# Fast Instruction Cache Performance Evaluation Using Compile-Time Analysis

DAVID B. WHALLEY
*Department of Computer Science, Florida State University, Tallahassee, FL 32306, U.S.A.*

## SUMMARY

**Cache performance has become a very crucial factor in the overall system performance of machines. Effective analysis of a cache design requires the evaluation of the performance of the cache for typical programs that are to be executed on the machine. Recent attempts to reduce the time required for such evaluations either result in a loss of accuracy or require an initial pass by a filter to reduce the length of the trace. This paper describes techniques that overcome these problems for instruction cache performance evaluation. Information calculated during the compilation is used to reduce the number of references in the trace. Thus, in effect references are stripped before the initial trace is generated. These techniques are shown to significantly reduce the time required for evaluating instruction caches with no loss of accuracy.**

KEY WORDS: Instruction Cache  Cache Simulation  Trace Generation  Trace Analysis

## INTRODUCTION

The time required to generate and analyze trace data is proportional to the number of references in the trace [Smi77]. Since realistic program traces can be quite lengthy, a trace is often only collected from a portion of the program's execution. However, it has been shown that the cache performance can vary greatly in different portions of a program's execution [BKW90]. Cache performance measurements obtained when unrealistic input data is used to shorten the length of the trace would also be of questionable value.

Many recently designed machines either have separate instruction and data caches or forego using a cache for data references and attempt to minimize data references only by using registers. There are many advantages to having an instruction-only cache. Most machines do not allow modification of instructions during the execution of a program. By only allowing read accesses to a cache, the design of an instruction cache is simplified. Also, instruction references typically have higher locality than data references. By using separate caches, the designers of a system can offer different configurations for each cache which may improve the overall performance. Since the type of reference, instruction or data, issued to the memory system is known by the CPU, there can be separate ports for instructions and data. Thus, the bandwidth between cache memory and the CPU can be improved.

Since instruction caches have become more common, there has been much recent work that attempts to reorganize the code of programs to improve the instruction cache performance [HwC89, McF89, PeH90]. In fact, most compiler optimizations can affect instruction cache performance since the optimizations can change the order and number of instructions that are executed by a program. Since these optimizations can affect different portions of a program's execution, the effect of a compiler optimization on instruction cache performance can only be accurately evaluated if the complete trace of instruction addresses from the program's execution is used.

This paper describes techniques that use compile-time analysis to reduce the time required for evaluating instruction cache performance. Unlike many data references, the address of each instruction remains the same during the entire execution of a program. Information can therefore be calculated prior to the instruction cache simulation that can be used to reduce the number of references that the cache simulator needs to process. Thus, in effect address references are being stripped before the initial trace is generated.

## RELATED WORK

Traditionally, the problem of evaluating the cache performance during the execution of a program has been separated into two tasks, trace generation and trace analysis. The first task is to generate the trace of addresses that will be presented to the cache. The address trace is typically either written to secondary storage, such as a file on disk, or stored in a trace buffer in memory. The second task is to analyze the addresses that were generated. This analysis is usually accomplished through the use of a separate program that reads the generated address trace and simulates the behavior of the cache.

Trace-driven cache simulation has long been the primary method used to analyze the performance of a cache. Simple approaches for generating trace data and simulating caches, however, can be both very time and space consuming. Two common methods used for generating trace data are forcing a program to trap after the execution of each instruction or to record references while simulating the execution of each instruction. Each of these methods can result in a program executing a 1000 times slower than normal execution [PeS77, Wie82, HLT87]. Trace data is typically stored in secondary storage and then later read by a cache simulator that will perform the analysis of the data. Realistic trace data, however, requires at least several million references which may make infeasible the use of disk as the storage media [BKW90]. Therefore, there has been much work on the problem of reducing the space and time requirements for trace-driven simulation.

One faster trace generation method is to modify the microcode of a microprogrammed machine. Relative to techniques using traps or simulated program execution, this method does not impose a large run-time penalty [ASH86]. The microcode of a machine, however, is often not accessible to the typical user. Even when it is accessible, it often requires great expertise to modify without adversely affecting the operation of the machine. Also, modification of microcode would not be applicable for machines which are not microprogrammed (e.g. many RISCs).

Program instrumentation, or inline tracing, is a technique that requires little overhead for generating a trace of addresses [HLT87, BKW90, EKK90]. Instructions are inserted to record addresses during the program's execution and must not change the normal execution behavior of the program. Therefore, the values in data registers or the condition codes may have to be saved and restored.

Unfortunately, even with fast trace generation techniques, evaluating the cache performance of a program's execution can be quite time-consuming since the largest factor in the time required for cache performance evaluation is analyzing the trace of addresses. Even after tuning a cache simulator for a specific cache configuration, a cache simulator can still require an order of magnitude more time than generating the trace itself [BKW90]. Therefore, there has been much attention given to reducing the number of references that need to be traced.

