated with this loop.

During the analysis of a path, the analyzer stores path information in instances of `struct path_node`. This information includes the total number of cycles required by the path and a set of

[8] The number of cycles required for some floating-point instructions on processors can vary depending upon the values of its operands.

```
struct loop_node {                                  /* Information stored with each loop */
  int max_iterations;                               /* Maximum number of iterations for the loop */
  int min_iterations;                               /* Minimum number of iterations for the loop */
  int wcet;                                         /* Estimated WCET of the loop */
  int bcet;                                         /* Estimated BCET of the loop */
  struct union_node *wc_pipeline_information;       /* Worst-Case pipeline info. for detecting hazards */
  struct union_node *bc_pipeline_information;       /* Best-Case pipeline info. for detecting hazards */
  struct path_node *path_list;                      /* Linked list of loop's paths */
  struct inst_node *first_misses_encountered;       /* Linked list of first misses encountered in loop */
  struct inst_node *first_hits_encountered;         /* Linked list of first hits encountered in loop */
  struct exit_block_node *exit_block_list;          /* Linked list of blocks to which loop can exit */
  struct loop_node *next;                           /* Pointer to next loop node in program */
};

struct exit_block_node {                            /* Information stored with loop's exit block */
  int wcet;                                         /* Estimated WCET of loop exiting to this block */
  int bcet;                                         /* Estimated BCET of loop exiting to this block */
  struct union_node *wc_pipeline_information;       /* Worst-case pipeline info. for detecting hazards */
  struct union_node *bc_pipeline_information;       /* Best-case pipeline info. for detecting hazards */
};

struct union_node {                                 /* Information to detect structural and data hazards */
  int cycles_from_begin[NUM_STAGES];                /* Cycle when particular stage is initially occupied */
  int beginning_occupant[NUM_STAGES];               /* Number of instruction that first occupies stage */
  int cycles_from_end[NUM_STAGES];                  /* When stage is last occupied */
  int ending_occupant[NUM_STAGES];                  /* Number of instruction occupying stage last */
  int reg_first_needed[NUM_REGS];                   /* Cycle when value of register is first needed */
  int reg_last_produced[NUM_REGS];                  /* Cycle when register is last used as destination */
};

struct path_node {                                  /* Information stored for each path in a loop */
  int path_type;                                    /* Type of path: continue, exit, or both */
  int wcet;                                         /* Estimated WCET of the path */
  int bcet;                                         /* Estimated BCET of the path */
  struct union_node *wc_pipeline_information;       /* Worst-Case pipeline info. for detecting hazards */
  struct union_node *bc_pipeline_information;       /* Best-Case pipeline info. for detecting hazards */
  struct block_node *block_list;                    /* Linked list of basic blocks in path */
  struct path_node *next;                           /* Pointer to next path in loop */
};

struct block_node {                                 /* Basic blocks contain list of instructions */
  struct inst_node *inst_list;                      /* Linked list of instructions in block */
  struct block_node *next;                          /* Pointer to next block in path */
};

struct inst_node {                                  /* Information stored for each instruction */
  int inst_type;                                    /* Opcode for this instruction */
  int register_operands[NUM_REGS_PER_INST];         /* Register operands for this instruction */
  struct cat_node *cat_list;                        /* Instruction cache categorization for this inst */
  struct inst_node *next;                           /* Pointer to next instruction in path */
};

struct cat_node {                                   /* Instruction cache categorizations */
  char wc_cat;                                      /* Worst-case categorization */
  char bc_cat;                                      /* Best-case categorization */
  struct cat_node *next;                            /* Pointer to next deeper nesting level cat's */
};
```

Figure 4. Data Structures for Timing Analysis

pipeline information. This information includes when each pipeline stage was first and last used within the path for avoiding structural hazards.[9] It is represented as the number of cycles from the beginning and end of the path for each pipeline stage. In addition, information indicating when each register was first and last used in the path is also maintained to avoid data hazards.[10] Again, this information is represented as the number of cycles from the beginning and end of the path for each register. The set of pipeline information, as stored in `path->wc_pipeline_information`, for avoiding hazards after the three instructions in Figure 1 have been analyzed is shown in Tables 3 and 4. Table 3 represents the information for avoiding structural hazards. Only the numbers shown in bold are required to be stored. These values represent when each stage was first used from the beginning of the path and last used from the end. The values in the table correspond to the information associated with the instruction numbers that are represented in bold in Figure 1. Table 4 represents the information for avoiding data hazards.

Stage	IF	ID	EX	FEX	MEM	WB	FWB
Beginning Inst	1	1	2	1	2	2	1
Cycles from Beg	**0**	**1**	**12**	**2**	**13**	**14**	**19**
Ending Inst	3	3	3	1	3	2	1
Cycles from End	**10**	**9**	**3**	**3**	**0**	**7**	**2**

Table 3. Structural Hazard Information for the Instructions in Figure 1.

Register	%g1	...	%o0	...	%o4	...	%i2	...	%f0	...	%f2	...
First Needed	**12**	**N/A**	**13**	**N/A**	**12**	**N/A**	**N/A**	**N/A**	**2**	**N/A**	**2**	**N/A**
Last Produced	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**N/A**	**9**	**N/A**	**N/A**	**N/A**	**3**	**N/A**

Table 4. Data Hazard Information for the Instructions in Figure 1.

[9] A structural hazard indicates that a stage of an instruction cannot be executed earlier due to the pipeline stage already being used.

[10] A data hazard indicates that a particular stage of an instruction cannot be executed earlier due to the pipeline stage using a source register that matches the destination register not yet updated by a pipeline stage of another instruction.

This set of pipeline information is created by processing one instruction at a time from the sequence of instructions that comprise a path. Figure 5 depicts an algorithm that creates this pipeline information for worst-case analysis. The best-case path analysis algorithm is analogous. Each instruction can be represented by the same form of pipeline information that is shown in Tables 3 and 4 for a path. This information is modified if it is found that the instruction's caching categorization indicates that the instruction fetch was a miss. The miss penalty is used to increment the total number of cycles and the cycles from the beginning (structural hazard information) for all other stages besides the IF stage and the first needed registers (data hazard information) for that instruction. The addition of an instruction to the pipeline information for a path will not only update the total number of cycles and the information associated with the end of the pipeline, but also the beginning of the pipeline if a referenced stage or register in the instruction had not been previously used.

```
void Time_Path (struct path_node *path) {
   struct block_node *block;
   struct inst_node *instruction;

   path->wc_pipeline_information = NULL.
   FOR each block in path->block_list DO
      FOR each instruction in block->inst_list DO
         IF (instruction->cat_list->wc_cat == first miss AND
            this instruction has not been encountered already) OR
            (instruction->cat_list->wc_cat == first hit AND
            this instruction has not been encountered already) OR
            instruction->cat_list->wc_cat == miss THEN
            Treat this instruction fetch as a miss in the pipeline.
         ELSE
            Treat this instruction fetch as a hit in the pipeline.
         Concatenate w.c. pipeline information for instruction->inst_type
            with path->wc_pipeline_information.
      END FOR
   END FOR
   path_ptr->wcet = temporal length of path->wc_pipeline_information.
}
```

Figure 5. Worst-Case Path Analysis Algorithm.

Retaining this set of pipeline information allows additions to the beginning or end of a path. Since both the pipeline requirements for a path and a single instruction can be represented with this set of pipeline information, concatenating two paths together can be accomplished in the same manner as concatenating an instruction onto the end of a path. The concatenation is accomplished one stage at a time. A stage from the second set of pipeline information is moved to the earliest cycle that does not violate any of the following conditions.

