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

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

Abstract

This paper introduces a new method for instruction cache analysis which 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 which 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 which increment frequency counters. The counters are typi-

cally associated with a basic block and 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

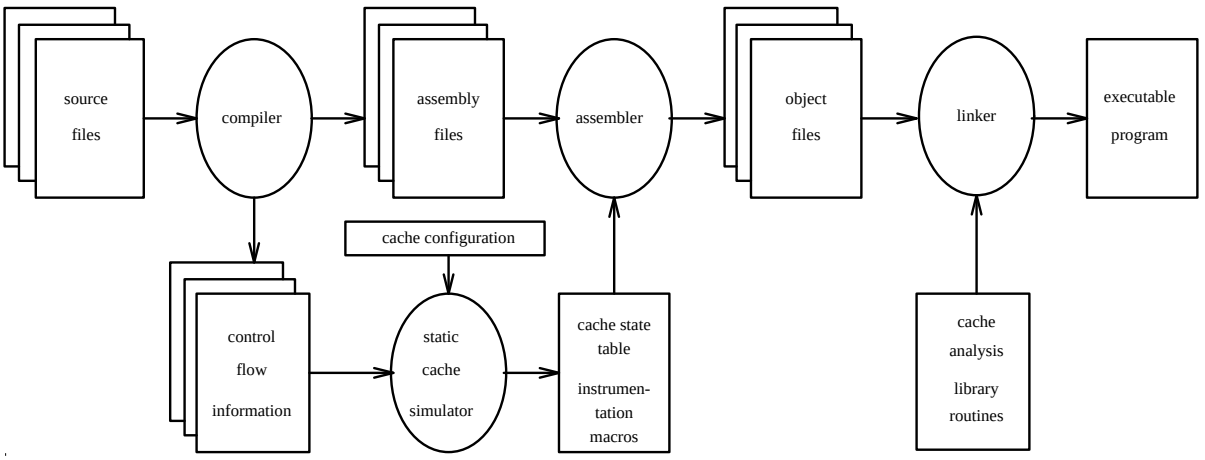


Figure 1: Overview of Static Cache Simulation

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. 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 subroutine 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 to an executable program and links in library routines which produce the final report of the cache analysis at run time[1].

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

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 which depend on the call site and call sequence.

If the simulator cannot determine statically if a line results hit or miss, 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.

The improvements of fast instruction cache analysis using static cache simulation over traditional trace-driven cache simulation are summarized below.

Unique Paths: The code is instrumented at UPs rather than at basic blocks. This reduces the number of instrumentation points.

Static Cache Simulation: A large percentage of the instruction references are statically identified as always hits and always misses. These references will not have to be simulated at run time at all.

Function Instances: The static cache simulation is refined by taking the call site of a function invocation into account. This reduces the simulation overhead required at run time even more since the behavior of more instruction references can be statically identified.

Inline Code Instrumentation and Frequency Counters: The remaining instruction references are simulated at run time by inlining short sequences of instrumentation code for each UP rather than calling a tracing routine. The compiler identifies the live registers at the instrumentation point. A set of unused registers is provided for the instrumented code to avoid unnecessary saves and restores. The instrumentation consists of incrementing simple frequency counters and state transitions.

This paper is structured as follows: Section 2 reviews related work in the area. Section 3 discusses interaction of our method with a compiler. Section 4 introduces static cache simulation. Section 5 explains how the program code is instrumented. Section 6 describes the final calculation of hits and misses. Section 7 provides a quantitative analysis of our method. Section 8 outlines future work. Section 9 presents the conclusions of this study. The Appendix provides further details on code instrumentation.

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 [PS77, Wie82, HLT87].

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 [BKW90] 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 8 to 12 times normal execution time were reported for the trace generation. Analysis of the trace was stated to require at least 10 times the overhead of the generation of the trace (or about 100 times slower than normal execution time).

