Fast Instruction Cache Analysis via Static Cache Simulation [*]

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

Abstract

This paper introduces a new method for instruction cache analysis that outperforms conventional trace-driven methods. The new method, **static cache simulation**, *analyzes a program for a given cache configuration and determines prior to execution time if an instruction reference will always result in a cache hit or miss. At run time, counters are incremented to provide the execution frequency of portions of code. In addition, the cache behavior is simulated for references that could not be predicted statically. The dynamic simulation employs a novel view of the cache by updating local state information associated with code portions. The total number of cache hits and misses can be inferred from the frequency counters at program exit. Measurements taken from a variety of programs show that this new method speeds up cache analysis over conventional trace-driven methods by almost an order of a magnitude. Thus, cache analysis with static cache simulation makes it possible to analyze the instruction cache behavior of longer and more realistic program executions.*

1 Introduction

Cache memories have become a major factor to bridge the bottleneck between the relatively slow access time to main memory and the faster clock rate of today's processors. The simulation of cache memories is common practice to determine the best configuration of caches during the design of computer architectures. It has also been used to evaluate compiler optimizations with respect to cache performance.

Unfortunately, the cache analysis of a program can significantly increase the program's execution time, often by two orders of a magnitude. Thus, cache simulation has been limited to the analysis of programs with a small or moderate execution time and still requires considerable experimentation time before yielding results. In reality, programs often execute for a long time, but cache simulation simply becomes infeasible with conventional methods. The large overhead of cache simulation is imposed by the necessity of tracking the execution order of instructions.

On the other hand, instruction frequency measurements can be obtained by inserting instructions into a program that increment frequency counters. The counters are typically associated with a basic block and are incremented each time the basic block executes. The overhead induced by frequency measurements is less than a factor of two in execution time. This much lower overhead can be attributed to the fact that the execution order of instructions is irrelevant.

The method for instruction cache analysis discussed in this paper makes extensive use of frequency counters when instruction references are statically determined to be always cache hits or always cache misses. For the remaining instruction references, state information is associated with code portions and is updated dynamically. This state information represents a localized view of the cache and is used to determine whether the remaining program lines of a code portion are or are not cached. These localized states are in contrast to a comprehensive global view of the cache state as employed in conventional trace-driven simulation. In summary, the cheaper method of frequency counters is used where the order of execution is irrelevant and the remaining references are determined by local states, which also impose less execution overhead than one global cache state.

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 are translated by a compiler. The compiler generates assembly code with macro entries for instrumentation and passes information about the control flow of each source file to the static cache simulator.

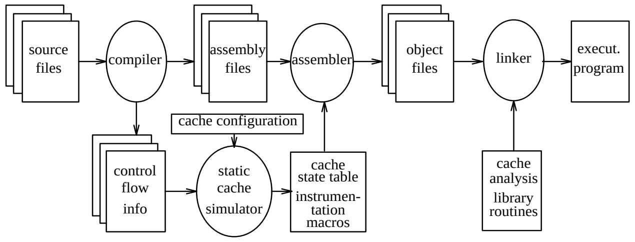


Figure 1: Overview of Static Cache Simulation

[*]E-mail: whalley@cs.fsu.edu, phone: (904) 644-3506. Supported in part by ONR contract # N00014-94-1-0006.

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, macro code for instrumenting the executable is generated together with tables to store cache information at run time. This output of the simulator is passed to the assembler, which translates the code generated by the compiler into instrumented object code. The linker combines these object files into an executable program and links in library routines, which produce the final report of the cache analysis at run time[1].

The approach taken by static cache simulation is quite different from traditional methods. The simulator attempts to determine statically whether a given program line will result in a cache hit or miss during program execution. This is achieved by the analysis of both the call graph of the program and control-flow graph for each function. A set of instructions executed in sequence is called a unique path (UP) if it can be distinguished from all other paths by at least one (unique) control-flow transition. To better predict the cache behavior, functions are further distinguished by function instances that depend on the call site and call sequence. If the simulator cannot determine statically if a line results in a hit or miss, then the cache behavior has to be determined at run time by updating local path states. The static analysis provides the information required to instrument the generated code with short instruction sequences. These sequences count the frequency of executions for a state of a path and update the states where necessary. The total hits and misses can be inferred from the state-dependent frequency counts after running the program.