There have been several methods proposed to improve cache simulation times by reducing the number of references in the trace. One method for reducing the length of the trace data is called trace stripping [Puz85]. This approach first simulates a direct-mapped cache and records only the references that are misses since hits do not result in changes to the cache state. The reduced trace can then be used to simulate caches with a greater number of sets and associativity as long as the line size is not changed and context switches are not introduced. Unfortunately, this technique requires the entire trace to be processed by the cache simulator once. Furthermore, the reduced trace may still be quite lengthy, which can result in large files and slow simulations. There have also been several methods that allow different cache configurations to be evaluated during a single simulation [WaB90, HiS89].

Another method to avoid processing the entire trace of references is to instead use several discrete samples of traces from the program execution to predict cache performance measures [LPI88]. While a significant loss of accuracy may not occur, the method may not have the desirable accuracy for measuring the effect of a new compiler optimization or reorganization technique. For instance, the optimization may be applied on a section of code that when executed is not in the samples of references that are collected.

On-the-fly analysis is a technique that avoids the I/O associated with storing the generated trace and retrieving the trace for input to the cache simulator. In this approach either the cache simulator is a separate process that reads a trace buffer containing the trace [BKW90] or the cache simulator is linked directly to the program and trace information is received as arguments via function calls [StF89]. Even though the space and I/O requirements are diminished, the trace analysis can still be quite time-consuming since the entire trace is being processed.

## TECHNIQUES FOR REDUCING INSTRUCTION CACHE EVALUATION TIMES

An optimizer of a compiler system was modified to be able to generate and analyze trace data. The program being measured is compiled in two passes. The first compilation pass serves to determine the address of each instruction. Rather than using a table lookup method, which may be rather complicated for highly encoded CISC architectures, a label is inserted before and after each instruction. A call is inserted at the beginning of the main function to invoke a routine that will dump out the size of each instruction, the difference between each pair of labels. Even the size of branch instructions, which may vary depending upon the distance to the branch target, may be accurately determined since no other trace instructions are inserted at this point. The first compilation pass also stores information about each function to be used by some of the techniques described later in the paper.

The second compilation pass inserts the instructions to invoke the cache simulator during the program's execution. The values of scratch registers and condition codes may have to be saved prior to the inserted instructions and restored afterwards. By using the data-flow information already calculated and available in the optimizer, these saves and restores are only inserted when necessary. A call to print the cache performance report is also inserted before any return instructions in the `main` function or calls to the `exit` function anywhere in the program. The cache simulator is linked with the program to allow on-the-fly trace analysis to occur while the program is being executed.

The techniques used to reduce the number of cache references processed by the cache simulator are described in the following sections. The cache references not processed are those references which are ascertained to be cache hits and will not change the state of the cache. Examples associated with the techniques are given in Motorola 68020 assembly code. Each technique builds upon the previously presented techniques. Technique A is a straight-forward approach. Techniques B and C recognize spatial locality to reduce the length of the trace and methods similar to these have been used in previous studies [MiF88, EKK90]. Techniques D-G use cache configuration information and the control flow of the program to avoid processing additional references due to both spatial and temporal locality.

**Technique A**

For Technique A, a call to a trace routine is inserted before each basic block in the program. Note, that basic blocks for tracing are delimited by labels and branching instructions, including calls. A block number, that uniquely identifies the basic block being executed, is passed to the trace routine which uses the number to access information associated with the block.[1] The trace routine in turn interfaces with the cache simulator, passing it the address and size of each instruction within the block. Since the cache simulator allows the option of periodic context switches, a simulated context switch invalidating the entire instruction cache can occur between two instructions in the same basic block. Assembly code with inserted instructions for Technique A is given in Figure 1.

```
 ...
L142:
 pea   #145            /* push block number 145 */
 jbsr  _traceblknum    /* call trace routine */
 addql #4,a7           /* adjust stack pointer */
 ...
 jne   L67
 pea   #146            /* push block number 146 */
 jbsr  _traceblknum    /* call trace routine */
 addql #4,a7           /* adjust stack pointer */
 ...
```

Figure 1: Assembly Code with Technique A

[1] Though a *cache line* is sometimes referred to as a *block*, in this paper the term *block* indicates a *basic block* in the compiled program.

**Technique B**

The instructions inserted into the program for Technique B are identical to the instructions inserted for Technique A. The trace routine being invoked, however, interfaces with the cache simulator differently. If no context switches can occur during the execution of the block, the cache simulator is invoked only once with the address of the first instruction within the basic block and the size of entire block passed as arguments. Thus, the cache simulator is treating the entire block as one large instruction. The actual number of references to the cache associated with the block, which is determined statically, may be greater than the number of instructions within the block. This situation occurs when one or more instructions span more than one cache line. The number of references, hits and misses, caused by invoking the simulator for the entire block as one large instruction is equal to the number of cache lines being referenced. The remaining references associated with the block due to spatial locality will always be cache hits and need not be simulated. Thus, the counter for the number of cache hits is incremented by the number of remaining references after the call to the simulator. If it is determined that a periodic context switch may occur during the execution of the block, then the cache simulator will be invoked for each individual instruction within the block as in Technique A.

