April 1991

UILU-ENG-91-2221
CRHC-91-14

Center for Reliable and High-Performance Computing

THE SUSCEPTIBILITY OF PROGRAMS TO CONTEXT SWITCHING

Wen-mei W. Hwu
Thomas M. Conte

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY none	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE none	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CRHC-91-14 UILU-ENG-91-2221	5. MONITORING ORGANIZATION REPORT NUMBER(S) none
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA, NCR, AMD, ONR, NSF
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA: NASA Langley Research Center Hampton, VA 23665 NSF: 1800 G. Street, Washington, DC 20552
8a. NAME OF FUNDING/SPONSORING ORGANIZATION same as 7a. 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NSF: MIP-8809478 NASA: NAG 1-613 ONR: N00014-88-K-0656
8c. ADDRESS (City, State, and ZIP Code) same as 7b.	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

The Susceptibility of Programs to Context Switching

12. PERSONAL AUTHOR(S)

Hwu, Wen-mei W., and Conte, Thomas M.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1991 April	31

16. SUPPLEMENTARY NOTATION

none

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

memory system performance, single-pass method, context swtiching

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Modern memory systems are composed of several levels of caching. Design of these levels is largely an empirical practice. One highly-effective empirical method is the single-pass method wherein all caches in a broad design space are evaluated in one pass over the trace. Multiprogramming degrades memory system performance since (process) context switching reduces the effectiveness of cache memories. Few single-pass methods exist which account for multiprogramming effects. This paper uses a general model of single-pass algorithms, called the recurrence/conflict model, and extensions to the model for recording the effects due to both voluntary and involuntary context switching. The method presented in this paper accurately records a program's susceptibility to context switching for all cache dimensions and all context switching intensities in a single pass. System load is parameterized using context switch intensity and the fraction of cache flushing. Several members of the SPEC benchmark set are used to comment on program susceptibility to context switching. The accuracy of the method is shown to be quite good by comparing it with two more-restrictive test methods. The results also agree well with multiprogramming effects reported by others.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

7b. NCR: Personal Computer Div.-Clemson, 1150 Anderson Dr., Liberty, SC 29657
Advanced Micro Devices: 5900 East Ben White Blvd., Austin, TX 78741
ONR: Department of the Navy, Office of Naval Research (Code 1114SE),
800 N. Quincy St., Arlington, VA 22217-5000

The Susceptibility of Programs to Context Switching

Wen-mei W. Hwu Thomas M. Conte

Center for Reliable and High-Performance Computing
University of Illinois
1101 West Springfield Avenue
Urbana, Illinois 61801
hwu@crhc.uiuc.edu

March 21, 1991

Abstract

Modern memory systems are composed of several levels of caching. Design of these levels is largely an empirical practice. One highly-effective empirical method is the single-pass method wherein all caches in a broad design space are evaluated in one pass over the trace. Multiprogramming degrades memory system performance since (process) context switching reduces the effectiveness of cache memories. Few single-pass methods exist which account for multiprogramming effects. This paper uses a general model of single-pass algorithms, called the recurrence/conflict model, and extensions to the model for recording the effects due to both voluntary and involuntary context switching. The method presented in this paper accurately records a program's susceptibility to context switching for all cache dimensions and all context switching intensities in a single pass. System load is parameterized using context switch intensity and the fraction of cache flushing. Several members of the SPEC benchmark set are used to comment on program susceptibility to context switching. The accuracy of the method is shown to be quite good by comparing it with two more-restrictive test methods. The results also agree well with multiprogramming effects reported by others.

The Susceptibility of Programs to Context Switching

1 Introduction

Multiple levels of caching and buffering have become the norm in memory system design. These systems are typically designed using simulation to determine the performance of a wide range of memory system organizations. The inputs to the simulator are benchmarks that represent nominal system workloads. The designer's job is to choose the most cost-effective organization using the simulation results as a guide. A class of powerful simulation methods, called single-pass stack methods, have become available to memory system designers [1],[2],[3],[4]. With these methods, the memory system performance of thousands of organizations can be determined using a single pass through the memory access trace of the benchmark, whereas traditional multiple-pass methods require one pass per potential memory system design.

Multiprogramming degrades memory system performance since (process) context switching reduces the effectiveness of cache memories. This occurs when cache contents that will be needed after the process returns are purged by the intervening processes. The cache contents that may fall victim to context switching are determined by the process' reference pattern (a program characteristic) and the cache dimension (a system design parameter). The portion of such cache contents that are actually purged by intervening processes are determined by load of the system: the number of ready processes and access patterns of these processes. The method presented in this paper accurately records, for all cache dimensions and all context switching intensities in a single pass, the total amount of cache contents that will be needed after the process returns. This information is defined as the *susceptibility* of the

program to the effect of context switching.

Several other approaches have been taken to measure the effects of context switching [5],[6],[7],[8],[9],[10],[11],[12]. The earliest approaches flushed the cache being simulated at fixed intervals in the trace [5][6]. Shedler and Slutz [7] approached the problem by stochastically merging several memory reference traces. Easton [8] used the average working set size of the memory reference trace to estimate cold-start miss ratios. Haikala [11] simplified Easton's approach by estimated cold-start miss ratios using a Markov chain model. Cold-start miss ratios can be used to approximate the multiprogramming effects. Switching between multiple memory reference traces at a fixed interval was used by Smith [9] to measure multiprogramming effects. Also, hardware measurements of a real multiprogrammed workloads were performed by Clark [10] and, Agarwal, *et al.* [12]. Apart from the approximations of Easton [8] and Haikala [11], no work has been done to extend single-pass methods to model the effects of context switching exactly. Since multiprogramming effects can account for a 4% to 12% degradation in performance [10],[11],[12], this omission in the literature has limited the usefulness of single-pass methods.

