Predicting Instruction Cache Behavior [*]

Frank Mueller, David B. Whalley
Dept. of Computer Science, B-173
Florida State University
Tallahassee, FL 32306-4019

Marion Harmon
Dept. of Computer and Information Systems
Florida A & M University
Tallahassee, FL 32307

e-mail: whalley@cs.fsu.edu *phone: (904) 644-3506*

Abstract

It has been claimed that the execution time of a program can often be predicted more accurately on an uncached system than on a system with cache memory [5, 20]. Thus, caches are often disabled for critical real-time tasks to ensure the predictability required for scheduling analysis. This work shows that instruction caching can be exploited to gain execution speed without sacrificing predictability. A new method called *Static Cache Simulation* is introduced which uses control-flow information provided by the back-end of a compiler. This simulator statically predicts the caching behavior of a large portion of the instruction cache references of a program. In addition, a *fetch-from-memory* bit is added to the instruction encoding which indicates whether an instruction shall be fetched from the instruction cache or from main memory. This bit-encoding approach provides a significant speedup in execution time (factor 3-8) over systems with a disabled instruction cache without any sacrifice in the predictability of worst-case execution time. Even without bit-encoding, the ability to predict the caching behavior of a large percentage of the instruction references is very useful for providing tight worst-case execution time predictions of large code segments on machines with instruction caches.

1 Introduction

Predicting the execution time of programs or code segments is a difficult task. Yet, in the context of hard real-time systems, it is essential to provide a schedule for tasks with known deadlines. Thus, tasks have to be analyzed to determine their best-case execution time (BET) and worst-case execution time (WET). The following problems have to be addressed to predict the execution time of a task or program:

- The number of loop iterations needs to be known prior to execution. It is often required that the maximum number of iterations is provided by the programmer [11].
- The possible execution paths in the control flow have to be analyzed to predict both BET and WET.
- Architectural features have to be taken into account (*e.g.* pipeline stalls).

Cache memories have become a major factor to bridge the bottleneck between the time to access main memory and the faster clock rate of current processors. In the context of real-time systems, caches have been regarded as a source for unpredictability which conflicts with the goal of making the execution of tasks deterministic [20]. For a system with an instruction cache as a primary (on-chip) cache, the execution time of an instruction can vary greatly depending on whether the given instruction is in cache or not. In addition, context switches and interrupts may replace the instructions cached by one task with instructions from another task or an interrupt handler. As a result, it has been common practice to simply disable the cache for sections of code when predictability was required [20].

This work shows that it is possible to predict some cache behavior with certain restrictions. Let a task be the portion of code executed between two scheduling points (context switches). When a task starts execution, the cache memory is assumed to be invalidated. During task execution, instructions are gradually brought into cache and often result in many hits and misses which can be predicted by Static Cache Simulation, a technique which analyzes control flow prior to execution time. Furthermore, a slight change in the architecture in conjunction with the simulator's analysis allows, without loss of predictability, significantly faster execution time than on system with a disabled instruction cache.

This paper is structured as follows: Section 2 reviews related work in the area. Section 3 introduces the method of Static Cache Simulation. Section 4 details a bit-encoding approach which can exploit caches for real-time systems. Section 5 provides a quantitative analysis of both Static Cache Simulation and the

[*]1st revision June 26, 1994. This work was supported in part by the Office of Naval Research under contract # N00014-94-1-0006

bit-encoding approach. Section 6 outlines future work and section 7 presents the conclusions of this study.

2 Related Work

The problem of determining the execution time of programs has been the subject of some research in the past. Sarkar [19] suggested a framework to determine both average execution time and its variance. His work was based on the analysis of a program's interval structure and its forward control flow. He calculated a program's execution time for a specific set of input data by using a description of the architecture and the frequency information obtained by incrementing counters during a profiling run. He assumed that the execution order of instructions does not affect this calculation. Thus, his method cannot capture the impact of caching on execution time.

For real-time systems, several tools to predict the execution time of programs have been designed. The analysis has been performed at the level of source code [18], at the level of intermediate code [16], and at the level of machine code [6]. Only Harmon's tool took the impact of instruction caches into account for restrictive circumstances, *i.e.* only for small code segments which entirely fit into cache.

Niehaus outlined how the effects of caching can be taken into account in the prediction of execution time [17]. He suggested that caches be flushed on context switches to provide a consistent cache state at the beginning of each task execution. He provided a rough estimate of the benefit of caches for speedup and tried to determine the percentage of instruction cache references which can be predicted as hits. The level of analysis remained at a very abstract level though as it only dealt with spatial locality for sequential execution and some temporal locality for simple loops. No general method to analyze the call graph of a task and control flow for each function was given.