**Technique C**

Technique C is similar to Technique B except for one difference. Instead of invoking the simulator once for an entire basic block, the technique invokes the simulator once for each sequence of executed blocks that are physically contiguous. The basic block number of the beginning of a sequence of blocks, also described as a superblock [EKK90], is saved. A call instruction to the trace routine is inserted before any unconditional jump, call, or return instructions. Handling conditional transfers of control is a little more complicated. The trace routine should only be invoked when the conditional branch is taken. However, the conditional branch target block could be in the middle of a different sequence of contiguous blocks. Therefore, the target of the conditional jump is replaced with the target of a newly created label. Assembly code is added at the end of the function that contains the new label, the call to the trace routine, the reset of the beginning block of a new sequence, and an unconditional jump back to the conditional jump's original target. An example of assembly code before and after modifications are made for Technique C is shown in Figure 2.

**Technique D**

The remaining techniques use cache configuration information to reduce the number of references to be processed by the cache simulator. The compiler reads in information that indicates the line size and number of sets of a direct-mapped cache. The compiled program can be used to simulate caches with different characteristics including

```
  before
----------
 jbsr  _foo          /* call */
 ...
 jne   L74           /* branch not equal */
 ...
 jeq   L78           /* branch equal */
 ...
 jra   L67           /* unconditional jump */


      after
------------------
 jbsr  _foo
 movl  #15,_startblk  /* follows a call inst */
 ...
 jne   LN10           /* L74 was replaced */
 ...
 jeq   LN11           /* L78 was replaced */
 ...
 pea   #17            /* push last block in seq */
 jbsr  _traceblknum   /* call trace routine */
 addql #4,a7          /* adjust stack pointer */
 movl  #32,_startblk  /* reset start block */
 jra   L67
 .
 .
 .
LN10:
 pea   #15            /* push last block in seq */
 jbsr  _traceblknum   /* call trace routine */
 addql #4,a7          /* adjust stack pointer */
 movl  #22,_startblk  /* reset start block */
 jra   L74            /* jump to orig label */
LN11:
 pea   #16            /* push last block in seq */
 jbsr  _traceblknum   /* call trace routine */
 addql #4,a7          /* adjust stack pointer */
 movl  #26,_startblk  /* reset start block */
 jra   L78            /* jump to orig label */
```

Figure 2: Assembly Code with Technique C

greater set associativity. As in Puzak's trace stripping method [Puz85], if the number of sets is not decreased and the line size remains the same, then the program need not be recompiled.

An array, where an element is indexed by a unique block number, is used to indicate if each basic block in the program is currently within the cache. The compiler determines if each loop in a function, proceeding from the outermost loop first, can fit in the cache and does not contain any calls to other functions that are being measured. The same cache simulator used to analyze the cache performance during the execution of a measured program was linked with the compiler so it could be invoked to check if a loop will fit in the cache. If the loop does fit, then instructions are inserted in the preheader block of the loop to clear the array elements associated with the blocks in the loop. Note, that if a preheader block is not available, then one will be created. For each block within the loop, instructions are inserted to determine if the block currently resides in the cache by checking the array element associated with the block. If the array element is currently set and a simulated context switch cannot occur while the instructions in the block are executed, then the number of cache hits and context switch information is adjusted.[2] It is assumed that the inserted instructions to perform these two checks are much less expensive than processing the reference by the cache simulator. If a context switch does occur, then the array elements within the loop are cleared. Thus, when a loop that fits in the cache is entered, except when a context switch occurs, the cache simulator is invoked at most once for each block in the loop. Technique C is used for code outside of loops or in loops that do not fit in the cache. For the example in Figure 3, which contains a loop with a single basic block, simulated context switches occur every 10,000 units of work. The context interval can be changed, but would require recompiling the program being measured.

The cache configuration information read by the compiler is used to determine if the cache lines that are associated with a basic block are currently in the cache. If these cache lines are resident and each cache line was the last line to be referenced within its set, then there is typically no information that need be updated about the state of the cache in the simulator. The replacement algorithm for determining which cache line to replace within the set can be usage or non-usage based [Smi82]. Non-usaged based algorithms, such as first-in-first-out (FIFO) or random, are not affected by the reuse of a line. The most common usage-based replacement algorithm for set-associative caches is least-recently-used (LRU) [HeP90]. If the current cache line being referenced is also the last line that was referenced within the set, then LRU information need not be updated. Therefore, a program containing a loop that fits in a direct-mapped cache and processed using Technique D would not require recompilation when the number of sets or associativity is increased.