One obvious extension to single-pass methods to model context switching effects is to flush the LRU stack periodically. The shortcoming of this approach is that one simulation would have to be performed for each context switching intensity (e.g., time quantum and I/O workload). A more desirable method is to record the context switching effects for all intensities in one pass. This paper introduces a single-pass method for measuring the susceptibility of a program to the effects of context switching for all cache dimensions and all intensities. It is demonstrated that the susceptibility measures can be combined with system load parameters and context switching intensity to yield the performance degradation

in various multiprogramming environments without resimulation. Obtaining memory system performance degradation under many different system loads allows the memory system to be designed with a degree of robustness. It further increases the advantage of single-pass stack methods over multiple-pass methods. To our knowledge, this is the first such study to make the dichotomy between program susceptibility and multiprogramming effects. The measured performance is validated both empirically and by comparing the results to those of other researchers.

2 Recurrences, Conflicts, and Context Switches

The metric used in many memory system studies is the miss ratio. This is the ratio of the number of references that are not satisfied by a cache at a level of the memory system hierarchy over the total number of references. The miss ratio has served as a good metric for memory systems since it is a characteristic of the workload (e.g., the memory trace) yet independent of the access time of the memory elements. Therefore, a given miss ratio can be used to decide whether a potential memory element technology will meet the required access time for the memory system. The recurrence/conflict model of the miss ratio is best illustrated with an example. Consider the trace of Figure 1. The *recurrences* in the trace are accesses E, F, G and H. Without context switching, all the four recurrences would produce a hit in an infinite cache. In the ideal case of an infinite cache in the absence of

Reference	A	B	C	D	E	F	G	H
Address	0	1	2	3	1	2	1	2

Figure 1: An example trace of addresses.

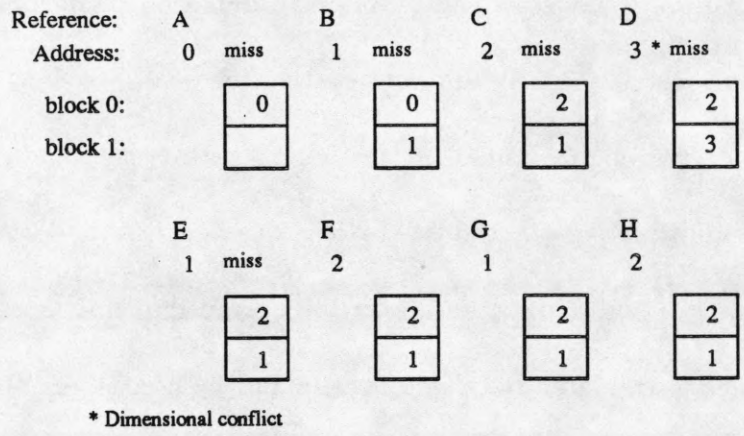

Figure 2: An example two-block direct-mapped cache behavior.

context-switching, the miss ratio may be expressed as,

$$\rho = \frac{N - R}{N}, \tag{1}$$

where R is the total number of recurrences and N is the total number of references. Non-ideal behavior occurs due to *conflicts*, and this paper considers two such types of conflicts: *dimensional conflicts* and *multiprogramming conflicts*. A *dimensional conflict* is defined as an event which converts a recurrence into a miss due to limited cache capacity or mapping inflexibility. For illustration, consider a direct mapped cache composed of two one-byte blocks shown in Figure 2. (Note that in practice, such a small cache would be impractical to build.) A miss occurs for the recurring recurrence E because reference D purges address 1 from the cache due to insufficient cache capacity. Hence, D represents a dimensional conflict for the recurrence E. The other misses, A, B, C and D, occur because these are the first references to addresses $0, 1, 2$ and 3, respectively.

A *multiprogramming conflict* is defined as an event which converts a recurrence into a miss due to a context switch. For example, both G and H are dimension hits of the cache in Figure 1. If a context switch occurs between references F and G which purges addresses

1 and 2 out of the cache, two multiprogramming conflicts will occur, one to reference G and one to reference H. Therefore, the following formula can be used for deriving cache miss ratio, ρ, for a given trace, a given cache dimension and a given pattern of context switching:

$$\rho = \frac{N - (R - C_D - C_M)}{N}, \tag{2}$$

where C_D the total number of dimensional conflicts, and C_M the total number of multiprogramming conflicts. This is a general model and can be extended account for other effects, such as conflicts due to multiprocessor cache coherence [13].

2.1 Reference streams and cache dimensions

The formal abstraction of a benchmark's trace is termed a "reference stream." This is a sequence of address references, $w(k)$, of length N ($0 \leq k < N$). The addresses are addresses in the lowest level of a cache hierarchy, which is assumed to be a linear space (e.g., the virtual space). When they are required, such references will be represented by lower-case Greek letters, such as α, β, γ. The reference stream is assumed to be generated by a single process in a multiprogramming system. A time variable, k, is a measure of the system clock. Also, a reference will be called as a *voluntary context-switch point* if the benchmark relinquished the CPU after the reference (e.g., a system call was performed).