2 Related Work

Evaluating cache performance has long been recognized as a challenging task to be performed in an efficient manner. Traces of the actual addresses referenced during the execution of programs have to be used to perform a realistic evaluation. The problem is that a realistic trace typically consists of millions of references. Evaluation of these traces can require excessive amounts of space and time when using simple approaches. For instance, a traditional approach is to generate the trace via trapping or simulation, write each generated address in the trace on disk, and analyze the trace via a separate program that reads the trace from disk and simulates the cache. Such an approach can easily slow the execution by a factor of a 1000 or more [14, 17, 9].

A technique called inline tracing can be used to generate the trace of addresses with much less overhead than trapping or simulation. Measurement instructions are inserted in the program to record the addresses that are referenced during the execution. Borg, Kessler, and Wall [5] modified programs at link time to write addresses to a trace buffer, and these addresses were analyzed by a separate higher priority process. The time required to generate the trace of addresses was reduced by reserving five of the general purpose registers to avoid memory references in the trace generation code. Overhead rates of 8x to 12x normal execution time were reported for the trace generation. Analysis of the trace was stated to require at least 10x the overhead of the generation of the trace (or about 100x slower than normal execution time).

Eggers *et. al.* [6] also used the technique of inline tracing to generate a trace of addresses in a trace buffer, which was copied to disk by a separate process. They used several strategies for minimizing the overhead of generating the trace. First, they produced a subset of the addresses from which the other addresses could be inferred during a postprocessing pass. For instance, they only stored the first address in a sequence of contiguous basic blocks with a single entry point and multiple exit points. Rather than reserving a set of registers to be used for the trace generation code, they identified which registers were available and thus avoided executing many save and restore instructions. The trace generation overhead was accomplished in less than 3x the normal execution time. In addition, writing the buffers to disk required a factor of 10x normal execution time. The postprocessing pass, which generates the complete trace from the subset of addresses stored, was much slower (about 3,000 addresses/sec). No information was given on the overhead required to analyze the cache performance.

Ball and Larus [3, 10] also reduced the overhead of the trace generation by storing a portion of the trace from which the complete trace can be generated. They optimized the placement of the instrumentation code to produce the reduced trace with respect to a weighting of the control-flow graph. They showed that the placements are optimal for a large class of graphs. The overhead for the trace generation was less than a factor of 5. However, the postprocessing pass generating a full trace required 19-60x the normal execution time.

Whalley [15, 16] evaluated a set of techniques to

[1]When the cache configuration changes, no recompilation is needed; only the static cache simulator, assembler, and linker have to be reinvoked.

reduce the time required to evaluate instruction cache performance. He linked a cache simulator to the programs instrumented with measurement code to evaluate the instruction cache performance during the program's execution. The techniques he evaluated avoided making calls to the cache simulator when it could be determined in a less expensive manner that the reference was a hit. The overhead time for the faster techniques was highly dependent upon the hit ratio of the programs. He reported 15x normal execution time for average hit ratios of 96% and 2x normal execution time for hit ratios exceeding 99%. These faster techniques also required recompilation of the program when the cache configuration was altered.

3 Compiler Interaction

To perform instruction cache analysis via static cache simulation on a program with our tools, the program is first compiled by a compiler specifically modified for this task. During the compilation the control flow of the functions of a program is partitioned into unique paths (UPs). Informally, a UP is a set of basic blocks [1] (vertices) connected by control-flow transitions (edges) that contain at least one unique transition, *i.e.* a transition that does not occur in any other UP. In this study, UPs were restricted to not cross loop boundaries, function boundaries, or calls. The notion of UPs allows one to identify and reduce the locations for code instrumentation (rather than inserting measurement code in every basic block). The resulting directed graph of UPs represents the control flow of the program in such a form that it can be easily processed later by the static cache simulator. A formal definition of UPs is given in [11, 12].

A path macro invocation is generated on a unique transition of each UP in the assembly code. The corresponding body of the macro, which provides the measurement code for the UP, is defined by the static cache simulator. Similarly, a call macro invocation is generated for each call to a function. The parameters for the call and path macros are a register containing the base address of the counter table for the current function instance and two other registers that the compiler determines to be unused. The compiler generates spill code to free registers if none are available.

4 Static Cache Simulation

The method of static cache simulation can be used to statically predict the behavior of a large portion of the instruction cache references for a given program 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.

By analyzing the call graph and the control flow of each function, static cache simulation attempts to determine if each instruction reference will result in a cache hit or miss during program execution. Since it is not always possible to determine if a reference will be a hit or miss, instructions are classified to be in the 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 hits or misses depending on the flow of control, then it is referred to as a conflict.