Technique D, as described so far, attempts to avoid calls to the cache simulator for blocks that will be in the cache already due to temporal locality. There are also situations when blocks will already be in the cache due to spatial locality. When a basic block in a loop that fits in the cache is totally contained in the last cache line referenced by the predecessor block and/or the first cache line of the successor block, then instructions are inserted to check the block

[2] Note that adjusting this information does result in a stronger coupling between the trace generation and trace analysis tasks.

```
 clrl  _marker+952       /* clear marker[238] */
 movl  #238,_lowmarker   /* first block cleared on switch */
 movl  #238,_highmarker  /* last block cleared on switch */
L405:
 cmpl  #0,_marker+952    /* check if block 238 is in cache */
 jeq   LN191             /* if not then invoke simulator */
 cmpl  #9992,_c_switch   /* check if switch in 8 cycles */
 jge   LN191             /* if so then invoke simulator */
 addql #8,_c_switch      /* adjust switch information */
 addql #8,_c_hits        /* adjust number of cache hits */
LN192:
 ...
 jne   L405
 movl  #-1,_highmarker   /* out of loop so nothing to clear */
 .
 .
 .
LN191:
 movl  #1,_marker+952    /* block 238 is now in cache */
 movl  #238,_startblk    /* first block in sequence */
 pea   #238              /* last block in sequence */
 jbsr  _traceblknum      /* call trace routine */
 addql #4,a7             /* adjust stack pointer */
 jra   LN192             /* execute insts in block 238 */
```

Figure 3: Assembly Code with Technique D

markers of the predecessor and/or successor blocks. If set, then the call to the cache simulator is avoided and the context switch information and cache hit counter are adjusted.

**Technique E**

Technique E is similar to Technique D except that inter-procedural cache analysis is performed. Initially, a call graph of the functions being measured is constructed using the information provided from the first compilation pass. Then, it is determined if each subtree of the graph, a function and the routines that can be invoked from that function, can fit in the cache at the same time. If an entire function and the routines that can be invoked will fit in the cache, then Technique D, except for clearing the block markers, is used throughout the function. In a function that does not fit in cache, instructions are inserted preceding calls to a function being measured that does fit in the cache to clear the markers associated with that function. Loops with calls to functions that are being measured may also be candidates for avoiding references to be processed by the cache simulator. If the entire loop and the functions that can be invoked from the loop can fit in cache at the same time, then the block markers associated with the blocks in the loop and with each of the functions that can be invoked are cleared in the preheader block of the loop. Inter-procedural cache analysis can still be used when call graphs are cyclic (i.e. recursive). Technique E (and Techniques F and G), however, cannot be used with indirect calls since the function being invoked is not known at compile-time and the call graph cannot be accurately constructed. The example given in Figure 3 for Technique D showed a loop with a single block that contained no calls. Figure 4 shows the same loop except with one call to a routine being measured. The two loop blocks and the three basic blocks in the routine fit in the cache at the same time.

```
 clrl  _marker+952       /* clear marker[238] */
 clrl  _marker+956       /* clear marker[239] */
 clrl  _marker+1056      /* clear marker[264] */
 clrl  _marker+1060      /* clear marker[265] */
 clrl  _marker+1064      /* clear marker[266] */
 movl  #238,_lowmarker   /* first block cleared on switch */
 movl  #266,_highmarker  /* last block cleared on switch */
L405:
 cmpl  #0,_marker+952    /* check if block 238 is in cache */
 jeq   LN191             /* if not then invoke simulator */
 cmpl  #9995,_c_switch   /* check if switch in 5 cycles */
 jge   LN191             /* if so then invoke simulator */
 addql #5,_c_switch      /* adjust context switch info */
 addql #5,_c_hits        /* adjust number of cache hits */
LN192:
 ...
 jbsr  _foo
 cmpl  #0,_marker+956    /* check if block 239 is in cache */
 jeq   LN193             /* if not then invoke simulator */
 cmpl  #9997,_c_switch   /* check if switch in 3 cycles */
 jge   LN193             /* if so then invoke simulator */
 addql #3,_c_switch      /* adjust context switch info */
 addql #3,_c_hits        /* adjust number of cache hits */
LN194:
 ...
 jne   L405
 movl  #-1,_highmarker   /* out of loop so nothing to clear */
```

Figure 4: Assembly Code with Technique E

**Technique F**

To be able to avoid address references being processed by the cache simulator, Technique E required that all the blocks in a loop and the blocks in the routines that can be invoked from the loop fit in the cache at the same time. Technique F relaxes this requirement. All of the blocks in the loop itself still have to fit in the cache at the same time. Also, each individual function invoked directly from the loop and the set of routines the function could in turn invoke have to fit in the cache at the same time. Otherwise there can be conflicts between blocks. For instance, a basic block in a routine invoked from the loop could conflict with a basic block in the loop or a block from a different routine invoked from the same loop. A heuristic was applied that requires that at least one half of the blocks in the loop and the routines invoked cannot conflict. If the heuristic is not satisfied then Technique C is used for the blocks in the loop. Otherwise, if a block in the loop conflicts with blocks in routines invoked from the loop, then instructions are inserted to clear the markers associated with the conflicting blocks in the invoked routines when the loop block is executed. Likewise, when a block in an invoked function conflicts with a loop block or a different routine that is invoked from the same loop, then instructions are inserted to clear the conflicting blocks before the call instruction to the invoked function. Figure 5 shows a loop containing a block that conflicts with a basic block in a routine invoked from the loop.

```
L85:
 ...
 cmpl #0,_marker+2120   /* check if block 530 is in cache */
 jeq  LN141             /* if not then invoke simulator */
 cmpl #9981,_c_switch   /* check if switch in 19 cycles */
 jge  LN141             /* if so then invoke simulator */
 addl #19,_c_switch     /* adjust context switch info */
 addl #19,_c_hits       /* adjust number of cache hits */
LN142:
 clrl _marker+7344      /* clear block 1836 in foo */
 ...
 clrl _marker+2120      /* clear block 530 in loop */
 jbsr _foo
 ...
 jne  L85
```