The dimension of a cache is expressed using the notation, (C, B, S), for a cache of size 2^C bytes, with block size 2^B bytes, and 2^S blocks contained in each associativity set. Note that $C \geq B + S$. The notation (C, B, ∞) is an abbreviation for the dimension of a fully-associative cache ($S = C + B$). For example, a cache of dimension $(10, 6, 0)$ is a 1KB direct-mapped cache with a block size of 64 bytes, and a cache of dimension $(21, 10, 11)$

(alternately, $(21, 10, \infty)$) is of size 2MB with 1KB-length blocks and it is fully-associative. A dash is substituted for an entry in the triple to indicate all caches of that dimension. Hence, $(-, 5, 1)$ are all caches with block size 32 bytes and 2-way associativity. Caches are assumed to use LRU replacement and map addresses into sets using bit selection [3].

It is useful to partition the reference stream by setting the block offset portion of all addresses in the stream to zero. This produces a *block reference stream*, $w_B(k)$, is defined such that,

$$w_B(k) = 2^B \left\lfloor \frac{w(k)}{2^B} \right\rfloor .$$

In binary, this is equivalent to setting the least-significant B bits to zero. The number of recurrences is measured for the block reference stream, and denoted $R[B]$. Dimensional conflicts, $C_D[C, B, S]$, are measured for each cache dimension using a single-pass technique [3],[2],[4].

2.2 Types of context switches

Context switching occurs due to two distinct events: (1) a *voluntary context switch*, where the benchmark relinquishes the processor, and, (2) an *involuntary context switch*, where the benchmark's execution is suspended due to external interrupts. Voluntary context switches are a characteristic of the benchmark. They occur at the same place in the execution between different benchmark runs. On the other hand, involuntary context switches are determined by the I/O system behavior (device interrupts), clock frequency (timer interrupts), etc. They do not occur at the same place between runs of the benchmark and are not characteristic of the benchmark. Since involuntary context switches occur at random instances, it is assumed that involuntary context switches can occur with equal probability for each reference in the

reference stream [11]. This probability will be denoted, q, and termed the involuntary *context switching intensity*. As an example, in the VAX 11/780 implementation of 4.3 BSD Unix, the timer interrupt frequency is once every 10 ms [14]. In the absence of other external interrupts, this frequency is equivalent to $q \approx 0.0001$.

Separation of the system's characteristics from the characteristics of the benchmark allows many different systems to be considered without re-simulating the benchmark's behavior. This is the main goal of single-pass techniques in general [2]. Although the occurrence of involuntary context switches is not a characteristic of the benchmark, the benchmark's susceptibility to their occurrence is. This susceptibility can be measured as the expected number of multiprogramming conflicts due to random involuntary context switching. A method to measure this susceptibility is presented below that records the benchmark's susceptibility to all context-switching intensities in a single-pass through the trace. The empirical results discussed in Section 3.4 demonstrate the validity of this single-pass approach.

The working set of a process (benchmark) may have been flushed from the cache before it re-enters the run state after a context switch. Let ξ represent the fraction of the cache's contents flushed between context switches.

The number of processes executed before a process returns from a context switch is a function of the system load and the operating system scheduling policy. Furthermore, the particular cache blocks flushed due to a context switch also depends on the reference patterns of the processes executing on the system. This makes ξ highly dependent on several volatile variables and therefore difficult to measure. Some virtual memory system implementations force a cache flush to eliminate problems with page sharing of writable pages [12]. Also, it has been shown that for small cache sizes, a context switch effectively flushes the cache,

therefore $\xi = 1$ [9]. For larger caches, this provides an upper bound for the effects of context switching. An analytical model for ξ was constructed by Agarwal, *et al.* [15], and the model's required parameters can be obtained using single-pass methods. This calculation of ξ can be used to scale the results of the method of this paper to accurately predict multiprogramming effects. This extension, however, is beyond the scope of this paper. The empirical results are presented with $\xi = 1$.

2.3 The components of multiprogramming conflicts

Multiprogramming conflicts are defined in terms of *potential victims.* A recurring reference that is not removed from a specific cache by a dimensional conflict, yet that may be removed by a context switch is a potential victim. Potential victims are defined as $V_V[C, B, S]$ and $\bar{V}_I[C, B, S, q]$, for all voluntary and involuntary context switches, respectively. $V_V[C, B, S]$ is the total number of potential victims due to voluntary context switching for caches of dimension (C, B, S). On the other hand, $\bar{V}_I[C, B, S, q]$ is the expected number of potential victims due to involuntary context switching of intensity q. The multiprogramming conflicts are expressed in terms of victims as,

$$C_M[C, B, S, q] = \xi \left(V_V[C, B, S] + \bar{V}_I[C, B, S, q] \right). \tag{3}$$

Determining the multiprogramming conflicts involves measuring $V_V[C, B, S]$ and $\bar{V}_I[C, B, S, q]$ from the reference stream. The measurement can be done using the LRU stack of a stack-based cache simulator, as explained below. $C_M[C, B, S, q]$ is then calculated by applying Equation 3 for a value of ξ.