(1) There is no structural hazard with another instruction. For instance, the beginning of the IF stage of instruction 2 in Figure 1 could not be placed in cycle 1 since that stage was already occupied.

(2) There is no data hazard due to a previous instruction producing a result that is needed by a source operand of the current instruction in that stage. For example, the beginning of the MEM stage for instruction 3 in Figure 1 could not be moved past the FEX stage of instruction 1 at cycle 19 due to the data hazard between the `faddd` and `std` instructions.

(3) The placement of the instruction does not violate its own pipeline requirements. For instance, the ID stage of instruction 2 has to occur at least 10 cycles after the beginning of its IF stage in Figure 1.

Other information associated with the pipeline analysis of a path need not be stored. For instance, it does not matter when instruction 2 entered the ID stage after the pipeline information has been calculated for all three instructions in Figure 1. No instruction being added to either the beginning or end of the pipeline could possibly have a structural hazard with the ID stage of instruction 2 since it would first have a structural hazard with the ID stage of instruction 1 or instruction 3, respectively. Thus, the amount of pipeline information associated with a path is dramatically reduced as opposed to storing how each stage is used during every cycle. Furthermore, no limit need be imposed on the amount of potential overlap when concatenating the analysis of two paths.

4. Loop Analysis

In order to predict the worst-case execution time of a loop, the timing analyzer has to predict the execution time of each possible path within the loop. The static cache simulator provides categorizations for each instruction. The timing analyzer will reserve either one cycle or the number of cycles associated with a cache miss for the IF (instruction fetch) stage for each instruction categorized as an always hit or always miss, respectively. Note, additional cycles in the IF stage may be required due to other pipeline stalls. If an instruction is categorized as a first miss, then the timing analyzer will treat the instruction fetch as a miss if the program line has not yet been encountered as a first miss in the timing of the loop. If the program line has been encountered, then the instruction fetch will be treated as a hit instead. Likewise, if an instruction is categorized as a first hit, then the timing analyzer will treat the instruction fetch as a cache hit on the first reference and a cache miss thereafter.

Each path starts with the loop header and is terminated by a block with a back edge[11] or a transition to an exit block outside the loop. Figure 6 shows a simple example that identifies a loop header, back edges, exit blocks, continue paths, and exit paths. Each path is designated as either a continue path (the last block is the head of a back edge transition), an exit path (the last block has a transition to an exit block outside the loop), or both. The number of loop iterations indicates the number of times the header of the loop is executed once the loop is entered.

---

[11] A back edge is a control-flow transition from a basic block in a loop to its loop header.

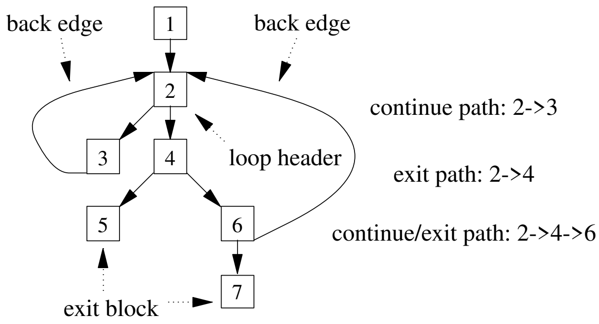

Figure 6. Example Introducing Loop Terminology

With pipelining it is possible that the combination of a set of paths may produce a longer worst-case execution time than just repeatedly selecting the longest path. For instance, consider a loop with two paths that take about the same number of cycles to execute. One path has a floating-point addition near the beginning of the path and the other path has a floating-point addition near the end. Alternation between the paths will produce the worst case execution time since there will be a structural hazard between the two floating-point additions.

To avoid the problem of calculating all combinations of paths, which would be the only method for obtaining perfectly accurate estimations, it was decided to union the pipeline effects of the paths for a single iteration of a loop together. A union, an instance of `struct union_node` in Figure 4, is dynamically allocated for each path and loop. Calculating the union of the beginning pipeline structural hazard information for a given stage in the WCET analysis is accomplished by determining the earliest initial occupation of that stage by any path in the union. Likewise, we calculate the WCET union of the ending pipeline structural hazard information for a given stage by finding the last occupation of that stage, relative to the last cycle of the longest path, by any path in the union. The BCET unioning of pipeline information is accomplished in an analogous manner. The beginning (ending) pipeline structural hazard information for each stage is updated to contain the latest initial (earliest final) occupation of that stage. If a path does not use a particular stage, then the BCET union will record that stage as empty. The data hazard information is handled similarly with the earliest and latest use of each register from the paths in the union being updated. This unioning of pipeline information simplified the algorithm and also did not cause a noticeable overestimation or underestimation in the worst or best-case analysis, respectively. The beginning pipeline information (stages and registers) is rarely affected since all paths through a loop start with the same loop header block. Paths through a loop often end with the same block of instructions. In addition, one path may be significantly longer or shorter than the others, so the ending pipeline information for worst and best-case analysis is often not affected.

Figure 7 shows a toy function and its corresponding SPARC assembly code.[12] There are two

```
      C Source Code         Inst    Assembly Code
-------------------------   ----    --------------------------
main()                         0     mov   %g0,%o1
{                              1     sethi %hi(L01),%o0
   int i, cnt = 0;             2     ldd   [%o0+%lo(L01)],%f2
   double dcnt = 0.0;          3     mov   %g0,%o2
   extern int incr;            4     sethi %hi(_dincr),%o3
   extern double dincr;        5     sethi %hi(_incr),%o4
                               6     cmp   %o2,5
   for (i=0; i < 10; i++)      7 L8: bge,a L9
      if (i < 5)               8     ld    [%o4+%lo(_incr)],%o0
         dcnt += dincr;        9     ldd   [%o3+%lo(_dincr)],%f0
      else                    10     ba    L6
         cnt += incr;         11     faddd %f2,%f0,%f2
}                             12 L9: add   %o1,%o0,%o1
                              13 L6: add   %o2,1,%o2
                              14     cmp   %o2,10
                              15     bl,a  L8
                              16     cmp   %o2,5
                              17     retl
                              18     nop
```

Figure 7. Example C Source Code and Corresponding SPARC Instructions.

[12] Note that the generated assembly code has been optimized by the compiler. The local variables `i`, `count`, and `dcount` have been allocated to registers `%o2`, `%o1`, and `%f2`, respectively. The instruction following each transfer of control takes effect before the transfer of control is taken since the SPARC has delayed branches. The `cmp` comparison preceding the `bge` branch (instruction 7) has been moved to both immediately precede the loop and in the delay slot (instruction 16) of the `bl` branch (instruction 15). Branches with a "`,a`" represent that the result of the instruction within the delay slot will be annulled if the branch is not taken.

possible paths of instructions through an iteration of the loop in the program, <7,8,12,13,14,15,16> and <7,8,9,10,11,13,14,15,16>. Figure 8 shows the instructions and the corresponding pipeline diagrams for the two paths within the loop.[13] To simplify the example, it is assumed that the loop has already been executed and all of the instructions and data are in cache (*i.e.* there are no instruction fetch or data memory misses). Table 5 shows the structural hazard information for the two paths in Figure 7 and how the information in path 1 has to be adjusted before being unioned. The worst-case union of the number of cycles from the beginning and end of the paths for a given stage will simply be the minimum number encountered. Likewise, the best-case union will be the maximum number encountered. The structural hazard information indicating the number of cycles from the end of path 1 has to be adjusted since its total number of cycles is 13 less than the cycles required by path 2. The