4.1 Decomposition

To statically determine a program's cache behavior as accurately as possible, the program is decomposed into smaller components. A program may be composed of a number of functions. The possible sequence of calls between these functions is depicted in a call graph. The control flow of each function can be represented by a control-flow graph where nodes are UPs and edges denote legal transitions of the control flow between UPs. The static simulator obtains this information from the compiler.

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, the directed acyclic call graph is transformed into a tree of function instances.

4.2 Instruction Categorization

Static cache simulation calculates the abstract cache states associated with the UPs. Such an abstract cache state specifies the possible cache contents before the UP is executed. First, formal definitions are provided. Then, the caching behavior of each instruction is categorized based on these definitions. An example is discussed in section 4.3 (see [11] for more details).

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

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

The notion of an abstract cache state is a compromise between the choice of an exhaustive set of all cache states that may occur at execution time and the exponential growth of such an exhaustive set during simulation. The next definition introduces the reaching state, which is used for the categorization of instructions following thereafter.

Definition 3 *A* **reaching state** *of a UP in a function instance is the subset of all program lines that can be reached through control-flow transitions from the UP of the function instance.*

For a given function instance, each instruction i within a UP 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 r. The program line l maps into cache line c, denoted by $l \rightarrow c$. The UP containing line l is referred to simply as UP.

always-miss: A cache miss is predicted if

- $i = i_0$: instruction i is the first reference to program line l in UP 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 a consecutive reference to program line l in UP. Or all of the following conditions hold:
- $i = i_0$: instruction i is the first reference to program line l in UP,
- $l \in s$: l is in the abstract cache state, and
- $\nexists_{k \neq l} \; k \in s \wedge k \rightarrow c$: no other line k (which maps into the same cache line as l) is in the abstract cache state.

first-miss: The first reference to an instruction will result in a cache miss and all subsequent references in cache hits if

- $i = i_0$: instruction i is the first reference to program line l in UP,
- $l \in s$: l is in the abstract cache state,
- $\exists_{k \rightarrow c, k \neq l} \; k \in s$: another line k (which maps into the same cache line as l) is also in the abstract cache state,
- $\forall_{k \rightarrow c, k \neq l} \; k \in s \Rightarrow k \notin r$: if any line k (which maps into the same cache line as l) is also in the abstract cache state, then the line is not in the reaching state of UP, and
- $\forall_{i \in \{i_1..i_{n-1}\}} \; category(i) \in \{always\text{-}hit, first\text{-}miss\}$: all other instructions within the same program line are always-hits or first-misses.

conflict: All other instructions are conflicts.

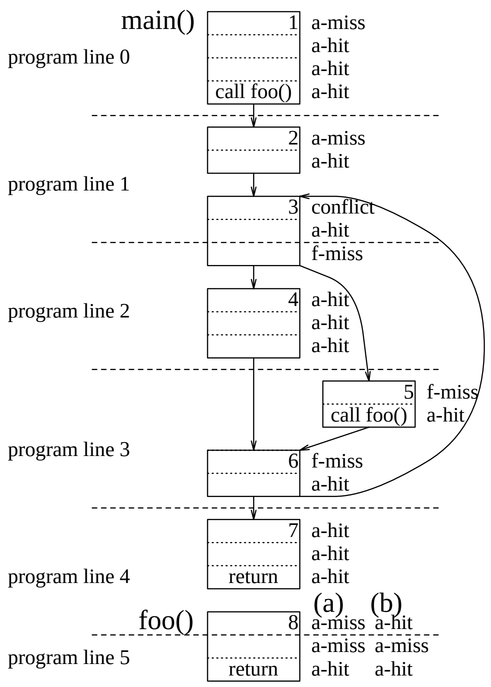
4.3 Implementation

The iterative algorithm in Figure 2 was used to calculate the abstract cache states. Each UP has an input and output state of program lines that can potentially be in cache at that point. Initially, the input states of the top paths (entry paths of the main function) are set to all invalid lines. The input state of a path is calculated by taking the union of the output states of its immediate predecessors. The output state of a path is calculated by taking the union of its input state and the program lines accessed by the path and subtracting the program lines with which the path conflicts. This calculation includes the interprocedural propagation of abstract cache states (not explicitly shown in the algorithm): At a path p with a call to function f as the last instruction, the output state of the UP is propagated to the input states of the entry paths of the corresponding function instance $f(i)$. Similarly, the union of the output states of $f(i)$'s exit paths provides the input state for p's successor paths.