2.4 Least-recently-used (LRU) stack operation

An LRU stack operates as follows: when an address, $w_B(k) = \alpha$, is encountered in the block reference stream, the LRU stack is checked to see if α is present on the stack. If α is not present, it is pushed onto the stack. However, if α is present (e.g, it is a recurring reference),

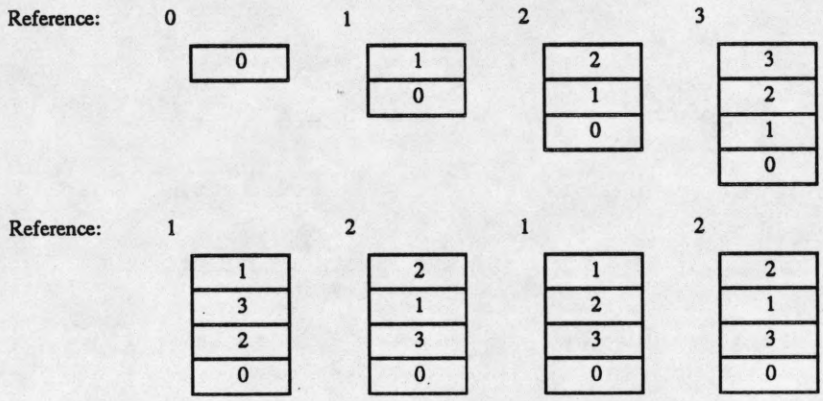

Figure 3: An example of LRU stack operation.

it is removed from the stack, then repushed onto the stack. This is illustrated in Figure 3 for the example reference stream at the beginning of this section (Figure 1). LRU stacks were first introduced by Mattson, *et al.* in [1].

An LRU stack is represented as $S_B(k)$, maintained for a block size B at time k. The ith ordered item of $S_B(k)$ is expressed as, $S_B(k)[i]$. The stack may also be expressed as an ordered list, such that $S_B(k) = \{S_B(k)[0], S_B(k)[1], \ldots, S_B(k)[|S_B(k)|]\}$. The following operations are defined for the stack:

the **push**$(\cdot)$ function,

$$\textbf{push}(S_B(k), \alpha) = \left\{ \alpha,\, S_B(k)[0],\, S_B(k)[1],\, \ldots,\, S_B(k)[|S_B(k)|] \right\},$$

the **where**$(\cdot)$ function,

$$\textbf{where}(S_B(k), \alpha) = i, \qquad \text{if } S_B(k)[i] = \alpha,$$

and, the **repush**$(\cdot)$ function,

$$\textbf{repush}(S_B(k), \alpha) = \Big\{ \alpha,\, S_B(k)[0],\, S_B(k)[1],\, \ldots,\, S_B(k)[\textbf{where}(S_B(k), \alpha) - 1],$$
$$S_B(k)[\textbf{where}(S_B(k), \alpha) + 1],\, \ldots,\, S_B(k)[|S_B(k)|] \Big\}.$$

where$(S_B(k), \alpha)$ and **repush**$(S_B(k), \alpha)$ are undefined when $\alpha \notin S_B(k)$. When $S_B(k)$ and α are understood, it is convenient to define $\Delta =$ **where**$(S_B(k), \alpha)$ Note that **push**$(\cdot)$ and **repush**$(\cdot)$ are defined as side-effect-free functions, rather than procedures. This is to remove dependence on the time variable, k.

For an address $\alpha = w_B(k)$, the least-recently used (LRU) management policy for a stack is shown in Figure 4. In Step 1.1, the references between the top of stack and the recurring reference have been referred to as the set $\{\beta_i \mid \beta_i = S_B(k-1)[i], 0 \leq i \leq \Delta\}$. The LRU

1. **if** $\alpha \in S_B(k-1)$ **then**
1.1 process the intervening references, $\{\beta_i\}$
1.2 $S_B(k) \leftarrow$ **repush**$(S_B(k-1), \alpha)$,
2. **else** $S_B(k) \leftarrow$ **push**$(S_B(k-1), \alpha)$

Figure 4: The least-recently used management policy for a stack, $S_B(k)$ (adapted from Mattson *et al.*).

policy is essentially a definition for calculating $S_B(k)$ from $S_B(k-1)$ and α.

2.5 Multiprogramming extensions to LRU stack operation

The procedure for determining $V_V[C, B, S]$ using an LRU stack operates as follows. When α is processed, if it is not a recurring reference (i.e., the test of Step 1 of Figure 4 fails), then it

cannot be a victim since it cannot produce a hit. However, if α is a voluntary context switch point, it is marked as such when it is pushed on the stack in Step 2 (see Figure 5). Now