Eggers *et. al.* [EKKL90] also used the technique of inline tracing to generate a trace of addresses to 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 3 times the normal execution time. In addition, writing the buffers to disk required a factor of 10 times normal execution time. The postprocessing pass, which generates the complete trace from the subset of addresses stored, was much slower and produced about 3,000 addresses per second. No information was given on the overhead required to actually analyze the cache performance.

Ball and Larus [BL92, Lar83] 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 to regenerate the full trace required 19-60 times the normal execution time.

Whalley [Wha92, Wha93] evaluated a set of techniques to 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 15 times normal execution time for average hit ratios of 96% and 2 times 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 [ASU86] (vertices) connected by control-flow transitions (edges) that contain at least one unique transition, *i.e.* a transition which does not occur in any other UP[2]. In this study, UPs were restricted to not cross loop boundaries, function boundaries, or calls. The notion of UPs allows one to identify and minimize 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 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 which 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 [HP90]. 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.

[2]Due to space limitations, a formal definition cannot be given here but can be found in another paper by the authors.

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 currently 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. The calculation is performed by repeated traversal of the call graph's functions, their function instances, and the UPs of each function's control-flow graph.

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 which 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 which may occur at execution time and the exponential growth of such an exhaustive set during simulation.

For a given function instance, each instruction i within a UP is categorized based on its position in the corresponding program line $l = i_0..i_{n-1}$ and on the corresponding abstract cache state s. The program line l maps into cache line c, denoted by $l \rightarrow c$. The UP is referred to as $UP_1(l)$, the path which contains line l. A UP_1 with line l may reach another UP_2 with line k within the combined control-flow graphs and call graph by one or more transitions, denoted as $UP_1(l) \stackrel{+}{\rightarrow} UP_2(k)$.

always-miss: A cache miss is predicted if

- $i = i_0$: instruction i is the first reference to program line l in $UP_1(l)$ 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 $UP_1(l)$. Or
- $i = i_0$: instruction i is the first reference to program line l in $UP_1(l)$,

- $l \in s$: 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.

conflict: A reference may result in either a cache hit or cache miss at execution time if

- $i = i_0$: instruction i is the first reference to program line l in $UP_1(l)$,
- $l \in s$: l is in the abstract cache state, and either
- $\exists_{k \neq l} \; k \in s \wedge k \rightarrow c$: another line k (which maps into the same cache line as l) is also in the abstract cache state, or
- $\exists_{UP_2(l) \neq UP_1(l)} \; UP_2(l) \stackrel{+}{\rightarrow} UP_1(l)$: another path UP_2 contains the same line l and reaches UP_1 in the control flow[3].

Line l is said to be in conflict with line k in this case such that during execution the cache line c may be alternately replaced by program line l and k.

first-miss: All other instructions are first misses.