Input: Function-Instance Graph of the program and UPPA for each function.
Output: Abstract Cache State for each UP.
Algorithm: Let `conf_lines(UP)` be the set of program lines (excl. the program lines of UP) mapping into the same cache line as any program line within the UP.

```
input_state(main):= all invalid lines;
WHILE any change DO
    FOR each instance of a UP in the program DO
        input_state(UP):= $\phi$;
        FOR each immediate predecessor P of UP DO
            input_state(UP):=
                input_state(UP) $\cup$ output_state(P);
        output_state(UP):= [input_state(UP) $\cup$
            prog_lines(UP)] $\setminus$ conf_lines(UP);
```

Figure 2: Algorithm to Calculate Cache States



```
 "I" = invalid

cache   0 1 2 3 0 1 2 3 0 1 cache ln. 0 1 2 3 0 1 2 3 0 1
program I I I I 0 1 2 3 4 5 prog. ln. I I I I 0 1 2 3 4 5

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

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

Figure 3: Example with Flow Graph

The algorithm is a variation of an iterative data-flow analysis algorithm commonly used in optimizing compilers. Thus, the time overhead of the algorithm is comparable to that of data-flow analysis and the space overhead is $O(pl * UPs * fi)$, where pl is the number of program lines, UPs is the number of paths, and fi the number of 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. In addition, the order of processing basic blocks is irrelevant for the correctness of the algorithm. The reaching states can be calculated using the same base algorithm with `input_state(main) = conf_lines(UP) =` ϕ.

Figure 3 depicts the calculation of input and output states. The paths are restricted to basic blocks to simplify the example. In the example, there are 4 cache lines and the line size is 16 bytes (4 instructions). Thus, program line 0 and 4 map into cache line 0, program line 1 and 5 map into cache line 1, program line 2 maps into cache line 2, and program line 3 maps into cache line 3. 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 some blocks inside the loop change on the second pass. Pass 3 results in no more changes.

After determining the input states of all blocks, each instruction is categorized based on its abstract cache state (derived from the input state) and the reaching state.[2] By inspecting the input states of each block, one can make some observations that may not have been 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. This can be determined by observing that line 4 is in `in(7)` and no conflicting program line is in `in(7)`. It was also determined that the first instruction in basic block 8b will always be in cache (always hit) due to temporal locality. The static simulation determined that the last instruction in block 3 will not be in cache on its first reference, but will always be in cache on subsequent references (first miss). This is indicated by `in(3)`, which includes program line 2 but also a conflicting program line "invalid" for cache line 3. Yet, the conflicting program line cannot be reached. This is also true for the first instructions of block 5 and 6, though a miss will only occur on the first reference of either one of the instructions. This is termed a *group first miss*. Finally, the first instruction in block 3 is classified as a conflict since it could either

[2]The reaching state for all paths contains line 1–5, except for reach(7) which is empty.

be a hit or a miss (due to the conditional call to foo). This is indicated by `in(3)`, which includes program line 1 and a conflicting program line 5 that can still be reached.

The current implementation of the static cache simulator imposes some restrictions. First, only direct-mapped cache configurations are supported. 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]. Second, context switches cannot be simulated using this method. Third, recursive programs currently 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 cache simulator must be able to generate an explicit call graph.

5 Code Instrumentation

After decomposing the program into function instances and UPs, there still remain lines that are analyzed to be in conflict with another line. It is inevitable to maintain information at run time to determine which program line is currently cached and to update this information dynamically. This is achieved by maintaining a *path state* at execution time. A path state only reflects the conflicts local to the current path (in contrast to a cache state that comprises the global state of a cache memory). Consider the example in Figure 4, which will be discussed step-by-step in the following. Path 1 contains program line a which conflicts with program line x since both map into cache line c and line x can be reached from path 1. Thus, the shared path state (SPS) for path 1 keeps track of whether or not program line a is in cache. More detailed examples are given in [11].