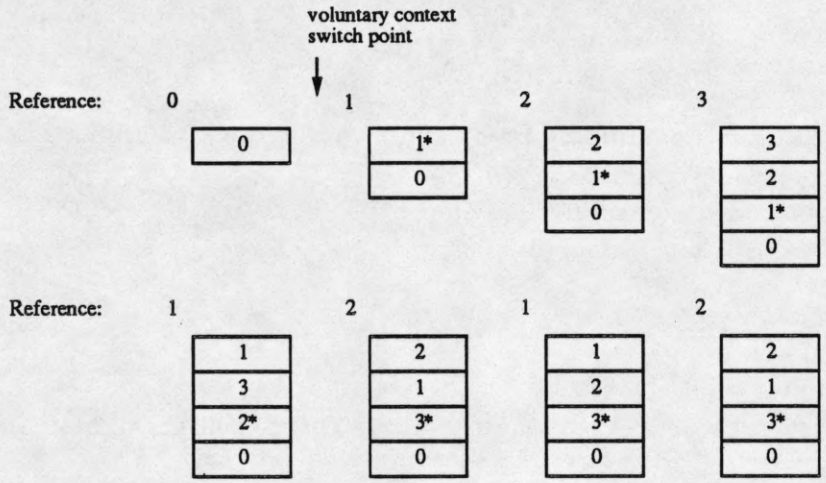

Figure 5: An example for voluntary context switch of the modified LRU stack operation.

assume that α is a recurring reference and therefore already on the stack. If the intervening references on the stack, $\{\beta_i\}$, contain an address marked as a voluntary context switch point, $V_V[C, B, S]$ is incremented for all dimensions in which α does not have a dimensional conflict. This is done since the presence of a voluntary context switch point in $\{\beta_i\}$ implies a voluntary context switch occurred between references to α. The LRU stack contents for the example are illustrated in Figure 5, where a marked stack address is depicted using an asterisk. To insure all subsequent recurring references are subject to a voluntary context switch point, when a marked reference is repushed, the reference immediately above it inherits the context switch point (this occurs for the fourth and fifth reference of Figure 5).

The procedure for determining $\bar{V}_I[C, B, S, q]$ using an LRU stack is somewhat more complicated. Recall that an involuntary context switch may occur between every reference. Let L, the *context switch distance*, be the number of potential involuntary context switch points for the recurring reference α at time k (i.e., $\alpha = w_B(k - L) = w_B(k)$). Let p_L be the

probability that at least one involuntary context switch occurs between times $k - L$ and k. Then,

$$p_L = \sum_{j=1}^{L} \binom{L}{j} q^j (1-q)^{L-j}. \tag{4}$$

Define $n_L[C, B, S]$ to be the number of recurrences not subject to dimensional conflicts that have a context switch distance of L. Therefore,

$$\bar{V}_I[C, B, S, q] = E\Big[n_L[C, B, S]\Big] = \sum_L p_L n_L[C, B, S]. \tag{5}$$

Equation 5 expresses the expected number of potential victims due to involuntary context switching. This equation is more general than the approaches of others because no assumptions must be made concerning the probability of accessing an associativity set in the cache [8],[11]. Equation 5 fits naturally into a stack-based method. The new metric $n_L[C, B, S]$ can be recorded by annotating the references on the stack. As before, if the address α is not on the stack, it cannot cause a miss due to involuntary context switching. When it is pushed onto the stack in Step 2 of Figure 4, a counter of the number of context switch points affecting α is kept, defined as $c_I(\alpha)$. Initially, $c_I(\alpha) = 1$. To see how this operates, assume that α is on the stack. In Step 1.1.2 and 1.1.3.3, one plus the sum of the counters of $\{\beta_i\}$ is used to calculate the involuntary context switch distance, L. Notice that $c_I(\alpha)$ is not part of L. In Step 1.1.5, $n_L[C, B, S]$ is incremented for all caches in which there are no dimensional conflicts. Let $S_B(k-1)[\Delta - 1] = \beta_0$, the address that is directly above α in the stack $S_B(k-1)$. Then, as a bookkeeping step, $c_I(\beta_0)$ is incremented by $c_I(\alpha)$. In this way, all the references deeper in the stack than α in $S_B(k-1)$ will arrive at the correct context switch distance.

The methods for calculating both $V_V[C, B, S]$ and $n_L[C, B, S]$ are presented in Figure 6.

```
1.      if α ∈ S_B(k − 1) then
1.1.1     vol_cs ← false
1.1.2     L ← 1
1.1.3     for i ← 0 to Δ do
1.1.3.1             β_i ← S_B(k − 1)[i]
1.1.3.2             if β_i marked as a voluntary context switch point then
1.1.3.2.1               vol_cs ← true
1.1.3.3             L ← L + c_I(β_i)
1.1.4     for all (C, B, S) without a dimensional conflict do
1.1.5       n_L[C, B, S] ← n_L[C, B, S] + 1
1.1.6       if vol_cs then V_V[C, B, S] ← V_V[C, B, S] + 1
1.2.1     c_I(β_{Δ−1}) ← c_I(β_{Δ−1}) + c_I(α)
1.2.2     c_I(α) ← 1
1.2.3     S_B(k) ← repush(S_B(k − 1), α),
2.      else
2.1       c_I(α) ← 1
2.2       S_B(k) ← push(S_B(k − 1), α)
```

Figure 6: An LRU stack method modified for context switching.