4.3 Implementation

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

```
input_state(top) := all invalid lines
WHILE any change DO
   FOR each instance of a unique path 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

UP has an input and output state of program lines which 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

[3]This could be a path of a different function overlapping in line l, another path of the same function overlapping in line l, or the same path of a different function instance.

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.

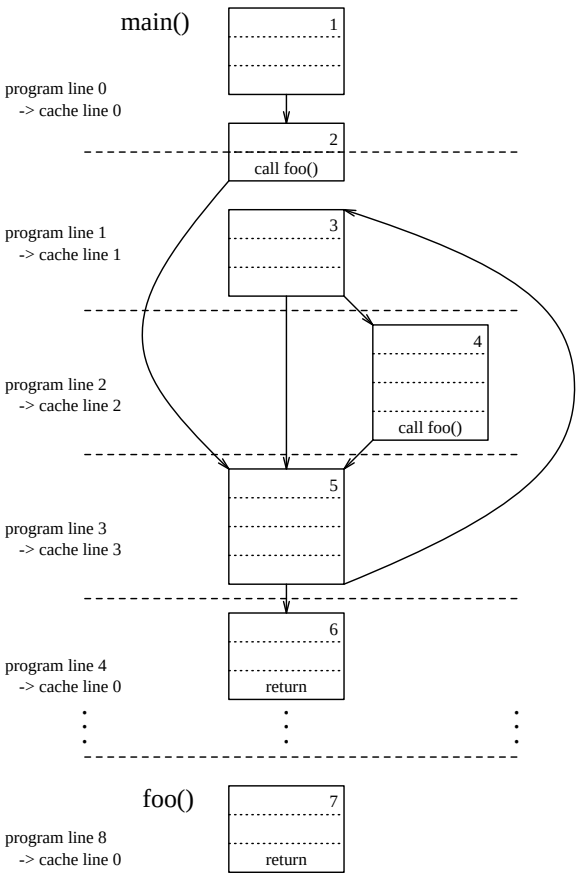
The calculation of these abstract cache states requires a time overhead comparable to that of data-flow analysis used in optimizing compilers and a space overhead linear to the number of program lines, UPs, and function instances. The correctness of the algorithm for data-flow analysis is discussed in [ASU86].

Figure 3 illustrates a simple example of calculating input and output states. The paths are restricted to basic blocks to simplify the example. Note that the immediate successor of a path with a call is the first path in that instance of the called function. Assume there are 4 cache lines and the line size is 16 bytes (4 instructions). Path 7a is the instance from the first call to foo in path 2 and path 7b is the instance from the second call in path 4. Two passes are required to calculate the input and output states of the paths, given that the paths are processed in the order shown in Figure 3. Only the states of paths 3, 4, 5, and 6 changed on the second pass. Pass 3 results in no more changes.

After determining the input states of all paths, each instruction is categorized according to the criteria specified in the previous section. By inspecting the input states of each path, 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 cache simulation determines that the first instruction in UP 3 will always be in cache (always hit) due to spatial locality and due to the absence of conflicting paths. It also determines that the first instruction in path 7b will always be in cache (always hit) due to temporal locality. Finally, the static cache simulation determined that the first instructions in paths 4 and 5 will not be in cache on their first reference, but will always be in cache on subsequent references (first miss).

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 [Hil88]. 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.

The execution of the static cache simulator depends on the complexity of the analyzed control flow. The static simulator has not been tuned yet to reduce the execution time. However, preliminary results indicate that the execution time of the static simulator will only be a small portion of the execution time of instrumented programs. This can be explained by the fact that the overhead of static cache simulation depends on the control flow of a program while the program's execution time is proportional to the length of the trace.




```
 "I" = invalid

  cache line 0 1 2 3 0 1 2 3 0 1 2 3 0                0 1 2 3 0 1 2 3 0 1 2 3 0
program line I I I I 0 1 2 3 4 5 6 7 8                I I I I 0 1 2 3 4 5 6 7 8

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

  cache line 0 1 2 3 0 1 2 3 0 1 2 3 0                0 1 2 3 0 1 2 3 0 1 2 3 0
program line I I I I 0 1 2 3 4 5 6 7 8                I I I I 0 1 2 3 4 5 6 7 8

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

Figure 3: Example with Flow Graph

4.4 Recomposition

After decomposing the program into function instances and UPs, there still remain lines which are analyzed to be in conflict with another line. It is inevitable to maintain information at run time to determine which program line that 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).

Naively, a path state may be kept on the most specialized level (for every 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*. Second, local path states of neighboring UPs which share at least one instruction are merged into one *shared path state* (SPS).

The former merger allows uniform instrumentation of code rather than distinguishing instances dynamically at every instrumentation point or replicating code for each instance. In both cases, the overhead of dynamic simulation for conflicts is greatly reduced. 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.

5 Code Instrumentation

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 Shared Path States

For each SPS, a state field is generated in the state table (see Figure 4 in the Appendix). 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. If a SPS is constant at run time (no conflicting lines), then the state field is omitted from the table.

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 (see Figure 5 in the Appendix). 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, which is discussed in the context of the path macros.

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 of the call. The function instance can thereby be identified by path macros (see Figure 7 in the Appendix).

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.