Path 1 Instructions

```
inst  7: bge,a L19
inst  8: ld    [%o4+%lo(_incr)],%o0
inst 12: add   %o1,%o0,%o1
inst 13: add   %o2,1,%o2
inst 14: cmp   %o2,10
inst 15: bl,a  L18
inst 16: cmp   %o2,5
```

Path 1 Pipeline Diagram

	cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	24	25
stage	IF	**7**	8	12	13	13	14	15	**16**									
	ID		**7**	8	12	12	13	14	15	**16**								
	EX				**8**		12	13	14		**16**							
	FEX																	
	MEM					**8**		12	13	14		**16**						
	WB						**8**		12	13	14		**16**					
	FWB																	

Path 2 Instructions

```
inst  7: bge,a L19
inst  8: ld    [%o4+%lo(_incr)],%o0
inst  9: ldd   [%o3+%lo(_dincr)],%f0
inst 10: ba    L16
inst 11: faddd  %f2,%f0,%f2
inst 13: add   %o2,1,%o2
inst 14: cmp   %o2,10
inst 15: bl,a  L18
inst 16: cmp   %o2,5
```

Path 2 Pipeline Diagram

	cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	24	25
stage	IF	**7**	8	9	10	11	13	13	14	15	**16**							
	ID		**7**	8	9	10	11	11	13	14	15	**16**						
	EX				**8**	9				13	14		**16**					
	FEX								**11**	11	11	11	11	11	11	...	**11**	
	MEM					**8**	9	9			13	14		**16**				
	WB						**8**					13	14		**16**			
	FWB								**9**									**11**

Figure 8. Pipeline Diagrams for the Two Paths through the Loop in Figure 7.

[13] Note instructions 7, 10, and 15 are transfers of control. The actual transfer of control (*i.e.* updating the program counter) occurs in the ID stage. Thus, there are no additional pipeline stages associated with these instructions. Also note the one cycle stall between instructions 8 and 12 in the EX stage of path 1 due to a load hazard. Finally, the `ldd` (instruction 9) requires two cycles to complete the MEM stage [1].

resulting worst-case union of the structural hazard information of the two paths would be identical to the structural hazard information for path 2. Likewise, the best-case union would be identical to the information for path 1. Note that the data hazard information would change slightly since instruction 12 references register `%o0` as a source operand and `%o1` as both a source and destination. Yet, representing access to these registers would not likely have an effect when the timing analysis is performed between this path and its predecessor and successor paths since the EX stage is used before and after cycle 6, which is when instruction 12 enters the EX stage.

Path 1 Info	IF	ID	EX	FEX	MEM	WB	FWB
Cycles from Beg	0	1	3	N/A	4	5	N/A
Cycles from End	4	3	2	N/A	1	0	N/A
Adj End Cycles	17	16	15	N/A	14	13	N/A
Path 2 Info	**IF**	**ID**	**EX**	**FEX**	**MEM**	**WB**	**FWB**
Cycles from Beg	0	1	3	7	4	5	7
Cycles from End	15	14	13	1	12	11	0

Table 5. Structural Hazard Information for the Paths in Figure 8.

Let n be the maximum number of iterations associated with a loop. The algorithm for estimating the worst-case execution time for a loop is shown in Figure 9. The algorithm contains three phases. During the first phase, the loop is analyzed one iteration at a time. For each iteration, the algorithm chooses the path with the greatest WCET. The first phase continues as long as new first miss instructions are encountered on each iteration. The WHILE loop in the algorithm represents this first phase, and it terminates when the number of calculated iterations reaches n - 1 or no more first misses (first hits) are encountered as misses (hits). Thus, the WHILE loop will iterate up to (n - 1) or (m + 1), where m is the number of paths in the loop since a first miss (first hit) can miss (hit) at most once during the loop execution. During the second phase of the algorithm, a longest path is calculated for all

```
struct loop_node *loop;
struct path_node *path, *chosen_path;

loop->first_misses_encountered = NULL.
loop->first_hits_encountered = NULL.
loop->wc_pipeline_information = NULL.
curr_iter = 0.
WHILE curr_iter != n - 1 DO
   curr_iter += 1.
   Invoke Time_Path() for all continue paths in loop->path_list.
   chosen_path = longest continue path for this iteration.
   Append first misses that were misses in chosen_path to loop->first_misses_encountered.
   Append first hits that were hits in chosen_path to loop->first_hits_encountered.
   For every continue path in loop->path_list,
      concatenate path->wc_pipeline_information with loop->wc_pipeline_information.
   IF no new first misses or first hits are encountered in chosen_path THEN
      BREAK.
Concatenate path->wc_pipeline_information with loop->wc_pipeline_information
   for all paths (n - 1 - curr_iter) times.
FOR each set of exit paths in loop->path_list that have a transition
   to a unique exit block in loop->exit_block_list DO
   Invoke Time_Path() for each path in the set.
   chosen_path = longest exit path in the set.
   Append first misses that were misses in chosen_path to loop->first_misses_encountered.
   Append first hits that were hits in chosen_path to loop->first_hits_encountered.
   Concatenate path->wc_pipeline_information with loop->wc_pipeline_information
      for all exit paths in the set.
   Store this information with this exit block in loop->exit_block_list.
```

Figure 9. Worst-Case Loop Analysis Algorithm.

the remaining iterations except the last iteration. In the third and final phase, the last iteration of the loop is handled separately. If the loop being analyzed has only one iteration, as is the case with a function, only this third phase is performed.

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 path for a given iteration of the loop will produce a longer worst-case time than that calculated by the algorithm. Since the pipeline effects of each of the paths within the loop are unioned, it only remains to be shown that the caching effects are treated properly. The instruction fetch 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 and the pipeline analysis was correct, 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 in the WHILE loop of the algorithm 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. An instruction classified as a first miss will be counted as a miss only the first time it is encountered within the WHILE loop of Figure 9. Because of this dual caching behavior of a first miss instruction, it is necessary to perform more than one pipeline analysis of a path since the caching behavior of the instructions comprising the path can change between iterations.

Once no more first miss instructions are encountered that miss, the pipeline effects associated with the path chosen will not change since the caching behavior of the instructions within a path will always be treated the same. The pipeline effects of the last chosen continue path are efficiently replicated for all but one of the remaining iterations. The last iteration of the loop is treated separately. 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 used for estimating the best-case execution time for a loop is somewhat simpler. Let n be the minimum number of iterations associated with a loop. Like the corresponding algorithm for worst case, the best-case loop analysis algorithm contains three phases. However, during the first phase, a shortest path is found only for the first iteration of the loop. The second phase of the algorithm determines the shortest path for the middle $n - 2$ iterations of the loop. The third phase finds the shortest exit path from the loop in the final iteration. The algorithm for estimating the BCET for a loop is shown in Figure 10.