Naively, a path state may be kept on the most specialized level (for each function instance and path). But this would require a considerable amount of interaction between UPs. In the worst case, the execution of a UP of some function instance would not only have to update its path state but every other path state conflicting with a line of this path and any function instance. Cache state information is therefore merged after simulation in two stages to comprise path states. First, the conflicts of the cache states of a UP of all instances of a function are merged into one *local path state* (LPS). Second, local path states of neighboring UPs that share at least one instruction are merged into one *shared path state* (SPS). The latter is illustrated in Figure 4. The LPS of path 1 contains the conflicting program line a while the LPS of path 2 contains

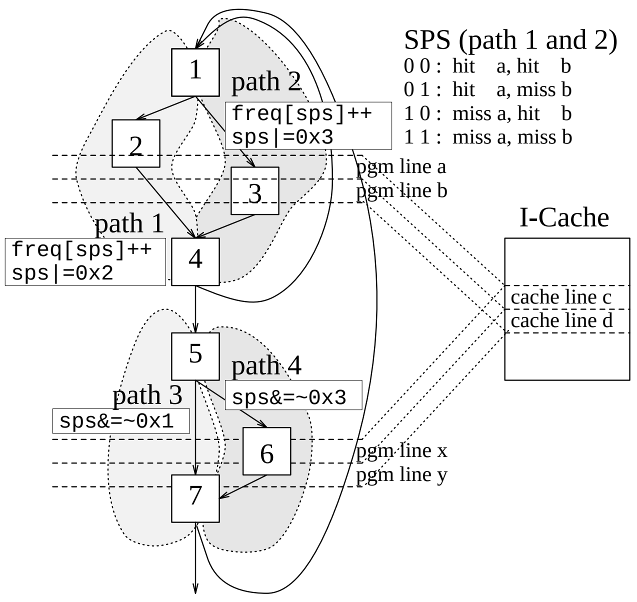


Figure 4: Frequency Counters Indexed by the SPS

both conflicting program lines a and b. The two paths overlap in block 1 and 4. Thus, the SPS for paths 1 and 2 contains the program lines a and b. LPSs allow uniform instrumentation of code rather than distinguishing instances dynamically at every instrumentation point or replicating code for each instance. Both merging operations greatly reduce the overhead of dynamic simulation for conflicts. While a SPS only needs to maintain one state to keep track of conflicts dynamically, the state may comprise a wider range of values to combine all possible conflicts of overlapping paths.

Generated code is instrumented by inserting instructions at the unique transition of each UP to keep track of the SPSs and record the frequency of executed instructions for this path and state. At the exit points of the program, an epilogue is inserted to call a library routine, which calculates the total hits and misses from the gathered state-dependent frequencies.

The code emitted by the compiler back end includes macro calls for each UP and for each call site. The simulator generates the corresponding macros bodies, produces tables to store SPSs and frequency counters at run time, and provides other constant data structures for the final calculation of hits and misses.

5.1 Updating Shared Path States

For each SPS, a state field is generated in the state table. These states are modified at run time by the macro code of UPs. The value of such a state denotes which lines are cached out of a set of conflicting lines. The initial value denotes the set of lines cached prior to the first execution of any corresponding UP. The value can be used as an index into the frequency counter array of the current UP. Thus, state-dependent frequency counting can be performed by using the SPS as an index into the counter array and incrementing the corresponding counter. Furthermore, if an SPS is constant at run time (no conflicting lines), then the state field is omitted from the state table.

Consider Figure 4 again. The SPS of paths 1 and 2 is used to simulate the hits and misses of program lines a and b. This SPS has two bits (due to two conflicting program lines) to hold the possible encoding of cached program lines of the SPS (as shown in the figure). The state is updated on the execution of path 1 to include program line a. The execution of path 2 includes both a and b in the state, the execution of path 3 excludes b, and the execution of path 4 excludes both a and b. Simple bit manipulations suffice for these updates, as indicted by the pseudo code in the figure. The separate counter array for path 2 is incremented in the same manner. The SPSs for paths 3 and 4 are not shown to simplify the example.

5.2 Frequency Counters

For each UP of every function instance, an array of frequency counters is used to keep track of the execution frequency of the UP. The size of the array is determined by the number of permutations of conflicting lines for a SPS. Since the size is growing exponentially with the number of conflicting lines, an alternate counter array with a constant size of two entries is provided for large numbers of conflicting lines in the SPS. There is a time/space trade-off between the two alternatives (discussed in the context of the path macros).

Figure 4 shows the frequency counter, indexed by the SPS of path 1 and 2, which is incremented. Path 2 has an array of four frequency counters, corresponding to each possible value of the SPS. An increment of the first counter element corresponds to hits on line a and b, an increment of the second counter element indicates a hit on a and a miss on b, etc. The frequency counter increments for paths 3 and 4 are not shown.

5.3 Macros for Calls

Macro code is generated at call sites to pass the base address of the counter table for the callee's function instance as an additional parameter. The function instance can thereby be identified by path macros.

5.4 Macros for Paths

The code emitted for path macros increments the frequency counter indexed by the SPS, updates the SPS to reflect that the lines of the current path are now cached, and updates any other SPS of conflicting paths. If a different path *shares* a line (but not the SPS) with the current path, the line is marked as cached in the SPS of the conflicting path. Conversely, if a different path *conflicts* with the current SPS in a line, the line is marked as uncached in the SPS of the conflicting path, as discussed before in Figure 4.