A few attempts have been made to improve on the predictability of caches by architectural modifications to meet the needs of real-time systems. Kirk [10] outlined such a system which relied on the ability to segment the cache memory into a number of dedicated partitions, each of which can only be accessed by a dedicated task. But this approach introduced new problems such as exhibiting lower hit ratios due to the partitioning and increasing the complexity of scheduling analysis by introducing another resource (cache partitioning) as an additional degree of freedom in the allocation process.

Other suggested architectural modifications often dedicate a bit in the instruction encoding which is used by the compiler to affect the cache behavior. McFarling [12] used such an approach to exclude instructions from cache that were not likely to be in cache on subsequent references. Chi and Dietz [4] introduced a data cache bypass bit on load and store instructions which, when set, indicates that the processor should go directly to memory (without caching the value as a side-effect) or goes to the cache when clear. Their idea is to improve execution speed by keeping data values either in registers or in cache, thus avoiding storage mirroring among the fasted components in the memory hierarchy (registers and data caches). Our work emphasizes instruction caches rather than data caches. In contrast to McFarling's study and the work by Chi and Dietz, we are primarily concerned about the predictability of instruction caching and secondarily about execution speed.

3 Static Cache Simulation

The method of Static Cache Simulation can be used to statically predict the behavior of a large number of the instruction cache references for a given program/task with a specific cache configuration. Unlike many data references, the address of each instruction is known statically. This is certainly true for code which is physically locked into memory. It also holds for virtual memory mapping if and only if the page size is an integer multiple of the instruction cache size, which is typical for many systems [7]. In this case, the relocation of a virtual page would not affect the mapping of program lines into cache lines.

Figure 1 depicts an overview of the tools and interfaces involved in instruction cache analysis using Static Cache Simulation. The set of source files of a program

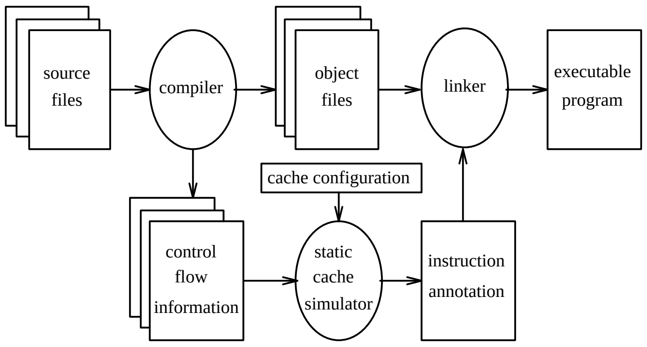


Figure 1: Overview of Static Cache Simulation

are translated by a compiler. The compiler generates object code and passes information about the control flow of each source file to the static cache simulator. The static cache simulator performs the task of determining which instruction references can be predicted prior to execution time. It constructs the call graph of the program and the control-flow graph of each function based on the information provided by the compiler.

The cache behavior is then simulated for a given cache configuration. Furthermore, the static simulator produces instruction annotations and passes them to the linker which modifies the object code according to the annotations and creates an executable program.

The task of Static Cache Simulation is to determine whether each instruction reference will result in a cache hit or miss during program execution. This is done by analyzing the call graph and the control flow for each function. Since it is not always possible to determine if a reference is a hit or miss, instructions are classified into categories of *always-hit, always-miss, first-miss,* or *conflict.* If an instruction is always (never) in cache, then it is denoted as an always-hit (always-miss). If an access to an instruction results in a miss on the first access and in hits for any subsequent accesses, then it is classified as a first-miss. If an access to a program line results in either a hit or a misses depending on the flow of control, then it is referred to as a conflict.

The following subsections describe this process in more detail. A formal approach to Static Cache Simulation can be found elsewhere [13].

3.1 Decomposition

To statically determine a program's or task's cache behavior as accurately as possible, the program/task is decomposed into smaller components. A program/task may be composed of a number of functions[1]. The possible sequence of calls between these functions is depicted in a call graph. Each function can be represented by a control-flow graph where nodes are basic blocks[2] and edges denote legal transitions of the control flow between basic blocks.

Functions are further distinguished by function instances. An instance depends on the call sequence, that is, it depends on the immediate call site within its caller as well as the caller's call site, etc. The instance i of a function corresponds to the ith occurrence of the function within a depth-first traversal of the call graph. Thus, a directed acyclic call graph (without recursion) is transformed into a tree of function instances.

3.2 Instruction Categorization

Static Cache Simulation calculates the abstract cache states associated with basic blocks. The calculation is performed by repeated traversal of the call graph's functions, their function instances, and the basic blocks of each function's control-flow graph.

[1]We will use the term function rather than procedure, subroutine, subprogram, or other equivalent notions.

[2]A basic block is a sequence of instructions where only the first instruction may be preceded by a label and only the last instruction may be a transfer of control.