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

```
struct loop_node *loop;
struct path_node *path, *chosen_path;

loop->bc_pipeline_information = NULL.
IF n > 1 THEN
   Invoke Time_Path() for all continue paths in loop->path_list,
      where all first misses are treated as misses and all first hits are treated as hits.
   chosen_path = shortest continue path for this iteration.
   For every continue path in loop->path_list,
      concatenate path->bc_pipeline_information with loop->bc_pipeline_information.
   Find the shortest continue path where all first misses are
      treated as hits and all first hits are treated as misses.
   Concatenate path->bc_pipeline_information with loop->bc_pipeline_information
      for all paths (n-2) times.
   For each set of exit paths that have a transition to a unique exit block DO
      Invoke Time_Path() for each path in the set.
      Find the shortest exit path in the set where all first misses
         are treated as hits and all first hits are treated as misses.
      Concatenate path->bc_pipeline_information with loop->bc_pipeline_information
         for all the exit paths in this set.
      Store this information with this exit block in loop->exit_block_list.
ELSE
   For each set of exit paths in loop->path_list that have a transition
      to a unique exit block in loop->exit_block_list DO
      Invoke Time_Path() for each path in the set.
      Find the shortest exit path in the set where all first misses
         are treated as misses and all first hits are treated as hits.
      Concatenate path->bc_pipeline_information with loop->bc_pipeline_information
         for all the exit paths in this set.
      Store this information with this exit block in loop->exit_block_list.
```

Figure 10. Best-Case Loop Analysis Algorithm.

produce a shorter best-case time than that calculated by the algorithm. The pipeline information for the first iteration is typically calculated within the IF-THEN portion (*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, the algorithm will calculate the shortest path for the first iteration. The shortest continue path will then be calculated 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. The pipeline information for the first iteration will be concatenated to the pipeline information calculated for the next n-2 iterations. The algorithm in Figure 10 examines the last iteration separately since paths associated with the exit blocks may be shorter than the shortest continue path. When the number of loop iterations is one (*i.e.* the loop is actually a function), first misses and first hits will be treated as misses and hits, respectively in the pipeline analysis of the exit path. Thus, the algorithm estimates a bound that is at least as small as the actual best-case bound.

It is important to note that the worst-case and best-case loop analysis algorithms are not perfectly analogous. Consider a loop having three paths with information depicted in Table 6. Paths 1 and 2 each have a distinct first miss instruction, while Path 3 has no first misses. According to the worst-

How Path Is Evaluated	Path 1	Path 2	Path 3
Treat first misses as misses	19	18	13
Treat first misses as hits	10	9	13

Table 6. Information about Hypothetical Loop with Three Paths

case loop analysis algorithm, the timing analyzer selects Path 1 for the first iteration, Path 2 for the second iteration, and Path 3 for all other iterations. For this example, the worst-case algorithm computes the WCET exactly for any number of loop iterations. For best case, Path 3 will be chosen for the first iteration. But starting with the second iteration, all first misses will be treated as hits, so Path 2 will be selected for all iterations after the first. Thus, the timing analyzer will compute a BCET of $13 + 9*(n - 1)$ cycles for this loop, where n is the minimum number of loop iterations.

However, the true BCET of this loop can be slightly greater. If the loop has just one iteration, the timing analyzer correctly predicts that Path 3 should be taken, and there is no underestimation in the BCET. If the loop has two iterations, then Path 3 should be taken for both iterations, yielding 26 cycles for the loop. The timing analyzer would compute 22 cycles if there are two iterations, a BCET underestimation of four cycles. On the other hand, if there are three or more iterations, the BCET is realized if the loop takes Path 2 for every iteration. In this case, the timing analyzer will underestimate the BCET of the loop by five cycles, and this underestimation is due to the incorrect prediction of which path had been chosen for the first iteration. In order to make an exact prediction in best case, it becomes necessary to re-examine path choices for prior iterations. We believe that having to re-examine all combinations of path choices for prior iterations to compute the BCET of a current iteration is overly inefficient. As a result, the best-case loop analysis algorithm shown in Figure 10 assumes that the same path will be taken during the middle iterations of the loop at the expense of a small underestimation in the total BCET.

5. Program Analysis

A timing analysis tree is constructed to predict the worst-case times of code segments containing nested loops and function calls. In the context of the notation in Figure 4, the root of this tree is an instance of `struct loop_node` representing `main()`. Each node of the tree represents either a loop or a function in the function instance graph. Each node is assumed to be a natural loop.[14] The nodes representing the outer level of function instances are treated as natural loops that will iterate only once when entered.

The loops in the timing analysis tree are processed in a bottom-up manner. In other words, the worst-case and best-case times for a loop are not calculated until the times for all of its immediate child loops are known. The algorithm given in the previous section described how a loop containing no other loops would be analyzed. The timing of a non-leaf loop is accomplished using this algorithm and the pipeline information and total times from its immediate child loops. 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. A unique set of timing information is stored for the child loop with each of these exit blocks. If a path within a loop enters a child loop, then the pipeline information and total time from the appropriate exit block are used at that point during the analysis of the path. For instance, if the loop in Figure 6 exits to block 5, then the last iteration of the loop will be shorter than if it had exited to block 7. Thus, the possible paths within non-leaf loops that contain child loops can also be calculated.[15]

The transition of an instruction categorization from the child loop level to the current loop level will be used to determine if any adjustment to the child loop time is required. The transitions between categorizations requiring adjustments are described in Table 7.

---

[14] 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.

[15] The timing analysis across loop levels is only briefly introduced in this section. It is described in more detail elsewhere [2, 4].

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.
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.

Table 7. Use of Child Loop Times.

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.[16] For instance, consider a program with two nested loops and each loop iterates 10 times. An instruction within both loops is classified as a *fm* at both the inner and outer loop levels. The instruction should miss only during the first iteration of the inner loop within the first iteration of the outer loop (1 miss, 99 hits). If no adjustment were made and the inner (child) loop pipeline information was used directly, then an overestimation would result since the analyzer would treat the instruction as initially missing for each iteration of the outer loop (10 misses, 90 hits). The *m=>fh* adjustment is necessary since the first reference to the instruction in the outer loop will be a hit. These same adjustments were used in previous work on bounding only instruction cache performance [4, 6].

Making these adjustments when pipelining is involved resulted in some slight mispredictions. The problem is that the caching behavior of a particular instruction depends on the loop level being analyzed. When a worst-case adjustment at an outer loop level would be needed for an instruction having a transition in Table 6, we conservatively added the maximum number of cycles associated with a cache miss penalty to the total time of the path containing the instruction and treated the instruction

[16] 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.

fetch as a cache hit within the path pipeline analysis for the inner loop. When the instruction fetch should be viewed as a cache hit at an outer loop level, the previously added miss penalty cycles were subtracted from the loop's time. This strategy permitted a single pipeline analysis of each loop, yet adjustments could still be made at outer levels of the program. A worst-case overestimation occurs when the instruction fetch is regarded as a miss and the cache miss penalty could have been overlapped with other pipeline delays (as shown in Figure 1).

For best-case estimations we treated the fetch of an instruction having a transition in Table 6 as a cache miss within the path pipeline analysis of the inner loop. When the instruction fetch should be viewed as a cache hit at an outer loop level, then the miss penalty will be subtracted from the total time of the path. If the miss penalty could be overlapped with some hazard (as shown in Figure 1), then an underestimation will result.

The timing analyzer could achieve an exact prediction by storing pipeline information about both cases (whether an instruction having such a instruction categorization transition between loop levels should be treated as a miss or a hit in the pipeline). There could be several instructions within a single loop having such caching categorization transitions between loop levels. Storing pipeline information about both cases for each instruction would result in an exponential space and complexity since all combinations of categorizations would have to be analyzed.