Alternately, code is emitted to increment a general frequency counter for large SPSs. Since no counter array is generated for large SPSs, indexing into an array becomes obsolete. Rather, the SPS is first combined with an AND mask to single out the conflict lines of only the current path. Then, the number of remaining on-bits is counted and added to a second counter, which accumulates references to conflicting lines resulting in misses. This alternate method requires less counter space but increases execution time by determining the number of set bits in a loop[3]. In general, alternate methods of code instrumentation optimize special cases to reduce the instrumentation overhead.

5.5 First Miss Table

If a path of a function instance contains a line that is classified as a first miss, an entry for this line is created in the first miss table. If another path shares the same line and also counts this line as a first miss, this path's instance is also included in the same table entry. This table is used to adjust the total number of hits and misses as explained in the next section.

6 Calculation of Hits and Misses

The total number of hits and misses can be inferred from the state-dependent frequency counters and from the first miss table. This calculation is performed after running the instrumented program as part of its exit code. The calculation is independent from the number of SPSs or any other code generation parameters and can thus be hidden in a library routine, which is linked with the instrumented program.

6.1 Hits/Misses based on Frequencies

For each path of each function instance, the product of a frequency count and the number of always hits (misses) is added to the total number of hits (misses). First misses, weighted by the frequency, are also added to the total number of hits at this point.

The index into the counter array indicates the number of hits and misses for conflicting lines, which are

[3]RISC architectures as well as most CISC architectures do not provide a special bit-counting instruction.

then also multiplied by the corresponding frequency. A zero index indicates that all conflicting lines are cached while the last index corresponds to misses of all conflicting lines (see SPS bit encoding in Figure 4).

Not all cache line configurations may be valid during the execution of the program for a given path and instance. In other words, the frequency count for some indices should be zero. But to minimize the amount of state changes during run time, a conflicting SPS is not updated if it can be determined at simulation time that the corresponding cache state cannot occur. Therefore, only a subset of counter indices may actually correspond to a valid cache configuration for a given path and instance. The number of conflicting lines is thus inferred from the array index combined with an AND mask with bits set in the position of valid cache lines. Consider path 1 in Figure 4. The AND mask for this path is $0x2$ since only bit 2 (corresponding to program line a in the encoding of the SPS) is referenced when executing path 1.

If the number of states in the SPS was large and the alternate counting method was applied, then the always hits (misses) and first misses are still counted based on the frequency counter. The number of misses due to conflicts is readily available in one counter. The number of hits can be calculated as the total frequency times the number of conflict lines less the number of misses due to conflicts.

6.2 First Miss Adjustment

Since first misses were exclusively counted as hits with respect to the frequency, the hits and misses have to be adjusted. For each entry in the first miss table, the counters of corresponding paths (and instances) are checked. If the frequency of at least one paths is greater than zero, the total number of hits is decremented while misses are incremented by one.

7 Measurements

This section evaluates the benefits of instruction cache analysis via static cache simulation. Cache measurements were obtained for user programs, benchmarks, and UNIX utilities. The measurements were produced by modifying the back-end of the optimizing compiler VPO (Very Portable Optimizer) [4] and by performing static cache simulation. The simulation was performed for the Sun SPARC instruction set, a RISC architecture with a uniform instruction size of one word (four bytes).

The parameters for cache simulation included direct-mapped caches with sizes of 64B to 8kB. The cache line size was fixed at 4 words. No context switches were simulated. The size of the programs varied between 2kB and 18kB. This provided a range of measurements from capacity misses dominating for small cache sizes to some programs entirely fitting in cache for large cache sizes. The number of instructions executed for each program comprised a range of 1 to 19 million using realistic input data for each program.