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

Definition 2 *The* **abstract cache state** *of a basic block b in a function instance is the subset of program lines which can potentially be cached prior to the execution of b.*

The notion of an abstract cache state is a compromise between a feasible storage complexity of the proposed method and the alternative of an exhaustive set of all cache states which may occur at execution time with an exponential storage complexity.

Definition 3 *The* **reaching state** *of a basic block b in a function instance is the subset of program lines which can be reached through control-flow transitions from b.*

For a given function instance, each instruction i within a basic block b is categorized based on its position in the corresponding program line $l = i_0..i_{n-1}$, on the corresponding abstract cache state s, and on the reaching state t. The program line l maps into cache line c, denoted by $l \rightarrow c$.

always-miss: A cache miss is predicted if

- $i = i_0$: instruction i is the first reference to program line l in b and
- $l \notin s$: l is not in the abstract cache state.

always-hit: A cache hit is predicted if

- $i \in \{i_1..i_{n-1}\}$: instruction i is not the first reference to program line l in b. Or
-
  - $i = i_0$: instruction i is the first reference to program line l in b,
  - $l \in s$: l is in the abstract cache state, and
  - $\forall_{m \rightarrow c, m \neq l} \; m \notin s$: no other line m (which maps into the same cache line as l) is in the abstract cache state.

first-miss: A miss on the first reference and hits for consecutive references is predicted if

- $i = i_0$: instruction i is the first reference to program line l in b,
- $l \in s$: l is in the abstract cache state,
- $\exists_{m \rightarrow c, m \neq l} \; m \in s$: another line m (which maps into the same cache line as l) is also in the abstract cache state,

- $\forall_{m \rightarrow c, m \neq l} m \in s \Rightarrow m \notin t$: all other program lines m (which maps into the same cache line as l) cannot be reached anymore if they are in the abstract cache state, and
- $\forall_{0 \leq x \leq n-1} category(i_x) \in$ {always-hit, first-miss}: all other instructions of program line l are either always hits or first misses.[3]

conflict: A reference may result in either a cache hit or cache miss at execution time in all other cases.

The categorization can be used to statically infer non-trivial caching behavior as will be shown in the next subsection.

3.3 Implementation

The iterative algorithm in Figure 2 was used to calculate the abstract cache states. Each basic block has

```
input_state(top) := all invalid lines;
WHILE any change DO
   FOR each instance of a basic block B
                                 in the program DO
      input_state(B) := NULL;
      FOR each immediate predecessor P of B DO
         input_state(B) += output_state(P)
      END FOR;
      output_state(B) := (input_state(B) +
         prog_lines(B)) - conf_lines(B)
   END FOR
END WHILE
```

Figure 2: Algorithm to Calculate Cache States

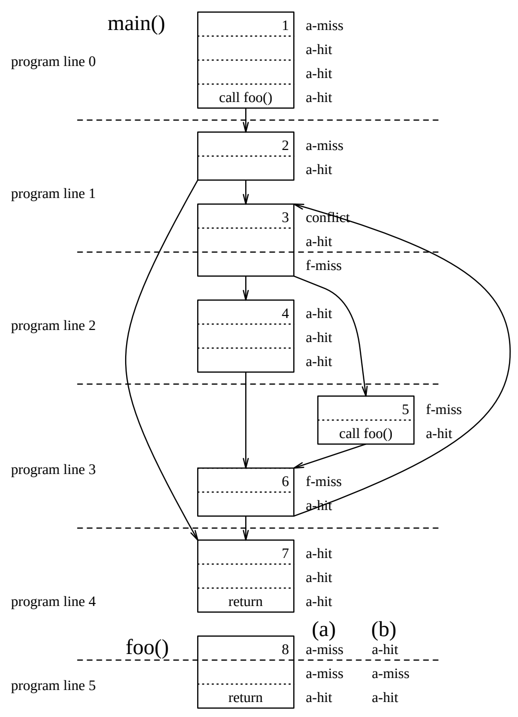
an input and output state of program lines which can potentially be in cache at that point. Initially the top block's input state (entry block of the main function) is set to all invalid lines. The input state of a block is calculated by taking the union of the output states of its immediate predecessors. The output state of a block is calculated by taking the union of its input state and the program lines accessed by the block and subtracting the program lines with which the block conflicts. The calculation of these abstract cache states requires a time overhead comparable to that of data-flow analysis used in optimizing compilers and a space overhead linear to the number of program lines, basic blocks, and function instances. The correctness of the algorithm for data-flow analysis is discussed in [1]. The calculation can be performed for an arbitrary control-flow graph, even if it is irreducible. The order of processing basic blocks is irrelevant.