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[4]. Figure 8 in the Appendix depicts examples of path macro code.

5.5 First Miss Table

If a path of a function instance contains a line which is classified as a first miss, an entry for this line is created in the first miss table (see Figure 6 in the Appendix). 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

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

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 and Misses based on Frequency Counters

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 then also multiplied by the corresponding frequency (see Figure 5 in the Appendix). A zero index indicates that all conflicting lines are cached while the last index corresponds to misses of all conflicting lines.

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 index 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.

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 (see Figure 6 in the Appendix), the counters of corresponding paths (and instances) are checked. If the frequency of one of the 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 listed in Table 1. The measurements were produced by modifying the back-end of an optimizing compiler 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 1kB, 2kB, 4kB, and 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 (see column 3). This

Name	Description	Size [bytes]	Instructions exec.	Instructions in tree	Measure Pts. static	Measure Pts. exec.
cachesim	Cache Simulator	8,460	2,995,817	14,357	85.37%	70.01%
cb	C Program Beautifier	4,968	3,974,882	14,012	96.54%	54.75%
compact	Huffman Code Compression	5,912	13,349,997	4,237	74.07%	63.92%
copt	Rule-Driven Peephole Optimizer	4,148	2,342,143	1,599	87.18%	72.02%
dhrystone	Integer Benchmark	1,916	19,050,093	697	89.66%	76.62%
fft	Fast Fourier Transform	1,968	4,094,244	581	78.43%	59.20%
genreport	Detailed Execution Report Generator	17,720	2,275,814	9,983	79.86%	56.92%
mincost	VLSI Circuit Partitioning	4,448	2,994,275	2,385	91.60%	68.14%
sched	Instruction Scheduler	8,272	1,091,755	6,256	78.56%	75.64%
sdiff	Side-by-side Differences between Files	7,288	2,138,501	19,214	77.75%	57.89%
tsp	Traveling Salesman	4,724	3,004,145	1,542	68.31%	62.32%
whetstone	Floating point benchmark	4,816	8,520,241	1,737	73.77%	62.75%
average		6,220	5,485,992	6,383	81.76%	65.02%

Table 1: Test Set of C Programs

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 each program comprised a range of 1 to 19 million using realistic input data for each program (see column 4) . Column 5, instructions in the tree, refers to the static number of instructions in the program after expanding the call graph to a function instance tree. The measurements in the next section are based on this number.

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 81% of the measurement points required for the traditional trace-driven methods, *i.e.* about 19% fewer measurement points statically. The run-time savings (column 7) are even higher, requiring only about 65% 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.

7.1 Static Analysis

This section describes the analysis that was performed statically on the test programs. Table 2 shows 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 hits is slightly above 70% in average and does not change as the cache size is varied. Always hits are dominated by hits due to subsequent references within a program line (past the first reference) which do not depend on the cache size. The slight variation of always hits is mainly due to multiple function instances. Consider a first call to a function which will cache the function's program lines. Subsequent calls always result in hits for these program lines, given a sufficiently large cache capacity such that no conflicting lines could be executed between the two function calls.

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

	1kB cache				2kB cache			
Name	Hit	Miss	Firstmiss	Conflict	Hit	Miss	Firstmiss	Conflict
cachesim	70.47%	7.05%	0.72%	21.77%	71.30%	5.20%	1.99%	21.52%
cb	79.19%	2.25%	0.00%	18.56%	80.90%	0.89%	0.00%	18.21%
compact	69.15%	5.43%	0.09%	25.32%	69.22%	2.27%	0.09%	28.42%
copt	70.54%	6.63%	6.57%	16.26%	70.92%	5.19%	14.76%	9.13%
dhrystone	69.44%	12.20%	6.46%	11.91%	70.16%	5.45%	24.39%	0.00%
fft	73.49%	4.65%	17.56%	4.30%	75.04%	3.79%	21.17%	0.00%
genreport	71.50%	9.76%	6.14%	12.60%	72.31%	9.05%	7.13%	11.51%
mincost	71.36%	9.64%	1.09%	17.90%	73.71%	5.53%	5.16%	15.60%
sched	66.10%	5.66%	0.08%	28.16%	66.42%	2.99%	0.08%	30.51%
sdiff	68.77%	12.50%	0.97%	17.76%	69.17%	10.44%	2.52%	17.87%
tsp	72.31%	14.46%	4.28%	8.95%	72.63%	10.44%	9.34%	7.59%
whetstone	75.13%	12.55%	0.23%	12.09%	75.99%	8.81%	0.23%	14.97%
average	71.45%	8.57%	3.68%	16.30%	72.31%	5.84%	7.24%	14.61%