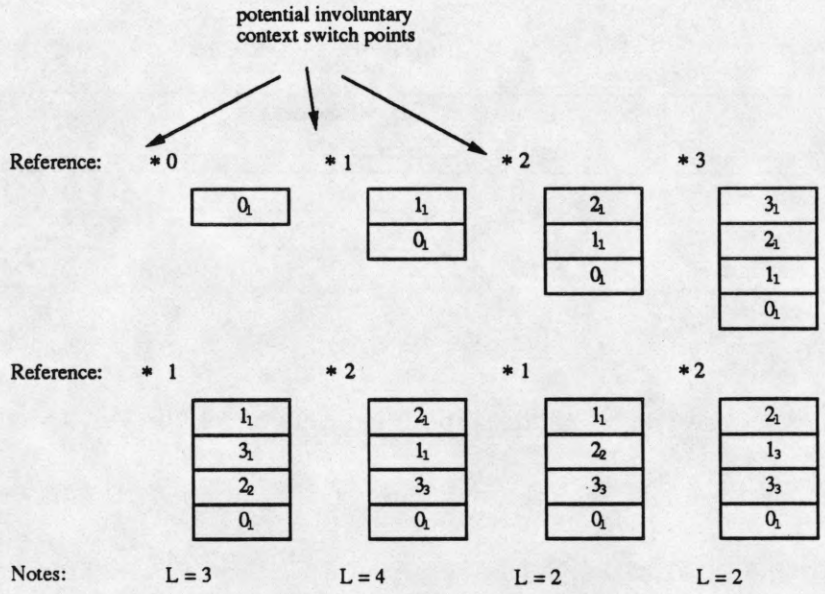

Figure 7: An example for involuntary context switching of the modified LRU stack operation.

An example of the operation of the method is shown in Figure 7. The example also shows the calculated values of L. Notice that since the calculation of $n_L[C, B, S]$ is independent of the context switching intensity distribution assumptions, it is possible to substitute other context switching intensity distributions without altering the method.

3 Empirical Results of Program Susceptibility

The susceptibility of programs to context switching was measured for three members of the SPEC benchmark set, version 1.0 [16], and the results are presented in this section. The benchmarks were: gcc, spice2g6, and espresso. The gcc benchmark is a run of the GNU C compiler (version 1.34) compiling portions of itself; the espresso benchmark is a run of the Espresso PLA minimizer with several PLA's as input; and, the spice2g6 benchmark is version 2G6 of the SPICE circuit simulator written in FORTRAN with a greycode counter circuit as its input. Some benchmark characteristics are presented in Table 1

Table 1: Benchmark characteristics.

Benchmark	Number of references	Fully-associative Cache designs with $\rho \leq 1\%$		
gcc	3.3×10^7	$(16, 4, \infty)$	$(16, 5, \infty)$	$(16, 6, \infty)$
espresso	1.1×10^8	$(15, 4, \infty)$	$(15, 5, \infty)$	$(15, 6, \infty)$
spice	6.2×10^8	$(18, 4, \infty)$	$(17, 5, \infty)$	$(17, 6, \infty)$

The benchmarks were traced using a modified version of the AE tracing tool [17] extended to include system calls and the behavior of all Unix generic library calls [18]. All benchmarks were compiled with GNU C (version 1.37.1) with all compiler optimizations enabled [19]. A FORTRAN-to-C translator [20] was used for the FORTRAN benchmark, spice2g6. This translator operates essentially on the statement-by-statement level, min-

imizing spurious translation effects. Instruction density varies widely across architectures, whereas the data layout of a multiple-architecture compiler like GNU C does not. Therefore, we chose to exclude instruction references and consider only data memory references. The design space used for our simulations is all caches up to 2 gigabytes, with block sizes of 16 bytes, 32 bytes and 64 bytes, and, associativity levels of one-way (direct-mapped), two-way, four-way, and fully associative.

The results that follow are used to comment on the susceptibility of involuntary context switching followed by a discussion of the effects of voluntary context switching. The dimensional conflicts that occur due to different cache sizes are discussed in Section 3.3 to compare their performance degradation with that of context switching. The accuracy of the single-pass method is discussed by comparing the method's results with the results from more limited techniques. Also, a discussion of how the observed context switching behavior compares to other researchers is presented.

3.1 Involuntary context switching susceptibility

It is useful to define $\Delta\rho = C_M/N$ as the susceptibility measure. This is the difference between the uniprogramming and multiprogramming miss ratios. Figure 8, 9 and 10 presents $\Delta\rho$ for the three benchmarks for block sizes 16 bytes, 32 bytes and 64 bytes, respectively. These figures are for caches $(31, -, \infty)$, to eliminate the effects of dimensional conflicts and consider only involuntary context switching. Also, $\xi = 1$ (complete cache flushing).

Two observations are immediately apparent from the figures. The susceptibility to context switching decreases as block size increases from Figure 8 to Figure 10. For example, when $q = 0.001$, $\Delta\rho(16) = 8\%$ whereas $\Delta\rho(64) = 5.2\%$ for spice2g6. The other observation

is that when the intensity is smaller, $\Delta\rho$ approaches zero such that *context switching has little effect for* $q \leq 0.0001$. This value of q corresponds to a context switching intensity of once every 10,000 instructions.