Figure 3 illustrates a simple example of calculating input and output states. Assume there are 4 cache lines and the line size is 16 bytes (4 instructions). The immediate successor of a block with a call is the first block in that instance of the called function. Block 8a corresponds to the first instance of foo() called from block 1 and block 8b corresponds to the second instance of foo() called from block 5. Two passes are required to calculate the input and output states of the blocks, given that the blocks are processed in the order shown in Figure 3. Only the states of blocks 3, 4, and 5 changed during the second pass. Pass 3 results in no more changes. The reaching states are as follows: Block 7 cannot reach any program lines, and all other blocks can reach lines 1 to 5. The calculation of the reaching states can be performed by the same algorithm with `input_state(top) = conf_lines(B) =` ϕ.

After determining the abstract cache states (input states) of all blocks, each instruction is categorized according to the criteria specified in the previous section. By inspecting the states of each block, one can make some observations that may not be detected by a naive inspection of only physically contiguous sequences of references. For instance, the static simulation determined that the first instruction in block 7 will always be in cache (always hit) due to spatial locality. It also determined that the first instruction in basic block 8b will always be in cache (always hit) due to temporal locality. The last instruction in block 3 will not be in cache on the first reference, but will always be in cache on subsequent references (first miss). This is also true for the first instruction of block 5 and the first instruction of block 6, though a miss will only occur on the first reference of either one of the instructions. This situation is termed a *group first miss*. The first instruction in block 3 is classified as a conflict since it could either be a hit or a miss. The line is in conflict with the second instruction of block 8b, an always miss, due to the conditional execution of the call to foo() in block 5.

The current implementation of the static simulator imposes some restrictions. First, only direct-mapped cache configurations are allowed. Recent results have shown that direct-mapped caches have a faster access time for hits, which outweighs the benefit of a higher hit ratio in set-associative caches for large cache sizes [8]. Another restriction is that recursive programs are not allowed since cycles in the call graph would complicate the generation of unique function instances. Finally, indirect calls are not handled since the static simulator must be able to generate an explicit call graph.

[3]This additional requirement is a correction to the version of this paper published in the LCTS'94 workshop.



```
 "I" = invalid

  cache line 0 1 2 3 0 1 2 3 0 1                0 1 2 3 0 1 2 3 0 1
program line I I I I 0 1 2 3 4 5                I I I I 0 1 2 3 4 5

PASS 1
------
 input(1) = [I I I I            ]  output(1) = [  I I I 0          ]
input(8a) = [  I I I 0          ] output(8a) = [    I I         4 5]
 input(2) = [    I I         4 5]  output(2) = [    I I   1     4  ]
 input(3) = [    I I   1     4  ]  output(3) = [      I   1 2   4  ]
 input(4) = [      I   1 2   4  ]  output(4) = [      I   1 2   4  ]
 input(5) = [      I   1 2   4  ]  output(5) = [          1 2 3 4  ]
input(8b) = [          1 2 3 4  ] output(8b) = [            2 3 4 5]
 input(6) = [      I   1 2 3 4 5]  output(6) = [          1 2 3 4 5]
 input(7) = [    I I   1 2 3 4 5]  output(7) = [    I I   1 2 3 4 5]

PASS 2
------
 input(1) = [I I I I            ]  output(1) = [  I I I 0          ]
input(8a) = [  I I I 0          ] output(8a) = [    I I         4 5]
 input(2) = [    I I         4 5]  output(2) = [    I I   1     4  ]
 input(3) = [    I I   1 2 3 4 5]  output(3) = [      I   1 2 3 4  ]
 input(4) = [      I   1 2 3 4  ]  output(4) = [      I   1 2 3 4  ]
 input(5) = [      I   1 2 3 4  ]  output(5) = [          1 2 3 4  ]
input(8b) = [          1 2 3 4  ] output(8b) = [            2 3 4 5]
 input(6) = [      I   1 2 3 4 5]  output(6) = [          1 2 3 4 5]
 input(7) = [    I I   1 2 3 4 5]  output(7) = [     I I  1 2 3 4 5]
```

Figure 3: Example with Flow Graph

4 Bit-Encoding Approach

Based on the categorization of instruction references introduced in the previous section, a bit-encoding approach has been formulated. The intention of this approach is to provide better performance than uncached systems (as currently used in real-time systems) and better predictability over conventional caches with a moderate sacrifice in execution speed. A *fetch-from-memory* bit is encoded into the instruction format by dedicating a single bit position. If the bit is set in an instruction, then the instruction will be fetched from main memory. If the bit is not set, then the instruction will be fetched from cache.