Figure 5: Assembly Code with Technique F

**Technique G**

Techniques D-F attempt to find basic blocks that are already resident in the cache because of temporal locality due to loops. There is, however, another situation when temporal locality can result in blocks being resident. This situation occurs when a routine is invoked from more than one location and some blocks in the routine have not been replaced when the second call occurs. As stated previously, the block markers of a function that do fit in the cache are cleared before the call instruction to that function in a routine that does not fit in cache. At the point when a call to such a function is encountered in Technique G, the compiler attempts to determine if any block markers in the function have already been cleared and the lines associated with those blocks have not been replaced. Any block that is determined to be resident, and thus its associated marker is already cleared, need not have its marker cleared again. The example in Figure 6 has two calls to the same function that fits in the cache. In this example, the instructions following the first call reside in the same line as only one of the blocks in the function being invoked.

```
clrl _marker+528        /* clear marker[132] */
clrl _marker+532        /* clear marker[133] */
clrl _marker+536        /* clear marker[134] */
clrl _marker+540        /* clear marker[135] */
movl #132,_lowmarker    /* first block cleared on switch */
movl #135,_highmarker   /* last block cleared on switch */
jbsr _foo
...                     /* insts conflicting with block 132 */
clrl _marker+528        /* clear marker[132] */
jbsr _foo
movl #-1,_highmarker    /* out of func so nothing to clear */
```

Figure 6: Assembly Code with Technique G

## RESULTS

The set of test programs used in this experiment and their associated code size are shown in Table I. The code size for each program does not include the routines from the run-time library since their source code was not available. The techniques discussed in this paper could be used to process assembly or object files. Unfortunately, this would complicate implementation of the techniques since portability would be decreased and the control-flow and data-flow information already available in a compiler would have to be recalculated. A C compiler for the Motorola 68020/68881 was implemented within the *ease* environment [DaW90], which consists of a compiler generation system called *vpo* [BeD88] and measurement tools. The compiler was modified to implement each of the seven techniques described in the previous section. Cache performance measurements were obtained for each program within the test set using each of the techniques. The measurements obtained for each specific program were not affected by the technique used. Identical results occurred, the exact number of hits and misses, despite periodic simulated context switches and each program requiring at least one million

| Name | Description or Emphasis | Size in Bytes |
|---|---|---|
| compact | Huffman Coding Compression | 4322 |
| cpp | C Preprocessor | 12678 |
| diff | Differences between Files | 9166 |
| lex | Lexical Analyzer Generator | 26318 |
| sed | Stream Editor | 13946 |
| sort | Sort or Merge Files | 5500 |
| tbl | Table Formatter | 24592 |
| yacc | Yet Another Compiler-Compiler | 22392 |

Table I: Test Programs

cache references.

Periodic context switches were simulated by invalidating the entire cache every 10,000 units of work. A cache hit was assumed to require one work unit while a cache miss was assumed to require ten. The context switch interval and estimated time units required for a hit versus a miss are the same as those used in Smith's cache studies [Smi82]. Though the experiments in this paper simulated context switching based on estimated cache work to check that identical measurements were obtained with the different techniques, other methods to determine context switch points could also be used.

Tables II and III show the number of times that the cache simulator was invoked for each program using each of the techniques for a 1K byte direct-mapped cache with a 16 byte line size. The hit ratio is given to indicate the percentage of references that are candidates for not being processed by the cache simulator. Note that the number of calls to the cache simulator using Technique A is the same as the number of instructions that were executed.[3] The results using Technique B indicate that there were on average 2.77 instructions per basic block being executed. There was also on average 5.29 contiguous instructions being processed by the cache simulator using Technique C. Technique D, which requires no inter-procedural analysis, resulted in a substantial improvement over Technique C. This indicates that a large percentage of instructions executed in programs occur in loops with no calls. Techniques E, F, and G appear to be more closely affected by the hit ratio. The results indicate that Technique F has a slight improvement over Technique E. Technique G, however, rarely resulted in fewer references being processed as compared to Technique F. It is interesting to note that using Techniques E, F, and G can occasionally result in more address references being processed by the cache simulator. This situation can occur when the loops that do fit in cache are also in the functions that fit in cache. Since each block in these functions are processed individually, then the blocks outside the loops in these functions would require more references to be processed by the cache simulator since the method used in Technique C is not applied.