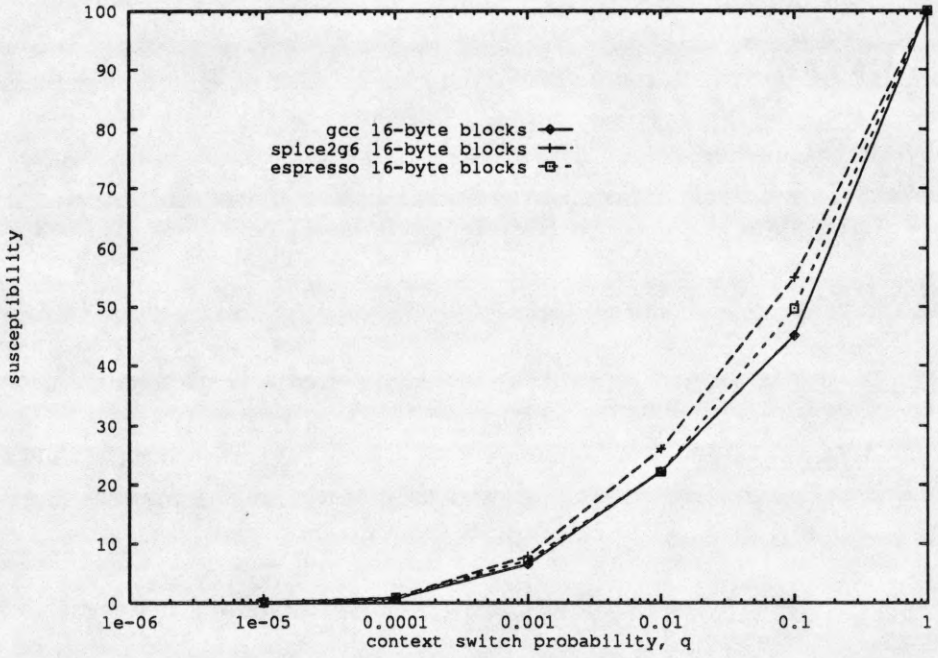

Figure 8: $\Delta\rho$ (involuntary) of gcc, espresso and spice2g6 vs. q for block size 16 bytes

To answer why the susceptibilities of the benchmarks assume the values they do, the distribution of n_L vs. L can be used. These distributions are presented in Figures 11, 12 and 13 for benchmarks gcc, espresso and spice2g6, respectively. The values of L have been divided into several categories of roughly exponentially increasing size (e.g., the second category contains 24 values of L, whereas the third contains 96 values of L, etc.). The final category contains all values of L for $L \geq 512$. Benchmarks with high occurrences of large context switch distances are most susceptible context switches because there is a higher chance of a context switch for large L from Equation 4. In Figures 8 through 10, spice2g6 has the highest susceptibility to context switching. Comparing Figure 13 to Figures 11 and 12

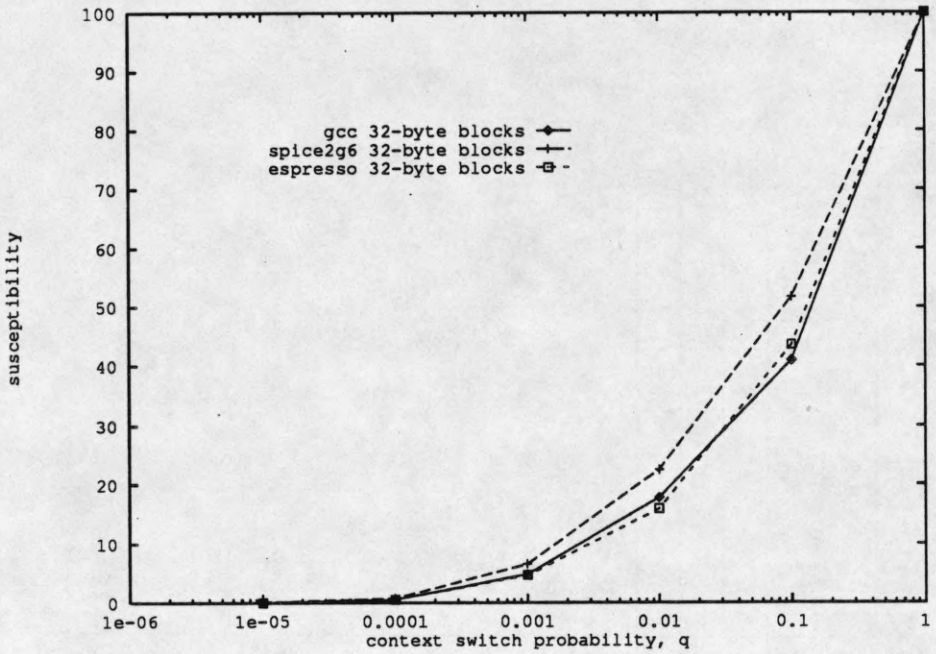

Figure 9: $\Delta\rho$ (involuntary) of gcc, espresso and spice2g6 vs. q for block size 32 bytes