Table 1 shows the measurements of each test program for a 1kB cache. The static measurements reflect the percentage of always hits, always misses, first misses, and conflicts out of the total number of instructions in the function instance tree. It can be seen that a large number of hits and misses can be predicted statically. The number of always hits is slightly above 70% in average and does not change significantly with varying cache sizes. The number of first misses increases for larger caches while conflicts and misses decrease at the same time. This can be explained as follows. First misses occur when a program line without any conflicts is placed in cache on its first reference and remains in cache thereafter. For very small caches, always misses dominate due to capacity misses. For medium-sized caches, program lines tend to conflict with one another more frequently resulting in more conflict instructions. As the programs begin to fit into cache, fewer program lines are in conflict and more references become first misses due to the increased cache capacity. In the worst case, only every sixth instruction is statically predicted as a conflict and will have to be simulated at execution time. At best, there are virtually no conflicts and almost the entire runtime simulation can be performed using efficient frequency counters.

Column 6 indicates the percentage of measurement points required for our method versus the number of measurement points inserted in a conventional cache simulation (*i.e.*, one measurement point per basic block). Our method requires only 76% of the measurement points required for the traditional trace-driven methods, *i.e.* about 24% fewer measurement points statically. The run-time savings (column 7) are even higher, requiring only about 69% of the measurement points executed under traditional trace-driven cache simulation. The additional dynamic savings are due to reducing sequences of basic blocks inside loops to fewer UPs, sometimes just to a single UP.

The static cache simulation results were verified (for each program execution and cache size) by ensuring that the exact same number of hits and misses were produced as obtained by traditional trace-driven cache analysis. As the cache size increases, the hit ratio (column 8) increases as well. Column 9 and 10 represent the quotient of the execution time of a program with

	Static					Dynamic			
Name	Hit	Miss	Firstmiss	Conflict	Measure Pts.		Hit Ratio	Trace	**SSim**	Conflict
cachesim	70.83%	6.99%	0.70%	21.48%	73.38%	60.56%	77.19%	8.41	**1.53**	34.12%
cb	79.03%	2.35%	0.00%	18.63%	89.62%	65.61%	93.84%	33.56	**3.51**	30.67%
compact	70.12%	4.96%	0.12%	24.80%	68.89%	56.56%	92.90%	22.29	**2.31**	21.34%
copt	70.89%	7.41%	7.03%	14.67%	84.19%	74.88%	93.64%	16.43	**1.58**	30.00%
dhrystone	70.03%	10.71%	7.30%	11.96%	81.61%	72.73%	83.73%	19.89	**1.31**	16.01%
fft	74.07%	4.85%	16.42%	4.66%	78.43%	74.08%	99.95%	5.79	**0.95**	8.80%
genreport	70.61%	9.95%	5.61%	13.84%	71.58%	81.31%	97.45%	13.57	**1.91**	28.92%
mincost	72.79%	9.96%	1.14%	16.11%	83.19%	76.27%	89.08%	23.47	**2.23**	30.67%
sched	67.65%	5.06%	0.09%	27.19%	73.16%	58.29%	96.41%	25.90	**3.62**	42.01%
sdiff	68.94%	12.06%	0.89%	18.11%	72.13%	77.82%	97.61%	32.10	**3.99**	28.40%
tsp	72.61%	13.50%	3.88%	10.01%	64.08%	58.67%	86.98%	5.70	**1.19**	17.63%
whetstone	75.70%	12.84%	0.24%	11.22%	70.49%	68.25%	100.00%	13.44	**1.36**	23.56%
average	71.94%	8.39%	3.62%	16.06%	75.90%	68.75%	92.40%	18.38	**2.12**	26.01%

Table 1: Measurements for 1kB Direct-Mapped Cache

instrumentation over the execution time for the same program without instrumentation. Column 9 refers to a trace-driven method that has been optimized such that the cache simulator is only called once per basic block[4]. Column 10 shows that cache simulation via static cache simulation is more efficient than the trace-driven method.[5] Column 11 refers to the analysis via static cache simulation. The percentage of conflicts (out of all instruction references) simulated at execution time is shown in the last column.

Figure 5 shows the overhead for different cache sizes. With the traditional trace-driven method, the execution time of instrumented programs is 14x to 24x slower than the execution time of regular programs without instrumentation. The overhead for the new method using static cache simulation is much lower, only a factor of 1.1 to 2.8. This overhead depends slightly on the ratio of program size and cache size. The variation can be explained as follows.

Let the *conflict degree* be the number of program lines that map into the same cache line. This is a useful term to characterize the size of shared path states (SPSs) and the execution overhead due to order-dependent simulation. For small caches, the conflict degree is relatively small. Many references will result in always misses due to a lack of cache capacity, which require only efficient frequency counting. For medium-sized caches, the conflict degree increases, peaking at a 512B cache for this test set, while always misses decrease. This requires an increased number of dynamically simulated state transitions for conflicts. For larger caches, capacity misses and the conflict degree of program lines decrease. They are replaced by first misses. With a diminishing number of conflicts for large caches, the size of SPSs decreases as the cache size increases. In other words, fewer and fewer conflicting program lines map into the same cache lines so that less instrumentation code to update conflicting SPSs is needed since hardly any conflicts remain. Thus, the cache simulation at execution time can be reduced to simple frequency counting, which imposes a much lower overhead than conventional cache simulation. To summarize this discussion, it is observed that the new method requires slightly more execution overhead for small caches than for large caches since more SPSs have to be updated dynamically.