| Program | Hit Ratio | Instructions Executed |
|---|---|---|
| compact | 95.25% | 4,699,295 |
| cpp | 93.78% | 1,322,671 |
| diff | 99.54% | 3,425,264 |
| lex | 99.48% | 36,844,880 |
| sed | 96.49% | 1,643,093 |
| sort | 96.91% | 1,778,463 |
| tbl | 86.59% | 2,715,097 |
| yacc | 98.63% | 23,960,045 |
| average | 95.83% | 9,548,601 |

Table II: Measurements with a 1K Byte Cache

| Program | Relative to Technique A | | | | | |
|---|---|---|---|---|---|---|
| | B | C | D | E | F | G |
| compact | 26.51% | 13.21% | 10.92% | 8.03% | 8.03% | 8.03% |
| cpp | 26.53% | 15.75% | 11.07% | 11.01% | 11.02% | 11.02% |
| diff | 29.83% | 15.07% | 3.22% | 1.22% | 0.77% | 0.77% |
| lex | 50.55% | 19.47% | 2.32% | 1.89% | 0.94% | 0.94% |
| sed | 45.86% | 26.84% | 6.99% | 6.92% | 5.32% | 5.32% |
| sort | 34.36% | 18.49% | 12.24% | 12.59% | 12.59% | 12.59% |
| tbl | 30.70% | 19.65% | 17.28% | 16.78% | 16.36% | 16.24% |
| yacc | 44.58% | 22.89% | 5.17% | 4.52% | 3.51% | 3.51% |
| average | 36.12% | 18.92% | 8.65% | 7.87% | 7.32% | 7.30% |

Table III: Calls to Cache Simulator with a 1K Byte Cache

Table IV shows the number of times that the cache simulator was invoked with cache sizes ranging from 2K to 16K bytes. Increasing the cache size did not vary the number of references processed by the cache simulator using Technique A since the number of instructions that were executed remained the same. Also, the number of references processed using Techniques B, C, and D varied only slightly as the cache size was increased.[4] Therefore, only the hit ratio and results from Techniques E, F, and G were presented in Table IV. Slightly varying number of references processed when the cache size was increased for each program using Technique D indicates that loops with no calls in the test set always fit in a 1K byte cache. Unlike Techniques A-D, Techniques E-G improved as the cache size and hit ratios increased. For a program that executes a very large number of instructions, more time may be saved

[3] The number of executed instructions is slightly less than the total cache references simulated since a Motorola 68020/68881 instruction may span two cache lines.

[4] When a context switch could occur in a basic block, the cache simulator processes each instruction within the block individually. Since changing the cache size typically resulted in context switches occurring in basic blocks with a different number of instructions, there was a slight variation in the number of times that the cache simulator was invoked.

| Cache Size | Program | Hit Ratio | Relative to Technique A | | |
|---|---|---|---|---|---|
| | | | E | F | G |
| 2K bytes | compact | 96.04% | 8.03% | 8.03% | 8.03% |
| | cpp | 96.52% | 11.03% | 11.03% | 11.02% |
| | diff | 99.68% | 1.22% | 0.77% | 0.77% |
| | lex | 99.49% | 0.94% | 0.94% | 0.94% |
| | sed | 98.23% | 6.92% | 3.93% | 3.93% |
| | sort | 99.83% | 9.91% | 5.41% | 5.41% |
| | tbl | 93.94% | 13.40% | 13.42% | 13.17% |
| | yacc | 99.28% | 2.88% | 2.89% | 2.89% |
| | average | 97.88% | 6.79% | 5.80% | 5.77% |
| 4K bytes | compact | 97.63% | 8.03% | 2.46% | 2.46% |
| | cpp | 97.72% | 11.23% | 11.07% | 11.07% |
| | diff | 99.75% | 0.28% | 0.28% | 0.28% |
| | lex | 99.59% | 0.94% | 0.94% | 0.94% |
| | sed | 98.23% | 3.93% | 3.93% | 3.93% |
| | sort | 99.83% | 0.22% | 0.23% | 0.23% |
| | tbl | 95.69% | 13.38% | 13.41% | 13.16% |
| | yacc | 99.45% | 2.86% | 2.88% | 2.88% |
| | average | 98.49% | 5.11% | 4.40% | 4.37% |
| 8K bytes | compact | 99.26% | 0.72% | 0.72% | 0.72% |
| | cpp | 98.06% | 10.90% | 10.30% | 10.30% |
| | diff | 99.75% | 0.28% | 0.28% | 0.28% |
| | lex | 99.60% | 0.92% | 0.92% | 0.92% |
| | sed | 99.11% | 0.82% | 0.82% | 0.82% |
| | sort | 99.82% | 0.22% | 0.22% | 0.22% |
| | tbl | 96.76% | 13.41% | 13.44% | 13.19% |
| | yacc | 99.48% | 0.97% | 0.99% | 0.99% |
| | average | 98.98% | 3.53% | 3.46% | 3.43% |
| 16K bytes | compact | 99.26% | 0.72% | 0.72% | 0.72% |
| | cpp | 98.39% | 1.36% | 1.36% | 1.36% |
| | diff | 99.75% | 0.28% | 0.28% | 0.28% |
| | lex | 99.60% | 0.44% | 0.41% | 0.41% |
| | sed | 99.11% | 0.82% | 0.82% | 0.82% |
| | sort | 99.83% | 0.22% | 0.22% | 0.22% |
| | tbl | 97.89% | 10.78% | 7.05% | 6.95% |
| | yacc | 99.48% | 0.63% | 0.63% | 0.63% |
| | average | 99.16% | 1.91% | 1.44% | 1.42% |