	4kB cache				8kB cache			
Name	Hit	Miss	Firstmiss	Conflict	Hit	Miss	Firstmiss	Conflict
cachesim	71.83%	4.53%	14.31%	9.32%	72.38%	3.59%	24.00%	0.04%
cb	81.23%	0.29%	15.57%	2.90%	81.27%	0.03%	18.71%	0.00%
compact	69.29%	0.47%	11.42%	18.81%	69.62%	0.47%	29.90%	0.00%
copt	71.42%	4.75%	23.83%	0.00%	71.42%	4.75%	23.83%	0.00%
dhrystone	70.16%	5.45%	24.39%	0.00%	70.16%	5.45%	24.39%	0.00%
fft	75.04%	3.79%	21.17%	0.00%	75.04%	3.79%	21.17%	0.00%
genreport	72.49%	8.63%	8.66%	10.21%	72.75%	8.14%	11.11%	7.99%
mincost	74.51%	2.77%	22.47%	0.25%	74.59%	2.77%	22.64%	0.00%
sched	66.46%	1.85%	2.33%	29.35%	66.61%	1.17%	32.23%	0.00%
sdiff	69.26%	10.24%	17.71%	2.80%	69.28%	10.02%	20.70%	0.00%
tsp	73.02%	5.58%	21.27%	0.13%	73.09%	5.58%	21.34%	0.00%
whetstone	76.45%	1.61%	17.67%	4.26%	77.32%	0.46%	22.22%	0.00%
average	72.60%	4.16%	16.73%	6.50%	72.79%	3.85%	22.69%	0.67%

Table 2: Static Measurements

cache thereafter. For smaller caches, program lines tend to conflict with another more frequently. As the programs begin to fit into cache, fewer program lines are in conflict. In the worst case, only every six 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.

7.2 Dynamic Analysis

Table 3 summarizes the dynamic measurements taken for the test programs. For a given cache size and program execution, our method produced exactly the same number of hits and misses that were obtained by traditional trace-driven cache analysis. As the cache size increases, the hit ratio (column 2) increases as well. Column 3 and 4 represent the quotient of the execution time of a program with instrumentation over the execution time for the same program without instrumentation. Column 3 refers to a trace-driven method which has been optimized such that the cache simulator is only called once per basic block[5]. Column 4 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.

With the traditional trace-driven method, the execution time of instrumented programs averages about 20 times slower than the execution time of regular programs without instrumentation. The overhead for our method using static cache simulation is much lower, only factor of 1.3 to 2.0. 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 which 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 cache sizes, the conflict degree is larger and there are more always misses due to capacity misses. As the cache size increases, capacity misses and the conflict degree of program line decrease. They are replaced by first-misses. With a diminishing number of conflicts for large caches, the 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. Consequently, less instrumentation code to update conflicting SPSs is needed. Finally, for large caches, hardly any conflicts remain. Thus, the cache simulation at execution time can be reduced to simple frequency counting which imposed a much lower overhead than conventional cache simulation. To summarize this discussion, we observe that our method requires slightly more execution overhead for small caches than for large caches since more SPSs have to be updated dynamically.

In general, 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 [Wha93] required an overhead factor of 6-16 over uninstrumented code for hit ratios between 96% and 99%. This highly tuned traditional method used code motion to move measurement points out of loops and required a recompilation pass. Under all conditions, this new method using static cache simulation outperforms over the best traditional methods published.