During best-case analysis, it is sometimes necessary to ignore a potential data hazard between a parent and child loop to avoid a potential overestimation in execution time. This situation can occur when a hazard is overlapped with some other delay (e.g. an instruction cache miss). The timing analyzer determines the number of cycles that a particular stage is vacant from the point it is first occupied to the point it is last occupied. If a data or structural hazard is detected for a particular stage between a parent and child loop, then the delay is reduced by number of vacant cycles for that stage in the child loop. If there were no vacant cycles, then the hazard could not be overlapped with other delays. This potential underestimation could be avoided by storing more information about the child loop. Again, this would result in increasing the complexity of the algorithm. A more detailed discussion about dealing with vacant cycles for best-case timing analysis is given elsewhere [8].

Fortunately, these adjustments are not that common. For instance, results indicated that only about 4.5% of the instructions within the function instance graph were classified as first misses or first hits and many of these did not require adjustments. Thus, these adjustments resulted in only small and relatively infrequent worst-case overestimations and best-case underestimations.

6. Results

Measurements were obtained on code generated for the SPARC architecture by the *vpo* optimizing compiler [9]. Six simple programs described in Table 8 were used to assess the effectiveness of the timing analyzer. A direct-mapped instruction cache configuration containing 8 lines of 16 bytes was used. Thus, the cache contained 128 bytes of instructions. 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 is often 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 2 of Table 8. 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. Column 4 shows the worst-case hit ratio of each program. Only *Matmul* had a very high ratio due to three tightly nested loops in a single function to perform the matrix multiplication.

The results of evaluating these programs are shown in Table 9. For each of the six modes of timing analysis, four values are given for each test program. The first value is the Observed Cycles, which represents the actual duration of executing the program. The second value is the Estimated Cycles, which is the timing analyzer's predicted WCET/BCET of the program. The next value, the Estimated Ratio, is the ratio of the estimated cycles to the observed cycles. This is a measure of how accurate the timing analysis is. A perfect prediction would result in a ratio of 1. The last value given is the Naive Ratio, which is what the estimated ratio would have been if the analysis had not been performed.

The observed cycles for these measurements were obtained by enhancing the *Ease* cache simulator [10]. This simulator produced the *pipeline only observed* cycles and the timing analyzer produced the *pipeline only estimated* cycles by assuming that all instruction fetches (IF stages) were cache hits and only required a single cycle. The *pipeline only worst-case naive* cycles were obtained by assuming

Name	Num Bytes	Num Func	Hit Ratio	Description or Emphasis
Des	2,240	5	81.41%	Encrypts and Decrypts 64 Bits
Matcnt	812	8	81.81%	Counts and Sums Nonnegative Values in a 100x100 Integer Matrix
Matmul	768	7	99.24%	Multiplies Two 50x50 Integer Matrices
Matsum	644	7	88.22%	Sums Nonnegative Values in a 100x100 Integer Matrix
Sort	556	5	83.99%	Bubblesort Array of 500 Integers into Ascending Order
Stats	1,428	9	88.41%	Std. Dev. & Corr. Coef. of Two Arrays of 1000 Floating-point Values

Table 8. Test Programs.

that only a single pipeline stage could be executing at one time (*i.e.* no overlap). The *caching only observed* cycles and *caching only estimated* cycles were obtained with the assumption that the pipeline had only a single stage (an IF), a cache hit required a single cycle, and a cache miss required an additional miss penalty of nine cycles. The *caching only worst-case naive* cycles were calculated by assuming every instruction fetch resulted in a cache miss. The *pipeline and caching estimated* cycles were produced by the techniques that were described in this paper for integrating the analysis of pipelining and instruction caching behavior. The *best-case pipeline and caching naive* cycles were obtained by assuming that each instruction required only a single cycle. All data cache references were assumed to be hits in the three sets of measurements.

Analysis	Worst-Case				Best-Case			
Pipeline Only	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	66,594	68,254	1.02	3.82	34,837	15,684	0.45	0.36
Matcnt	1,063,572	1,063,572	1.00	2.38	1,013,307	1,013,207	1.00	0.38
Matmul	4,347,806	4,347,806	1.00	2.13	4,347,541	4,347,541	1.00	0.33
Matsum	933,540	933,540	1.00	2.28	913,275	913,175	1.00	0.35
Sort	3,380,660	6,748,925	2.00	8.13	11,158	4,174	0.37	0.32
Stats	900,231	900,231	1.00	1.70	447,478	447,477	1.00	0.41
Caching Only	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	142,956	163,015	1.14	3.86	59,998	19,345	0.32	0.21
Matcnt	1,169,055	1,259,055	1.08	3.79	929,073	929,073	1.00	0.41
Matmul	1,527,648	1,527,648	1.00	9.36	1,527,648	1,527,648	1.00	0.94
Matsum	707,219	707,219	1.00	4.85	687,219	687,219	1.00	0.47
Sort	7,639,611	15,253,902	2.00	8.17	10,439	3,901	0.37	0.35
Stats	372,410	372,410	1.00	4.90	372,410	372,410	1.00	0.49
Pipeline & Caching	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	149,706	169,613	1.13	5.02	65,615	22,247	0.34	0.19
Matcnt	1,769,321	1,859,323	1.05	3.69	1,549,095	1,548,798	1.00	0.25
Matmul	4,444,911	4,445,413	1.00	4.98	4,444,666	4,420,068	0.99	0.32
Matsum	1,277,465	1,277,477	1.00	4.08	1,257,239	1,157,240	0.92	0.26
Sort	7,765,125	15,504,172	2.00	10.78	19,957	4,428	0.22	0.18
Stats	1,016,048	1,016,145	1.00	3.12	607,399	601,406	0.99	0.30

Table 9. Results for the Test Programs.

The *worst-case pipelining only* timing analysis had exact predictions for all programs except *Des* and *Sort*. The analysis of these two programs 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 deemed feasible by the timing analyzer 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. Note that both of these problems are encountered by other timing tools and are not directly related to the pipeline analysis.

The *best-case pipeline only* timing analysis resulted in exact predictions for *Matmul* and *Stats*. The predictions for *Matcnt* and *Matsum* were slightly underestimated due to diminishing the effect of data hazard because of vacant cycles within a child loop. Even though *Matmul* has no conditional control flow, its BCET is less than its WCET because the integer multiply instruction `smul` can spend 1-19 cycles in the EX stage. Floating-point instructions also take a varying time to execute, which can result in a WCET that is significantly greater than the corresponding BCET. The best-case predictions for *Des* and *Sort* were substantially underestimated for the same reasons they were overestimated in the worst-case analysis.

The worst-case and best-case *caching only* timing analysis results were also quite accurate. This analysis had exact predictions for *Matmul*, *Matsum*, and *Stats* since there were few conditional constructs except to exit loops. The *Matcnt* program used an `if-then-else` construct 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, which was purposely 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 instruction caching performance assumes 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 need not be calculated. Thus, one reference was counted repeatedly as a miss instead of a hit in the worst-case analysis. This path was executed 10,000 times and accounted for a 90,000 cycle [10,000*miss penalty] or an 8% overestimation. Note that the execution of this single path accounted for 40.61% of the total instructions referenced during the program execution. The best-case analysis for *Matcnt* was exact since the shorter path did not contain the call to add a nonnegative value. The programs *Des* and *Sort* had overestimations for the worst-case predictions and underestimations for the best-case predictions due to the same problems described previously for the *pipeline only* measurements. The worst-case naive ratio was lower than initially anticipated by the authors. These test programs contained many long running instructions (floating-point operations and integer multiply and divides) that were frequently executed and often resulted in stalls. In addition, transfers of control were also quite frequent and were only considered to require two pipeline stages in our analysis.