During each cache reference, the fetch-from-memory bit is evaluated in parallel with the tag comparison, as shown in the Appendix. The following logic is used to resolve instruction fetch requests:

- If the cache access results in a miss, then the corresponding program line is fetched from main memory taking n cycles and the fetch-from-memory bit is ignored. (The bit would not be available anyway until the instruction is fetched.)
- If the tag comparison matches and the cache line is valid, then the effect depends on the evaluation of the fetch-from-memory bit.
  - If the bit is clear, then the processor is directed to use the instruction without delay.
  - If the bit is set, then the corresponding program line is fetched from main memory taking n cycles.[4]

In the last subcase, a memory fetch is performed although the program line already resides in code. If the effect of such a memory fetch is only simulated to reduce bus contention, as proposed in an earlier version of the paper, it would be unpredictable whether an actual memory fetch occurs or not. Thus, bus contention may or may not occur. The current semantics forces a memory access such that bus contention can be predicted for any memory reference with a fetch-from-memory bit set if a data reference occurs at the same time.

The fetch-from-memory bit is set whenever the Static Cache Simulation categorizes an instruction as a conflict or an always-miss. Otherwise the bit is cleared. This is straight forward for always-hits. For first-misses, on the other hand, the cache look-up fails on the first reference and the program line is fetched from main memory. For any subsequent references to this address, the instruction is found in cache with the bit clear resulting in a cache hit and a one cycle access time. Thus, bit-encoding takes advantage of first-miss instructions.

If an instruction is in a function that has multiple instances and the instruction has not been categorized

[4] The semantics has been changed for a set fetch-from-memory bit since the publication in the LCTS'94 workshop.

the same in the different instances, then the static simulator must decide whether or not to set the fetch-from-memory bit. Currently, the static simulator conservatively decides to fetch from memory if one or more instances categorize the instruction as a miss or a conflict. Otherwise, the bit is cleared[5].

4.1 Speedup

In this section the execution time w.r.t. instruction fetch overhead is analyzed. Other factors, *e.g.* data references to main memory, may add to the execution time but should not be adversely effected by the benefits of instruction caching.

For any uncached system, let the fetch time of one instruction be n cycles. Furthermore, let i be the number of instructions executed. Then, a lower bound for the time for this execution is

$$t_{uncached} = i * n \text{ cycles.} \tag{1}$$

For a cached system, let $i = h + m$ be the number of instructions executed where h and m are the number of hits and misses respectively. Assume a cache look-up penalty of one cycle [21, 7]. Since a cache look-up always has to be performed before it can be decided whether the program line associated with an instruction has to be fetched from main memory, the lower bound for an execution in a cached system is

$$t_{cached} = h + m * (n + 1) \text{ cycles.} \tag{2}$$

For the bit-encoded cached system, let $i = h' + m'$ be the number of instructions executed where h' and m' are the number of instructions fetched from cache and memory respectively[6]. Then, a lower time bound can be given as

$$t_{bit\_encoded} = h' + m' * (n + 1) \text{ cycles.} \tag{3}$$

There is both spatial and temporal locality inherent in the code of almost all programs. For instance, assume that a cache line consists of multiple instructions. The first reference to an instruction in such a line may cause a miss. But if instructions are executed sequentially, consecutive references to instructions of the same line will result in hits. Also, assume that some portion of the code that can be executed in a loop does not conflict with any other program lines that can be accessed by the loop. Subsequent references to this code in the same execution of the loop will also result in hits. Based on this observation, we can assume the following inequalities for an average execution:

$m \ll h$, $m' \ll h'$, and $h' < h$.

On average, we conclude with the following relation for an execution.

$$t_{cached} < t_{bit\_encoded} < t_{uncached} \tag{4}$$

5 Analysis

This section analyzes the benefit of predicting the behavior of instruction cache references. Cache measurements were obtained for user programs, benchmarks, and UNIX utilities listed in Table 1. The measurements were produced by modifying the back-end of an optimizing compiler VPO (Very Portable Optimizer) [3] and by performing Static Cache Simulation. The compiler back-end provided the control-flow information for the static simulator. It also produced assembly code with instrumentation points for instruction cache simulation. The cache simulation for traditional caches was based on the instruction categorization by the static simulator and has been validated by comparison with another trace-driven cache simulator. The validity of the bit-encoding approach was derived from mapping the instruction categories into the values for the fetch-from-memory bit. The assembly code was generated for the Sun SPARC instruction set, a RISC architecture with a uniform instruction size of one word (four bytes).