[5] We used a traditional trace-driven method similar to "Technique B" in [Wha93] but our method was probably more tuned.

	1kB cache				2kB cache			
Name	Hit Ratio	Trace	SSim	Conflict	Hit Ratio	Trace	SSim	Conflict
cachesim	77.19%	8.51	1.61	34.67%	86.08%	8.04	1.56	37.35%
cb	93.84%	34.30	3.49	31.19%	99.06%	33.80	2.68	32.50%
compact	92.90%	22.70	2.64	33.39%	96.79%	21.89	2.05	33.60%
copt	93.64%	16.63	1.52	34.06%	98.10%	16.14	1.12	11.07%
dhrystone	83.73%	19.34	1.31	16.54%	100.00%	22.45	1.45	0.00%
fft	99.95%	7.88	1.15	8.80%	100.00%	7.42	1.13	0.00%
genreport	97.49%	13.51	1.56	25.16%	98.10%	12.70	1.37	23.79%
mincost	89.08%	24.21	2.03	29.13%	95.68%	21.40	1.54	26.20%
sched	96.41%	24.18	3.43	57.14%	99.75%	26.91	2.89	57.72%
sdiff	97.61%	27.48	2.88	29.07%	99.38%	29.15	2.27	28.59%
tsp	86.98%	5.51	1.09	17.75%	96.94%	5.17	1.02	9.68%
whetstone	100.00%	13.46	1.26	23.78%	100.00%	13.26	1.39	25.61%
average	92.40%	18.14	2.00	28.39%	97.49%	18.19	1.71	23.84%

	4kB cache				8kB cache			
Name	Hit Ratio	Trace	SSim	Conflict	Hit Ratio	Trace	SSim	Conflict
cachesim	99.22%	7.52	1.09	14.21%	99.98%	6.55	1.07	1.67%
cb	99.83%	26.59	1.70	7.63%	99.99%	27.88	1.45	0.00%
compact	99.82%	21.07	1.60	23.11%	100.00%	37.28	2.00	0.00%
copt	99.99%	18.54	1.18	0.00%	99.99%	16.34	1.16	0.00%
dhrystone	100.00%	23.50	1.46	0.00%	100.00%	22.16	1.32	0.00%
fft	100.00%	7.70	1.13	0.00%	100.00%	7.09	1.09	0.00%
genreport	98.11%	12.63	1.35	23.67%	99.93%	12.18	1.19	20.22%
mincost	99.99%	26.50	1.50	0.05%	99.99%	28.28	1.70	0.00%
sched	99.90%	23.93	1.88	49.44%	99.96%	27.00	1.38	0.00%
sdiff	99.99%	58.58	2.27	3.10%	99.99%	34.04	1.56	0.00%
tsp	99.99%	5.20	1.10	0.00%	99.99%	4.50	1.02	0.00%
whetstone	100.00%	12.73	1.20	12.15%	100.00%	13.06	1.12	0.00%
average	99.74%	20.37	1.46	11.11%	99.99%	19.70	1.34	1.82%

Table 3: Dynamic Measurements

8 Future Work

The definition of first misses is very conservative at this point. The static analysis could be extended based on a more detailed picture of the loop structure of the program. A refined notion of first misses could then refer to a miss of a program line on loop entry and consecutive hits inside the loop. This would be valuable when a number of program lines mapping into the same cache line can be in cache at loop entry but only one of these program lines lies inside the loop.

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, pointers, etc.

The idea of static cache simulation does not have to interact with a compiler, although compiler support seems to facilitate the task. Yet, it is possible to analyze the control flow of an arbitrary executable and to modify the binary by inserting instrumentation code (similar to the work in [LB92]). This would allow library code to be measured as well.

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. Other applications include detailed profiling and tracking of execution time for a real-time debugger.