The integrated *pipeline and caching* worst-case analysis also resulted in quite tight predictions.

Again the predictions for the programs *Matmul*, *Matsum*, and *Stats* were very accurate. Note that the estimated worst-case cycles were slightly greater than the observed cycles for these programs. This overestimation was due to the problem of an instruction's caching behavior changing between loop levels. These changes require an adjustment as shown in Table 6. The approach used by the authors was to treat such an instruction as a hit in the pipeline analysis and simply add the miss penalty to the total time. When the instruction should be viewed as a hit at an outer level, then this miss penalty was simply subtracted and an accurate estimation is obtained. However, in these three programs the potential overlap between a miss penalty and a stall due to a hazard were not always detected.[17] The *Des*, *Matcnt*, and *Sort* programs had its usual worst-case overestimations due to data dependencies, a cache conflict, and an inaccurate number of estimated loop iterations, respectively. The naive ratio indicates that much tighter WCET bounds can be obtained when the benefits of pipelining and instruction caching are analyzed.

The integrated *pipeline and caching* best-case analysis for the four programs (*Matcnt*, *Matmul*, *Matsum*, and *Stats*) without data dependency or loop iteration problems was within 8% of the observed cycles. The underestimations were largely due to inaccuracies resulting from a *fm=>fm* transition between inner and outer loops. The timing analyzer treats the instruction in this case as a miss in the pipeline best-case analysis and subtracts the miss penalty from the time of the path when the instruction will be viewed as a hit. Thus, if a portion of the miss penalty can be overlapped with a delay due to a data hazard, an underestimation will occur on each iteration except the first. In contrast, the worst-case analysis would treat the instruction as a hit in the pipeline analysis and only overestimate in a similar situation on the first iteration of the loop when the instruction reference was

[17] For instance, the 502 cycle overestimation in *Matmul* occurred from 50 miss penalties completely overlapping with stalls from an integer multiply instruction and 52 misses overlapping with one cycle load hazards.

regarded as a miss. In addition, some of the underestimation in the best-case analysis was from disregarding data hazard stall cycles between a parent and a child loop due to subtracting vacant cycles from the stall. Thus, it was common to have a larger underestimation in best-case analysis than an overestimation in worst-case analysis. Fortunately, most timing constraints are associated with meeting deadlines, which requires worst-case analysis, instead of finishing a task too soon, which would require best-case analysis. The other two programs (*Des* and *Sort*) were significantly underestimated due to data dependencies and loop iteration problems discussed previously.

If the *pipeline and caching* analysis had been handled independently, then the cache miss penalty would not have the opportunity to overlap with a pipeline stall, as shown in Figure 1. Thus, one would anticipate a greater overestimation in predicting WCET with an independent analysis approach. The effect of an independent analysis strategy would be to add the cache miss penalty to the total time of a path when an instruction fetch is predicted to be a miss and treat the instruction as a hit in the pipeline. The benefit of integrating the pipeline and instruction cache worst-case analysis is depicted in Table 10. Without an integrated analysis, the test programs would have been overestimated by an additional 3% on average. Note that the most significant effect was on the worst-case prediction of *Stats*, which was the only floating-point intensive test program. Programs requiring floating-point operations result in more frequent and lengthy delays that may sometimes be overlapped with instruction cache misses or any other source of multicycle pipeline stage occupation. Thus, the benefit of using an integrated analysis approach would be more pronounced in floating-point intensive programs.

Name	Estimated Ratio with Integrated Analysis	Estimated Ratio with Independent Analysis
Des	1.133	1.174
Matcnt	1.051	1.057
Matmul	1.000	1.000
Matsum	1.000	1.016
Sort	1.997	2.029
Stats	1.000	1.082
Average	1.197	1.226

Table 10. Ratios for Integrated versus Independent Worst-Case Analysis

7. User Interface

Once the initial timing analysis has been completed, a graphical user interface is invoked that is depicted in Figure 11. The main window on the left allows the user to quickly request timing predictions for functions, loops, paths, subpaths, or ranges of machine instructions and reports the

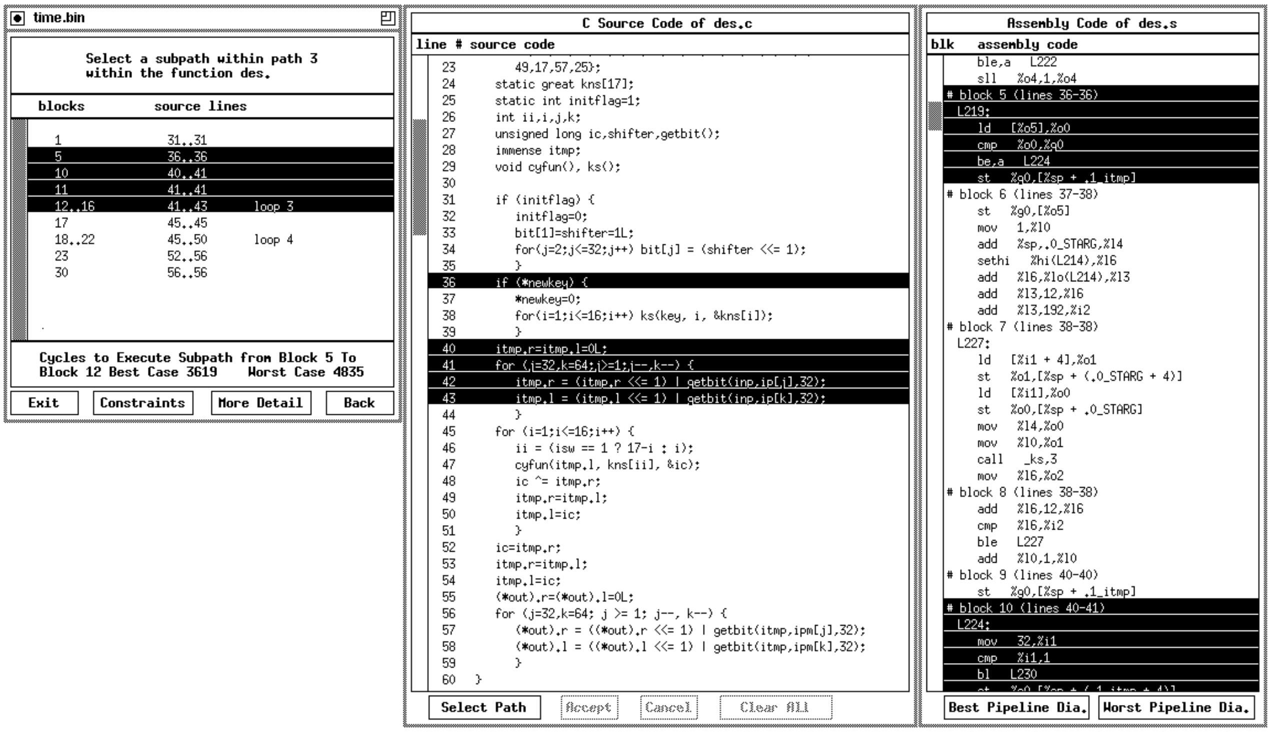

Figure 11. Timing Analyzer User Interface

associated timing predictions. The middle window depicts the C source code and the right window depicts the corresponding assembly code. Whenever a different construct is selected in the main window, the highlighted lines in the source and assembly windows are automatically updated and scrolled to the appropriate position. Note the source lines within the middle window are numbered. This allows the user to identify constructs that are referenced by line numbers within the main window and to correlate the source line ranges associated with each basic block depicted in the assembly code window. Selection of paths via the mouse on the source window is also supported. Since there may be more than one instance of a function within the timing analysis tree, the user interface displays the worst-case and best-case times from all of the instances of the construct associated with the user request. Whenever a different construct is selected, the highlighted lines in windows containing the source and assembly code are automatically updated and scrolled to the appropriate position. Thus, the user can quickly observe the relationship between timing constraints associated with the source code and sequences of machine instructions. This interface is described in more detail elsewhere [11].