The new method outperforms conventional trace-driven cache simulation by almost an order of a magnitude without compromising the accuracy of measurements. Even the best results published in [16] required an overhead factor of 2-15 over uninstrumented code for hit ratios between 96% and 99%. This highly tuned traditional method required a recompilation pass for

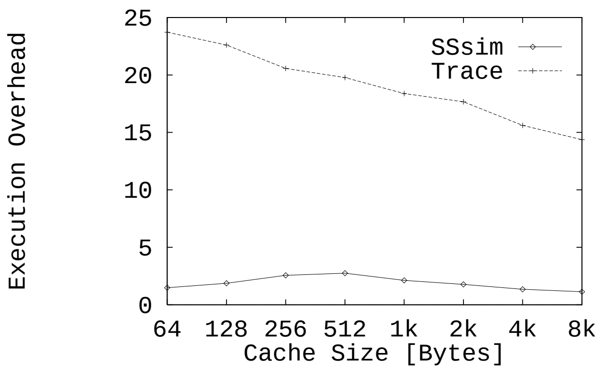


Figure 5: Average Overhead for varying Cache Sizes

[4]We used a traditional trace-driven method similar to "Technique B" in [16] but our version was probably finer tuned.

[5]For `tsp` in column 10, the instrumented code ran faster than the uninstrumented program, *i.e.* the ratio was smaller than 1. These results were reproducible. They may be caused by the different placement of code due to instrumentation, resulting in fewer misses for frequently executed loops.

better instrumentation. Under all conditions, the new method using static cache simulation outperforms the best traditional trace-driven methods published.

8 Future Work

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. Finally, data cache behavior could be analyzed statically as well under certain restrictive conditions, such as absence of heap allocation and pointers. There are several other applications of static cache simulation. For example, the worst-case execution time of real-time programs can be predicted more precisely for architectures with caches [2]. Other applications include detailed profiling and tracking of execution time for a real-time debugger[13].

9 Conclusion

A new method to evaluate instruction cache performance was designed and implemented. The cache performance of programs for various cache configurations can be obtained without recompiling the analyzed program. No special operating system support or dedicated registers are required. The new method outperforms conventional trace-driven cache simulation by almost an order of a magnitude without any loss of accuracy of the measurements. By making extensive use of static cache simulation and reducing code instrumentation to simple frequency counting in many places, this method reduces the execution overhead of analyzed programs to a factor of 2 on average. In addition, different cache sizes and resulting hit ratios have little influence on the overhead. Therefore, one can conclude that instruction cache analysis via static cache simulation is a general method to quickly obtain accurate measurements outperforming any other published methods.

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*, pages 172–181, December 1994.

[3] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[4] 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.

[5] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *International Symposium on Computer Architecture*, pages 270–279, May 1990.

[6] S. J. Eggers, D. R. Keppel, E. J. Koldinge, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–47, 1990.

[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] M. Huguet, T. Lang, and Y. Tamir. A block-and-actions generator as an alternative to a simulator for collecting architecture measurement. In *ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques*, pages 14–25, June 1987.

[10] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice & Experience*, 13(8):671–685, August 1983.

[11] F. Mueller. *Static Cache Simulation and its Applications.* PhD thesis, Dept. of CS, Florida State University, July 1994.

[12] F. Mueller and D. B. Whalley. Efficient on-the-fly analysis of program behavior and static cache simulation. In B. Le Charlier, editor, *Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 101–115. Springer, September 1994.

[13] 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.

[14] B. L. Peuto and L. J. Shustek. An instruction timing model of CPU performance. In *International Symposium on Computer Architecture*, pages 165–178, March 1977.

[15] D. B. Whalley. Fast instruction cache performance evaluation using compile-time analysis. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–22, June 1992.

[16] D. B. Whalley. Techniques for fast instruction cache performance evaluation. *Software Practice & Experience*, 19(1):195–203, January 1993.

[17] C. A. Wiecek. A case study of VAX-11 instruction set usage for compiler execution. In *Architectural Support for Programming Languages and Operating Systems*, pages 177–184, March 1982.