Name	Description
cachesim	Cache Simulator
cb	C Program Beautifier
compact	Huffman Code Compression
copt	Rule-Driven Peephole Optimizer
dhrystone	Integer Benchmark
fft	Fast Fourier Transform
genreport	Detailed Execution Report Generator
mincost	VLSI Circuit Partitioning
sched	Instruction Scheduler
sdiff	Side-by-side Differences between Files
tsp	Traveling Salesman
whetstone	Floating point benchmark

Table 1: Test Set of C Programs

The parameters for cache simulation included direct-mapped caches with sizes of 1kB, 2kB, 4kB, and 8kB (see column 1 in Tables 2 and 3). The cache line size was fixed at 4 words. The size of the programs varied between 500 and 4500 instructions (5kB – 18kB, see column 3 of Table 2). This provided a range of measurements from capacity misses dominating for small cache sizes to entire programs fitting in cache for large cache sizes. The number of instructions executed for

[5]It is possible in such a situation that the merged instruction could be safely classified as a first-miss and have its bit cleared. An example of this situation is the first instruction in block 8 of Figure 3. It is the authors' intention to analyze the control flow to recognize these situations in the future.

[6]$h'$and m' are approximately the same as the number of instructions executed with the fetch-from-memory bit clear and set respectively with the exception of first-misses which are counted as misses on the first reference and hits on subsequent references.

		number of		cache prediction				
Cache		instructions		DAG	tree			
Size	Name	DAG	tree	memory	always hit	always miss	first-miss	conflict
	cachesim	2,115	9,397	28.07%	69.30%	8.23%	0.74%	21.73%
	cb	1,242	7,017	31.08%	75.70%	2.85%	0.00%	21.45%
	compact	1,478	2,173	29.84%	69.67%	5.52%	0.18%	24.62%
	copt	1,037	1,152	21.99%	71.01%	7.73%	7.03%	14.24%
	dhrystone	479	549	23.17%	69.76%	11.29%	6.56%	12.39%
1kB	fft	492	528	9.55%	74.05%	4.92%	16.29%	4.73%
	genreport	4,430	7,060	19.77%	70.64%	11.30%	6.18%	11.88%
	mincost	1,112	1,657	28.33%	72.24%	8.57%	1.39%	17.80%
	sched	2,068	3,378	32.88%	66.55%	5.98%	0.15%	27.32%
	sdiff	1,822	10,407	28.06%	67.44%	12.28%	1.08%	19.21%
	tsp	1,181	1,236	22.95%	72.73%	13.59%	4.37%	9.30%
	whetstone	1,204	1,485	26.62%	75.69%	11.65%	0.27%	12.39%
1kB	average	1,555	3,837	25.19%	71.23%	8.66%	3.69%	16.42%
2kB	average	1,555	3,837	21.18%	72.09%	5.88%	7.28%	14.75%
4kB	average	1,555	3,837	11.35%	72.40%	4.36%	16.64%	6.60%
8kB	average	1,555	3,837	4.73%	72.61%	4.03%	22.77%	0.59%

Table 2: Static Measurements

each program comprised a range of 1 to 19 million using realistic input data for each program (see column 3 of Table 3).

5.1 Static Analysis

Static Cache Simulation classifies instructions into categories based on the predicted cache behavior. Table 2 shows the static number of instructions for each program (column 3) and the number of instructions associated with all function instances when the call graph is converted from a DAG to a tree (column 4). Column 5 denotes the percentage of instructions in the DAG which have the fetch-from-memory bit set. Columns 6 to 9 show the percentage of instructions in the tree for each category as determined by the static simulator. Notice that the cache behavior could be predicted statically for 84-99% of the instructions, depending on the ratio of program size and cache size. The remaining 1-16% are due to conflicts.

5.2 Dynamic Analysis

Table 3 illustrates the dynamic behavior of three systems: an uncached system (simulating a disabled instruction cache), a cached system with the bit-encoding approach, and a conventional cached system. Column 3 indicates the number of instructions executed. The hit ratio (percentage of cache hits of all instruction references) is shown for the bit-encoded system in column 4 and for conventional caches in column 5. Column 6 shows the percentage of executed instruction references which were classified as conflicts on a cached system. Column 7 indicates the estimated execution time in cycles for an uncached system. The percentage of cycles required for a bit-encoded system (column 8) and for a conventional cached system (column 9) are compared to an uncached system. The execution time is calculated based on the equations 1, 2, and 3 for $n = 9$.[7]

The bit-encoding approach results in lower hit ratios (72-98%) than on a conventional cached system (92-99%). Yet, caches are often disabled for critical real-time tasks to provide the predictability required by scheduling analysis. Thus, the bit-encoding approach should be compared to an uncached system. The bit-encoding method requires only 13-39% of the cycles used by the uncached systems. This provides a speedup of programs by a factor of 3-8 without sacrificing the predictability of a program's execution time. The result resembles the improvement over critical real-time tasks which require caches to be disabled. The results improve considerably as the cache size increases and entire programs fit into cache. The execution time required for a conventional cached system is only about 14% of an uncached system, but the predictability also decreases to the point where it becomes insufficient for scheduling analysis of critical tasks. This can be explained as follows:

Conflicts correspond to the instructions whose cache behavior could not be predicted prior to execution in a conventional cached system. The dynamic percentage of conflict references is higher than the static percent-

[7]The latency for a memory fetch is assumed to be $n = 9$ cycles, a cache look-up takes one cycle, and thus a cache hit also consumes one cycle while a miss takes $n + 1 = 10$ cycles. These assumptions are described as realistic by other researchers [21, 7]. A memory fetch in an uncached system fetches exactly one instruction while a memory fetch in a cached system fetches a line of 4 instructions. Fetching a line of multiple instructions is typically accomplished through a wider bus between cache and main memory for a cached system.