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 1.3 to 2.0. In addition, different cache sizes and resulting hit ratios have little influence on the overhead. Therefore, instruction cache analysis via static cache simulation is a general method to quickly obtain accurate measurements.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addision-Wesley, 1986.

[BKW90] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *Annual International Symposium on Computer Architecture*, pages 270–279, May 1990.

[BL92] 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.

[EKKL90] 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.

[Hil88] M. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(11):25–40, December 1988.

[HLT87] 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.

[HP90] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[Lar83] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice & Experience*, 13(8):671–685, August 1983.

[LB92] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. TR 1083, University of Wisconsin, March 1992.

[PS77] B. L. Peuto and L. J. Shustek. An instruction timing model of CPU performance. In *Annual International Symposium on Computer Architecture*, pages 165–178, March 1977.

[Wha92] 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.

[Wha93] D. B. Whalley. Techniques for fast instruction cache performance evaluation. *Software Practice & Experience*, 19(1):195–203, January 1993.

[Wie82] 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.

Appendix

```
___path3_cal_curstate:
        .word   12 ! SPS: 2 uncached lines at startup (1100b)
```

Figure 4: State Table Entry

```
! typical method based on frequency counter array
___path3_cal0_curstate: ! path 3 in function cal instance 0
        .word   0       ! AND mask
        .word   12      ! always hits  \
        .word   5       ! always misses > on each increment
        .word   0       ! first misses /
        .word   4       ! # shared path states
___path3_cal0:          ! frequency counts (counter array):
        .word   0       !   hits on line a and b
        .word   0       !   miss line a, hit line b
        .word   0       !   hit line a, miss line b
        .word   0       !   misses on line a and b

! alternative method for large path states
___path7_whet1_0_curstate:
        .word   4088    ! AND mask
        .word   23      ! hits
        .word   0       ! misses
        .word   1       ! first misses
        .word   -9      ! -(# shared path states)
___path7_whet10:
        .word   0       ! general frequency counter
        .word   0       ! counter for misses due to conflicts
```

Figure 5: Counter Table Entry

```
___conf_table:
        .word   ___path6_cal0_curstate
        .word   0
        .word   ___path8_cal1_curstate
        .word   ___path7_cal1_curstate
        .word   0
        ...
```

Figure 6: First Miss Table

The code emitted for call macros in Figure 7 performs the following operations. The callee's base address (instance) is dependent on the caller's base address (instance) and is determined by a sequence of compare and branch instructions. If the caller has only one instance, two instructions suffice. Otherwise six instructions are generated for each callee's instance corresponding to the current call site. (Notice that a `set` instruction is a synonym for the two instructions `sethi` and `or`.) Figure 7 depicts examples for both cases.

The number of instructions generated for a path macro varies. Figure 8 depicts examples for three cases. In the first case without conflicts, a frequency counter is simply incremented in three instructions. In the second case, conflicts are present. The SPS is used as an index into the frequency array, and the indexed counter is incremented. Then,

```
! one function instance
#define CALL18_main(base_in, base_out, temp) \
        set     ___inst_table_number1,%base_out ; \
___call_main18_cont:

! multiple function instances
#define CALL1_cal(base_in, base_out, temp) \
! first instance \
        set     ___inst_table_cal1,%temp ; \
        cmp     %temp,%base_in ; \
        sethi   %hi(___inst_table_jan12),%base_out ; \
        be      ___call_cal1_cont ; \
        or      %base_out,%lo(___inst_table_jan12),%base_out ; \
! next instance \
        set     ___inst_table_cal2,%temp ; \
        cmp     %temp,%base_in ; \
        sethi   %hi(___inst_table_jan14),%base_out ; \
        be      ___call_cal1_cont ; \
        or      %base_out,%lo(___inst_table_jan14),%base_out ; \
! last instance \
        set     ___inst_table_jan10,%base_out ; \
___call_cal1_cont:
```