Table IV: Calls to Cache Simulator with Larger Cache Sizes

by recompiling the program when the number of sets in the cache configuration being measured is increased.

Table V shows the execution time required using a 1K byte direct-mapped cache with a 16 byte line size relative to execution without tracing for each program.[5] Note that execution times of the programs being measured include both the time required for generating the trace and analyzing the references with the cache simulator. The ratio to execution time without tracing for the different programs with each technique varied. For instance, the ratio for Technique A for the `lex` program was over 3.8 times as great as the ratio for `tbl`. The ratio to execution time without tracing is affected by a number of factors which include the average execution time required for the non-tracing instructions executed, the average number of instructions in executed basic blocks, and the percentage of time spent in the library routines which were not measured.[6]

| Program | Ratio to Execution Time without Tracing | | | | | | |
|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G |
| compact | 124.57 | 51.35 | 40.23 | 33.22 | 25.70 | 25.74 | 25.80 |
| cpp | 113.05 | 47.83 | 40.24 | 31.55 | 31.62 | 31.62 | 31.67 |
| diff | 83.04 | 35.64 | 26.51 | 8.31 | 5.20 | 4.50 | 4.51 |
| lex | 188.68 | 118.70 | 73.37 | 15.13 | 13.68 | 10.69 | 10.69 |
| sed | 143.18 | 87.62 | 72.85 | 24.94 | 25.35 | 22.05 | 22.05 |
| sort | 161.03 | 83.41 | 64.75 | 46.39 | 47.64 | 47.72 | 47.64 |
| tbl | 49.65 | 25.22 | 22.88 | 21.01 | 21.10 | 20.82 | 20.35 |
| yacc | 124.56 | 75.12 | 54.82 | 17.29 | 16.46 | 14.20 | 14.09 |
| average | 123.47 | 65.61 | 49.46 | 24.73 | 23.34 | 22.17 | 22.10 |

Table V: Execution Time with a 1K Byte Cache

Table VI shows the execution time overhead with cache sizes ranging from 2K to 16K bytes. In general, the execution times decrease as the number of references processed by the cache simulator decrease. Thus, as the cache size increases, the execution times for programs using Techniques E, F, and G decrease. The execution times for Techniques A-D varied only slightly since the number of references processed by the cache simulator only changed slightly with different cache sizes simulated. Therefore, only the execution time ratios for Techniques E, F, and G are presented in Table VI.

[5] All execution times reported in this paper were obtained by determining the average of ten executions of each instance of a program.

[6] Smaller ratios to execution times without tracing were reported for a method similar to Technique B in the *trapeds* system [StF89] This discrepancy was probably due to their choice to simulate more floating-point intensive programs, to not introduce or check for pending context switches, and the use of a cache simulator tuned for specific cache configurations.

| Cache Size | Program | Ratio to Execution Time without Tracing | | |
|---|---|---|---|---|
| | | E | F | G |
| 2K bytes | compact | 25.58 | 25.63 | 25.57 |
| | cpp | 29.58 | 29.60 | 29.58 |
| | diff | 5.12 | 4.41 | 4.44 |
| | lex | 10.83 | 10.83 | 10.75 |
| | sed | 16.90 | 16.90 | 16.91 |
| | sort | 37.72 | 24.30 | 24.33 |
| | tbl | 15.96 | 16.32 | 16.01 |
| | yacc | 12.48 | 12.53 | 12.52 |
| | average | 19.27 | 17.57 | 17.51 |
| 4K bytes | compact | 24.47 | 11.56 | 11.51 |
| | cpp | 29.65 | 29.17 | 29.45 |
| | diff | 3.76 | 3.78 | 3.81 |
| | lex | 11.02 | 10.93 | 10.98 |
| | sed | 16.98 | 17.19 | 17.14 |
| | sort | 6.87 | 8.64 | 8.55 |
| | tbl | 15.42 | 15.42 | 15.05 |
| | yacc | 12.52 | 12.83 | 12.68 |
| | average | 15.09 | 13.69 | 13.65 |
| 8K bytes | compact | 6.67 | 6.67 | 6.67 |
| | cpp | 29.14 | 27.93 | 27.95 |
| | diff | 4.08 | 4.08 | 4.08 |
| | lex | 11.38 | 11.47 | 11.38 |
| | sed | 9.49 | 9.59 | 9.61 |
| | sort | 7.63 | 7.64 | 7.65 |
| | tbl | 15.22 | 15.21 | 15.29 |
| | yacc | 9.08 | 9.11 | 9.12 |
| | average | 11.59 | 11.46 | 11.47 |
| 16K bytes | compact | 7.44 | 7.44 | 7.45 |
| | cpp | 8.99 | 9.02 | 8.97 |
| | diff | 4.73 | 4.73 | 4.71 |
| | lex | 11.91 | 11.74 | 11.74 |
| | sed | 10.95 | 10.97 | 10.97 |
| | sort | 8.58 | 8.60 | 8.58 |
| | tbl | 13.29 | 10.14 | 10.02 |
| | yacc | 8.96 | 8.97 | 8.92 |
| | average | 9.36 | 8.95 | 8.92 |