8. Comparison with Previous Work

There has been much work on the issue of predicting execution time of programs. However, most approaches in the past have not dealt with the effects of pipelining and instruction caching [12, 13, 14]. There have also been some recent studies on predicting pipeline performance by Harmon *et al.* [15] and Narasimhan and Nilsen [16]. Yet, these studies did not address caching issues.[18] Furthermore, the former study was limited to nonnested functions and the latter study required the sequence of executed instructions to be known. Finally, there has been some recent work on predicting

[18] Harmon assumed the entire code segment would fit into cache. Thus, at most one miss could occur for each instruction reference.

instruction caching performance. Arnold *et al.* [4] implemented a timing analysis system to tightly bound instruction cache performance. However, this approach did not address pipelining issues.

Li *et al.* [17, 18] used an integer linear programming (ILP) approach to model instruction caching behavior. Their approach is also used to predict data and set-associative caching behavior [19]. The authors automatically derived constraints from a program's control-flow graph that could be solved using ILP. Additional user-provided constraints regarding data dependencies within the control flow can be easily integrated into the analysis. In their control-flow analysis, each set of instructions within a basic block mapping to the same cache line was identified as a line-block. Three possible states were 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 it, 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.

Apparently, the pipeline behavior was not modeled and it is unclear how well Li's approach will work when pipelining is addressed. However, it is possible that pipeline behavior for instructions within a single basic block can be modeled with Li's ILP approach. By performing no general pipeline analysis, this allowed their approach to disregard the potential effects of different paths on pipeline behavior. Thus, they had only two possible times for the instructions within a line-block, one with an instruction cache miss and one without a miss. Unfortunately, the state of the pipeline can affect the execution time associated with a sequence of instructions. Thus, there was also no method shown for detecting pipeline stalls or potential overlap between stalls and cache misses.

There has been only one previous study that attempted to address the issue of predicting the WCET of programs on machines with both pipelining and an instruction cache. Lim *et al.* [20] described a method of predicting the performance of pipelining and instruction caching, which is based on an extension of a previous timing tool [21]. They have also extended this tool to address data caching as well [22]. It has been proposed that the Lim approach can be extended to analyze set-associative caching behavior as well. Lim's method differs quite significantly from our approach described in this paper, which instead builds on flow analysis techniques found in optimizing compilers. Lim's method uses a timing schema associated with each source-level language program construct. They stored information about the number of cycles at the head and tail of a reservation table produced as a result of the pipeline analysis on the instructions associated with a program construct. In addition, this method stored information about the set of memory blocks whose first reference depends upon the cache contents prior to the execution of the construct. Lim also stored the set of memory blocks known to remain in cache after the execution of the construct. Eventually, this timing information is concatenated with another construct that would be executed immediately before the current construct. Their timing analyzer attempted to overlap the head of the reservation table of the current construct with the tail of the reservation table of the other construct as much as possible. Their *row-based* approach of concatenating reservation tables is equivalent to our tables of structural and data hazard information depicted in Tables 3 and 4. Likewise, the list of memory blocks known to be in cache after executing the other construct is used to adjust the time of the current construct by comparing this list to the list of first reference blocks in the current construct. This method stored multiple paths for conditional constructs, such as an `if-then-else`. They pruned or eliminated a particular path when it was found that the worst-case execution time of the path was faster than the best-case execution time of another path within the same construct.

The approach that Lim *et al.* used to analyze caching behavior limits the accuracy of the analysis. They used a single bottom-up pass when performing the timing analysis of a program. The caching behavior of a large percentage of the instruction fetches within a construct would be unknown until many of the surrounding constructs were processed. Their approach was to treat the instruction fetch as a hit within the pipeline and add the cycles associated with a cache miss penalty to the total time of the construct. When it was later found that an instruction reference was a hit, they would subtract the miss penalty from the total time. However, an overestimation may result when the instruction is not found in cache. As shown in Figure 1, the instruction fetch miss penalty of one instruction (instruction 2) can be completely hidden by a stall with a long running instruction (data hazard stall on instruction 3). Whether the fetch of instruction 2 was a hit or a miss would have no effect on the total number of cycles. The Lim method would rarely detect instruction fetches that would always be misses until the surrounding constructs are analyzed, which is after the pipeline analysis of a construct has already occurred. Our approach of categorizing the caching behavior of each instruction before starting the timing analysis allows the detection of such situations. For instance, about 25% of the instructions within the function instance graphs of the programs we evaluated were statically categorized as *always misses*. As Table 10 above indicates, we found that the *pipeline and caching* estimated ratio for the six test programs increased on average by about 3% when the complete miss penalty was always added for each predicted miss.

9. Future Work

We are working on several enhancements to the timing analyzer. We plan to automate the detection of many data dependencies using existing compiler optimization techniques to obtain tighter performance estimations [23]. We also plan to accurately calculate the number of iterations for loops which are dependent on the value of a loop counter variable of an outer loop. The retargetability of the timing analyzer will also be enhanced by isolating any remaining machine dependent information in data files.

We are exploring methods to predict the timing of other architectural features associated with RISC processors. Work is currently ongoing to verify that our technique accurately predicts performance for the MicroSPARC I by using a logic analyzer. This will require predicting the performance of other features, such as wrap-around filling of cache lines. 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 even when the addresses may differ for different invocations of the same function. 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. We are also currently working on extending the timing analyzer to predict the performance of set-associative caches.

10. Conclusions

This paper has presented a technique for predicting the worst and best-case execution time of programs on machines with pipelining and instruction caches. 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 these instruction categorizations when analyzing the pipeline performance of a path of instructions. Third, the timing analyzer uses a concise representation of the pipeline information to concatenate the performance of paths in an efficient manner when predicting the performance of loops. Fourth, a timing analysis tree is used to predict the performance of an entire program. Finally, a graphical user interface has been implemented that allows users to obtain timing predictions of portions of the program. The results indicate that the timing analyzer can quickly obtain tight predictions of performance.

11. Acknowledgements

Lo Ko and Emily Ratliff implemented the user interface. We are also grateful to the anonymous referees who provided helpful suggestions that improved the quality of the paper.

12. References

[1] Texas Instruments, Inc., *Product Preview of the TMS390S10 Integrated SPARC Processor*, 1993.

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

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

[4] R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon, “Bounding Worst-Case Instruction Cache Performance,” *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, pp. 172-181 (December 1994).

[5] F. Mueller and D. B. Whalley, “Fast Instruction Cache Analysis via Static Cache Simulation,” *Proceedings of the 28th Annual Simulation Symposium*, pp. 105-114 (April 1995).

[6] 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*, pp. 288-297 (December 1995).

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

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

[9] 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).

[10] 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).