Cache Size	Name	# instructions executed	hit ratio bit-enc.	hit ratio cached	conflicts cached	exec time [cycles] uncached	% of exec time bit-enc.	% of exec time cached
1kB	cachesim	2,995,817	65.70%	77.19%	28.52%	26,962,353	45.41%	33.92%
	cb	3,974,882	67.24%	93.84%	31.08%	35,773,938	43.87%	17.27%
	compact	13,349,997	67.12%	92.90%	32.45%	120,149,973	43.99%	18.21%
	copt	2,342,143	68.56%	93.64%	28.93%	21,079,287	42.55%	17.47%
	dhrystone	19,050,093	77.95%	83.73%	15.75%	171,450,837	33.16%	27.38%
	fft	4,094,244	91.17%	99.95%	8.80%	36,848,196	19.94%	11.16%
	genreport	2,275,814	74.64%	97.49%	24.58%	20,482,326	36.47%	13.63%
	mincost	2,994,275	67.35%	89.08%	28.06%	26,948,475	43.76%	22.03%
	sched	1,091,755	67.21%	96.41%	32.15%	9,825,795	43.90%	14.70%
	sdiff	2,138,501	71.20%	97.61%	28.40%	19,246,509	39.92%	13.50%
	tsp	3,004,145	72.01%	86.98%	22.06%	27,037,305	39.10%	24.13%
	whetstone	8,520,241	71.57%	100.00%	23.78%	76,682,169	39.54%	11.11%
1kB	average	5,485,992	71.81%	92.40%	25.38%	49,373,930	39.30%	18.71%
2kB	average	5,485,992	77.81%	97.49%	21.14%	49,373,930	33.30%	13.62%
4kB	average	5,485,992	90.73%	99.74%	9.12%	49,373,930	20.38%	11.37%
8kB	average	5,485,992	98.15%	99.99%	1.76%	49,373,930	12.97%	11.13%

Table 3: Dynamic Measurements

age given in Table 2 since conflicts typically occur in loops. Since 5-25% of the instructions executed were conflicts, the execution time of programs cannot be predicted as tightly in conventional cached systems with traditional timing tools, especially for smaller caches. However, more recent work by the authors shows that the instruction categorization of the static simulator may be used by a more sophisticated timing tool to provide tight worst-case execution time predictions with a 4-9 times speedup over uncached system using a conventional instruction cache [2].

6 Future Work

We are currently working on applying Static Cache Simulation to data caches under certain restrictions, such as the absence of pointers and dynamic memory allocation. Most other data references could be predicted statically. Previous work has shown improvements by balancing the number of instructions placed behind loads where the memory latency was uncertain [9]. By predicting the memory latency of a large portion of loads, instruction scheduling could be performed more effectively. For example, the number of instructions the scheduler would place between a load instruction and the first instruction referencing the loaded register should be greater for a data reference classified as an always miss than an always hit.

We are currently working on integrating the method of Static Cache Simulation with a tool which estimates a program's best-case execution time (BET) and worst-case execution time (WET) [6, 2]. Using the information provided by Static Cache Simulation, the BET and WET can be based on the categorization of instructions. This relieves the time-estimation tool from having to simulate all possible cache states. The instruction categorization is refined to provide a separate category for each loop level, thereby providing the base for tight execution time predictions.

With the bit-encoding approach, a traditional tool predicting the execution time can perform the same type of analysis and provide estimations for both BET and WET. But the execution time predictions can be tighter since the caching behavior is fully predictable. Instructions classified as always-hits can be assumed to require one cycle, and always-misses or conflicts can be estimated to take $n+1$ cycles. For a first-miss, the tool could distinguish between the first reference ($n+1$ cycles) and any subsequent references (one cycle) by simply tagging first-miss instructions which have been encountered. A traditional timing tool should be easily modified to take the effect of bit-encoding into account. The resulting execution time estimate will be as tight as for uncached systems since the estimation of the fetch cost accurately represents the number of cycles taken for an instruction in any category. There is no uncertainty with respect to the effect of an instruction classified as conflict, the fetch will always take $n+1$ cycles.