Table VI: Execution Time with Larger Cache Sizes

## FUTURE IMPROVEMENTS

The tracing overhead is dependent on the performance of the cache simulator. Less tracing overhead would be required if cache simulators were used that were tuned for a particular cache configuration. This scheme, however, would be less flexible since a program would have to be relinked each time the cache configuration was changed.

The introduction of periodic context switches also increased the execution times. Figure 3 shows an example of instructions inserted to keep track of the intervals between context switches for Techniques D-G. The absence of context switches would also allow other techniques to be used to further reduce the tracing overhead. For instance, the number of cache hits and misses for the blocks in a loop that fits in cache could be adjusted after the loop exits, rather than each time the block was executed. The marker of a block within the loop would be incremented each time the block is entered. Only the first reference to each block would require simulation. The order of simulating the first references to the different blocks within the loop would not affect the total number of hits and misses.

## CONCLUSIONS

The techniques presented in this paper have been shown to significantly reduce the time required for instruction cache performance evaluations as compared to more traditional approaches. This improvement occurred despite no special requirements to implement the techniques[7] and without any loss of accuracy. Technique D is particularly attractive since with no interprocedural analysis required it is simple to implement and still results in a significant improvement. Though only the number of instruction references to be processed can be reduced, the techniques can also be used when evaluating split instruction and data caches. There still should be a measurable improvement in this situation since typically the majority of address references being processed are instructions [BKW90]. The effective evaluation of large second-level caches may require billions of references to be traced. When positioned behind a split first-level cache, the techniques presented in this paper would be very useful.

## REFERENCES

[ASH86] A. Agarwal, R. L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th Annual Symposium on Computer Architecture*, pp. 119-127 (June 1986).

[BeD88] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

[BKW90] A. Borg, R. E. Kessler, and D. W. Wall, "Generation and Analysis of Very Long Address Traces," *Proceedings of the 17th Annual*

[7] Some approaches have dedicated a set of registers to be used exclusively for tracing and/or require special operating system support [BKW90, EKK90, ASH86, Wie82].

*International Symposium on Computer Architecture*, pp. 270-279 (May 1990).

[DaW90] J. W. Davidson and D. B. Whalley, "Ease: An Environment for Architecture Study and Experimentation," *Proceedings SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, pp. 259-260 (May 1990).

[EKK90] S. J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy, "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor," *Proceedings SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, pp. 37-47 (May 1990).

[HeP90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA (1990).

[HiS89] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers* **38**(12) pp. 1612-1630 (December 1989).

[HLT87] M. Huguet, T. Lang, and Y. Tamir, "A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements," *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pp. 14-25 (June 1987).

[HwC89] W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proceedings of the 16th Annual Symposium on Computer Architecture*, pp. 242-250 (May 1989).

[LPI88] S. Laha, J. H. Patel, and R. K. Iyer, "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," *IEEE Transactions on Computers* **37**(11) pp. 1325-1336 (November 1988).

[McF89] S. McFarling, "Program Optimization for Instruction Caches," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183-191 (April 1989).

[MiF88] C. L. Mitchell and M. J. Flynn, "A Workbench for Computer Architects," *IEEE Design & Test of Computers* **5**(1) pp. 19-29 (February 1988).

[PeH90] K. Pettis and R. Hansen, "Profile Guided Code Positioning," *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 16-27 (June 1990).

[PeS77] B. L. Peuto and L. J. Shustek, "An Instruction Timing Model of CPU Performance," *Proceedings of the 4th Annual Symposium on Computer Architecture*, pp. 165-178 (March 1977).

[Puz85] T. R. Puzak, *Analysis of Cache Replacement Algorithms*, PhD Dissertation, University of Massachusetts, Amherst, MA (February 1985).

[Smi77] A. J. Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data," *IEEE Transactions on Software Engineering* **3**(1) pp. 94-101 (January 1977).

[Smi82] A. J. Smith, "Cache Memories," *Computing Surveys* **14**(3) pp. 473-530 (September 1982).

[StF89] C. Stunkel and W. Fuchs, "TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation," *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pp. 70-78 (May 1989).

[WaB90] W. Wang and J. Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis," *Proceedings SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, pp. 27-36 (May 1990).

[Wie82] C. A. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 177-184 (March, 1982).