[11] L. Ko, D. B. Whalley, and M. G. Harmon, “Supporting User-Friendly Analysis of Timing Constraints,” *Proceedings of the ACM SIGPLAN Notices 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems* **30**(11) pp. 99-107 (November 1995).

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

[13] 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).

[14] P. Puschner and C. Koza, “Calculating the Maximum Execution Time of Real-Time Programs,” *Real-Time Systems* **1**(2) pp. 159-176 (September 1989).

[15] 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).

[16] K. Narasimhan and K. D. Nilsen, “Portable Execution Time Analysis for RISC Processors,” *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, (June 1994).

[17] 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).

[18] Y. S. Li, S. Malik, and A. Wolfe, “Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software,” *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 298-307 (December 1995).

[19] Y. S. Li, S. Malik, and A. Wolfe, “Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches,” *Proceedings of the Seventeenth IEEE Real-Time Systems Symposium*, (December 1996).

[20] 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).

[21] A. C. Shaw, “Reasoning about Time in Higher-Level Language Software,” *IEEE Transactions on Software Engineering* **15**(7) pp. 875-889 (July 1989).

[22] S.-K. Kim, S. L. Min, and R. Ha, “Efficient Worst Case Timing Analysis of Data Caching,” *Proceedings of the 1996 Real-Time Technology and Applications Symposium*, pp. 230-240 (June 1996).

[23] F. Mueller and D. B. Whalley, “Avoiding Conditional Branches by Code Replication,” *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation*, pp. 56-66 (June 1995).

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

Appendix

In the following, the informal description on instruction categorization of section 2 will be formalized. The categorization for direct-mapped instruction caches is based on the following definitions:

Definition 1 (Potentially Cached) *A program line l can potentially be cached if there exists a sequence of transitions in the combined control-flow graphs and function-instance graph such that l is cached when it is reached in the current block.*

The traversal of every possible sequence of blocks leads to an exponential explosion. We avoid this overhead by restricting the analysis to abstract cache states:

Definition 2 (Abstract Cache State (ACS)) *The abstract cache state of a program line l within a block and a function instance is the set of program lines that can potentially be cached prior to the execution of l within the block and the function instance.*

Given the control-flow information of a program and a cache configuration, the ACSs for each block have to be calculated. Using data-flow analysis (DFA), each block has an input state and an output state, corresponding to the ACS before and after the execution of the block, respectively. An iterative algorithm for the calculation of ACS' via DFA is given in Figure 3. The DFA requires a time overhead comparable to that of inter-procedural DFA performed in optimizing compilers. The space overhead is $O(pl * bb * fi)$, where pl, bb, fi denote the number of program lines, basic blocks, and function instances, respectively. The correctness of iterative DFA has been discussed elsewhere [24]. Additional DFA is required to determine the linear cache state and the post-dominator set for each block before a definition for instruction categories can be specified.

Definition 3 (Linear Cache State (LCS)) *The linear cache state of a program line l within a block and a function instance is the set of program lines that can potentially be cached in the forward control-flow graph prior to the execution of l within the block and the function instance.*

The forward control-flow graph is the acyclic graph resulting from the removal of back edges (backwards edges forming loops, see Figure 5 and [24]) in the regular control-flow graph. Informally, the LCS represents the hypothetical cache state in the absence of loops. It will be used to determine whether a program line may be cached due to loops or due to the sequential control flow.

Definition 4 (Post-dominator Set) *The post-dominator set of a program line l within a block and a function instance is the self-reflexive transitive closure of post-dominating program lines.*

Informally, the post-dominator set describes the program lines certain to be reached from the current block, regardless of the taken paths in the control flow. A more detailed discussion of post dominators can be found elsewhere [24]. The instruction categories can now be defined with respect to DFA. The following definition formalizes the worst-case instruction categories for each loop level.

Definition 5 (Instruction Categorization) :

- Let i_k be an instruction within a block, a loop λ, and a function instance.
- Let $l = i_0..i_{m-1}$ be the program line containing i_k and let i_{first} be the first instruction of l within the block.
- Let s be the ACS for l within the block.
- Let l map into cache line c, denoted by $l \to c$.
- Let u be the set of program lines in loop λ.
- Let $child(\lambda)$ be the child loop (inner-next loop within nesting) of λ for this block and function instance, if such a child loop exists.

- Let $header(\lambda)$ be the set of header blocks and $preheader(\lambda)$ be the set of preheader blocks of loop λ, respectively.[19]
- Let $s(p)$ be the abstract output cache state of block p.
- Let $linear$ be the LCS for l within the block.
- Let $postdom(p)$ be the set of self-reflexive post-dominating programming lines of block p.

Then,

$$WCET\text{-}category\ (i_k, \lambda) = \begin{cases} always\text{-}hit & if\ k \neq first \vee (l \in s \wedge \forall_{m \rightarrow c, m \neq l}\ m \notin s) \\ first\text{-}hit & if\ worst(i_k, child(\lambda)) = first\text{-}hit \vee \\ & \quad k = first \wedge l \in s \wedge \exists_{m \rightarrow c, m \neq l}\ m \in (s \cap u) \wedge \\ & \quad [\forall_{p \in preheaders(\lambda)}\ l \in s(p) \wedge \forall_{m \rightarrow c, m \neq l}\ m \notin (s(p) \cap u)] \wedge \\ & \quad \forall_{p \in headers(\lambda)}\ l \in postdom(p) \wedge \forall_{m \rightarrow c, m \neq l}\ m \notin (linear \cap u) \\ first\text{-}miss & if\ worst(i_k, child(\lambda)) = first\text{-}miss \wedge k = first \wedge l \in s \wedge \\ & \quad \exists_{m \rightarrow c, m \neq l}\ m \in s \wedge \forall_{m \rightarrow c, m \neq l}\ m \notin (s \cap u) \\ always\text{-}miss & otherwise \end{cases}$$

$$BCET\text{-}category\ (i_k, \lambda) = \begin{cases} always\text{-}miss & if\ k = first \wedge l \notin s \\ first\text{-}hit & if\ best(i_k, child(\lambda)) = first\text{-}hit \vee \\ & \quad k = first \wedge l \in s \wedge \exists_{m \rightarrow c, m \neq l}\ m \in (s \cap u) \wedge \\ & \quad \forall_{p \in preheaders(\lambda)}\ l \in s(p) \wedge \\ & \quad \forall_{p \in headers(\lambda)}\ l \in postdom(p) \wedge \forall_{b \in backedges(\lambda)}\ l \notin s(b) \\ first\text{-}miss & if\ best(i_k, child(\lambda)) \in \{first\text{-}miss, always\text{-}hit\} \wedge \\ & \quad k = first \wedge l \in s \wedge l \notin linear \\ always\text{-}hit & otherwise \end{cases}$$

While the definition seems complex, it can be implemented rather efficiently once DFA has been performed. First, simple set operations on bit vectors suffice to test the conditions. Second, if one conjunct in a condition fails, the remaining ones are not tested. Third, the implementation orders the conjuncts such that the least likely ones are tested first. The informal description in Section 2 describes each conjunct of the above definition verbally and may be used as a reference to further motivate the formal definition.

[19] The common notion of "natural loops" defines a loop to have only a single header [24]. This work extends this notion to handle more general control flow with unstructured loops. Multiple loop headers occur only for unstructured loops, which are handled by the simulator. Multiple loop preheaders occur when the loop can be entered from more than one block outside the loop, which can occur even for natural loops.