The static simulator could be extended in several ways. First, recursive functions could be handled by applying the described algorithm to calculate abstract cache states repeatedly on a function instance. Second, a modified algorithm and data structure could be designed to handle set-associative caches.

There are several other applications of Static Cache Simulation. For example, instruction cache analysis can be sped up by determining the caching behavior of a large number of references prior to execution time [14]. Other applications include detailed profiling and tracking of execution time for a real-time debugger [15].

7 Conclusion

Cache memories have often been disabled for critical real-time tasks to provide sufficient predictability for scheduling analysis. This paper shows that the behavior of instruction cache references can be predicted to a large extent prior to the execution of a program via the method of Static Cache Simulation. The cache simulator uses information provided by the back-end of a compiler to statically predict the cache behavior of 84-99% of the instructions. Furthermore, a fetch-from-memory bit has been proposed which is added to the instruction encoding. This approach provides a speedup in execution time by a factor of 3-8 over uncached systems without sacrificing the predictability of the program's worst-case execution time. The ability to predict the caching behavior of a large percentage of the instruction references (in a conventional cached system) or even all instruction references (using the fetch-from-memory bit) can be used to predict the execution time of large code segments on machines with instruction caches.

In summary, instruction cache behavior is sufficiently predictable to provide worst-case execution time predictions which are tight enough for scheduling analysis in a non-preemptive environment. Thus, the performance advantage of instruction caches can be exploited for critical real-time tasks by enabling either conventional or bit-encoded instruction caches.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Symposium on Real-Time Systems*, December 1994.

[3] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.

[4] C.-H. Chi and H. Dietz. Unified management of register and cache using liveness and cache bypass. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 344–355, June 1989.

[5] T. Hand. Real-time systems need predictability. *Computer Design (RISC Supplement)*, pages 57–59, August 1989.

[6] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *IEEE Symposium on Real-Time Systems*, pages 68–77, December 1992.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[8] M. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(11):25–40, December 1988.

[9] D. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289, June 1993.

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

[11] E. Kligerman and A. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

[12] S. McFarling. Program optimization for instruction caches. In *Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[13] F. Mueller and D. B. Whalley. Efficient on-the-fly analysis of program behavior and static cache simulation. In *Static Analysis Symposium*, September 1994.

[14] F. Mueller and D. B. Whalley. Fast instruction cache analysis via static cache simulation. TR 94-042, Dept. of CS, Florida State University, April 1994.

[15] F. Mueller and D. B. Whalley. On debugging real-time applications. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[16] D. Niehaus. Program representation and translation for predictable real-time systems. In *IEEE Symposium on Real-Time Systems*, pages 53–63, December 1991.

[17] D. Niehaus, E. Nahum, and J. A. Stankovic. Predictable real-time caching in the spring system. In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 80–87, 1991.

[18] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[19] V. Sarkar. Determining average program execution times and their variance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, June 1989.

[20] D. Simpson. Real-time RISCS. *Systems Integration*, pages 35–38, July 1989.

[21] A. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

Appendix

The access logic for an instruction cache using the proposed bit-encoded approach is illustrated in Figure 4. The instruction memory contains the cached instructions. It is accessed by using the index field to select the cache line and the offset field to select the instruction within that line. The tag memory contains the state bit and address tag for each cache line and is also accessed by using the index field. The match logic compares the tag of the instruction's physical address to the tag obtained by accessing the tag memory and verifies the state to ensure that the cache line is valid. In parallel, it also checks that the fetch-from-memory bit is clear. If any of these conditions are not met, then it informs the CPU to stall. The logic to request a main memory fetch or stall for the appropriate number of cycles is not shown in this figure.

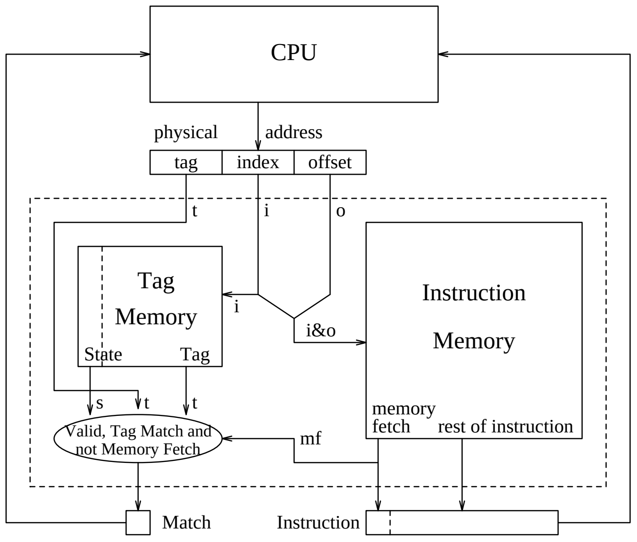


Figure 4: Access Logic for Bit-Encoded Approach