Figure 7: Call Macros

the SPS is updated to reflect changes in the cached program lines. Thus, 12 instructions are required for updating the counter and SPS. In addition, three instructions are required for updating every conflicting SPS in the worst case. In the second example, there are two such conflicting SPSs. In the third case, the alternate code instrumentation is used. A loop counts the number of on-bits in the SPS combined with the AND mask. This number is added to the second counter entry while the first entry, the frequency counter, is simply incremented. Then, the SPS is updated to reflect changes in the cached program lines. Here, 16 instructions are required plus a maximum of 30 iterations of 7 instructions inside the bit-counting loop.

```
#define PATH1_cal(base, temp1, temp2) \
        ld      [%base+___path1_cal0-___inst_table_cal0],%temp2 ; \
        inc     %temp2 ; \
        st      %temp2,[%base+___path1_cal0-___inst_table_cal0] ; \

#define PATH4_cal(base, temp1, temp2) \
! increment counter in array index by SPS of current path \
        sethi   %hi(___path3_cal_curstate),%temp1 ; \
        ld      [%temp1+%lo(___path3_cal_curstate)],%temp2 ; \
        add     %temp2,%base,%temp1 ; \
        ld      [%temp1+___path4_cal0-___inst_table_cal0],%temp2 ; \
        inc     %temp2 ; \
        st      %temp2,[%temp1+___path4_cal0-___inst_table_cal0] ; \
! update SPS of current path \
        sethi   %hi(___path3_cal_curstate),%temp1 ; \
        ld      [%temp1+%lo(___path3_cal_curstate)],%temp2 ; \
        andn    %temp2,8,%temp2 ; \
        st      %temp2,[%temp1+%lo(___path3_cal_curstate)] ; \
! update two conflicting SPSs of other paths \
        set     ___state_table,%temp1 ; \
        ld      [%temp1+___path20_main_curstate-___state_table],%temp2 ; \
        or      %temp2,4,%temp2 ; \
        st      %temp2,[%temp1+___path20_main_curstate-___state_table] ; \
        ld      [%temp1+___path19_main_curstate-___state_table],%temp2 ; \
        or      %temp2,4,%temp2 ; \
        st      %temp2,[%temp1+___path19_main_curstate-___state_table] ; \

#define PATH16_whet1(base, temp1, temp2) \
! apply AND mask to SPS of current path \
        sethi   %hi(___path6_whet1_curstate),%temp1 ; \
        ld      [%temp1+%lo(___path6_whet1_curstate)],%temp2 ; \
        ld      [%base+___path16_whet1_0_curstate-___inst_table_whet10],%temp1 ; \
        andcc   %temp2,%temp1,%temp1 ; \
        bz      ___path16_whet1_nocnt;\
! count on-bits \
        mov     %g0,%temp2 ; \
___path16_whet1_cnt: ; \
        bz      ___path16_whet1_ccnt;\
        btst    1,%temp1 ; \
        srl     %temp1,1,%temp1 ; \
        bz      ___path16_whet1_cnt ;\
        tst     %temp1 ; \
        b       ___path16_whet1_cnt ; \
        inc     %temp2 ; \
___path16_whet1_ccnt: ; \
! add number of on-bits to accumulated number of conflicts which were misses \
        ld      [%base+4+___path16_whet10-___inst_table_whet10],%temp1 ; \
        add     %temp1,%temp2,%temp2 ; \
        st      %temp2,[%base+4+___path16_whet10-___inst_table_whet10] ; \
___path16_whet1_nocnt: ; \
! increment the general frequency counter \
        ld      [%base+___path16_whet10-___inst_table_whet10],%temp2 ; \
        inc     %temp2 ; \
        st      %temp2,[%base+___path16_whet10-___inst_table_whet10] ;
        %\
! update the SPS of the current path \
        sethi   %hi(___path6_whet1_curstate),%temp1 ; \
        ld      [%temp1+%lo(___path6_whet1_curstate)],%temp2 ; \
        andn    %temp2,12,%temp2 ; \
        st      %temp2,[%temp1+%lo(___path6_whet1_curstate)] ; \
```

Figure 8: Path Macros