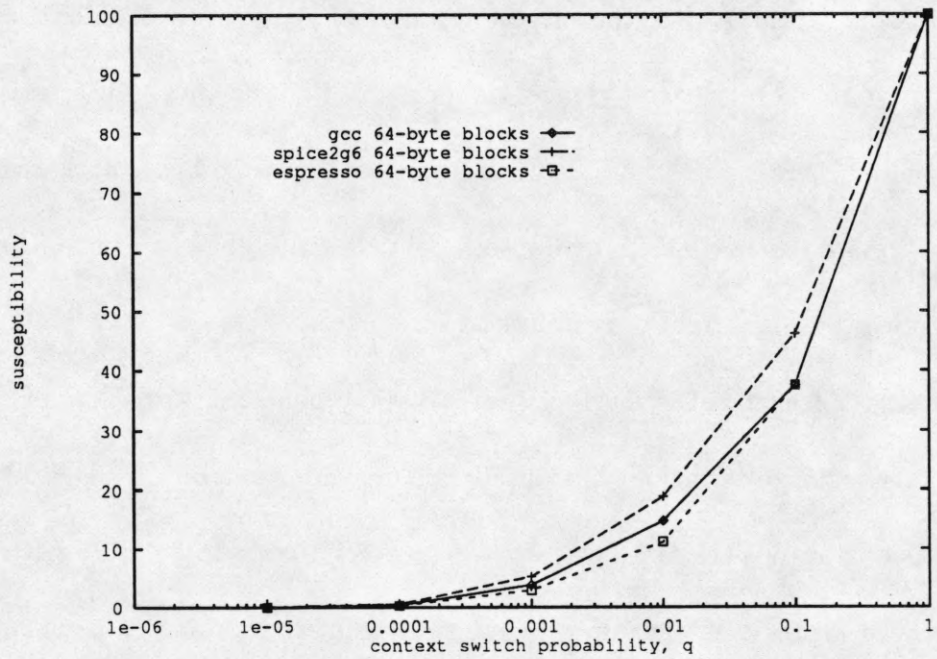

Figure 10: $\Delta\rho$ (involuntary) of gcc, espresso and spice2g6 vs. q for block size 64 bytes

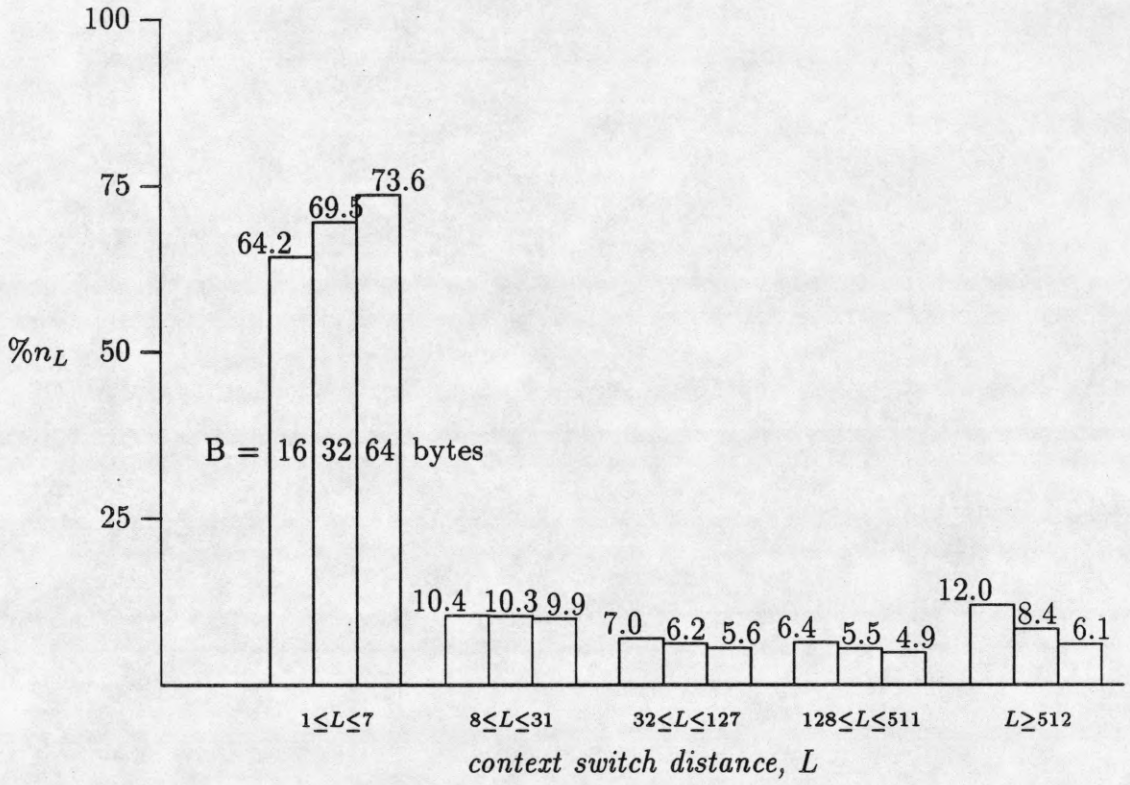

Figure 11: Distribution of n_L vs. L for gcc.

reveals the reason: spice2g6 has the largest number of recurring references separated by 512 or more intervening references. Hence, the probability of an involuntary context switch occurring should be greatest for spice2g6. Although for the 16-byte block case, spice2g6 is only slightly more susceptible to involuntary context switching than gcc or espresso, the middle categories ($8 \leq L \leq 511$) remain largest for spice2g6.

It is clear that spice2g6 is the most-susceptible of the benchmarks to involuntary context switching. The relative ordering of gcc and espresso depends more on the block size, however. For example, compare $\Delta\rho(16)$ in Figure 8 where gcc is more susceptible with $\Delta\rho(64)$ in Figure 10, where espresso is the more penalized. The n_L vs. L distribution for gcc in Figure 11 shows that gcc has less variation of run length between different block sizes. This is due in-part to the high use of quick recurrences in gcc: the category $1 \leq L \leq 7$ is the

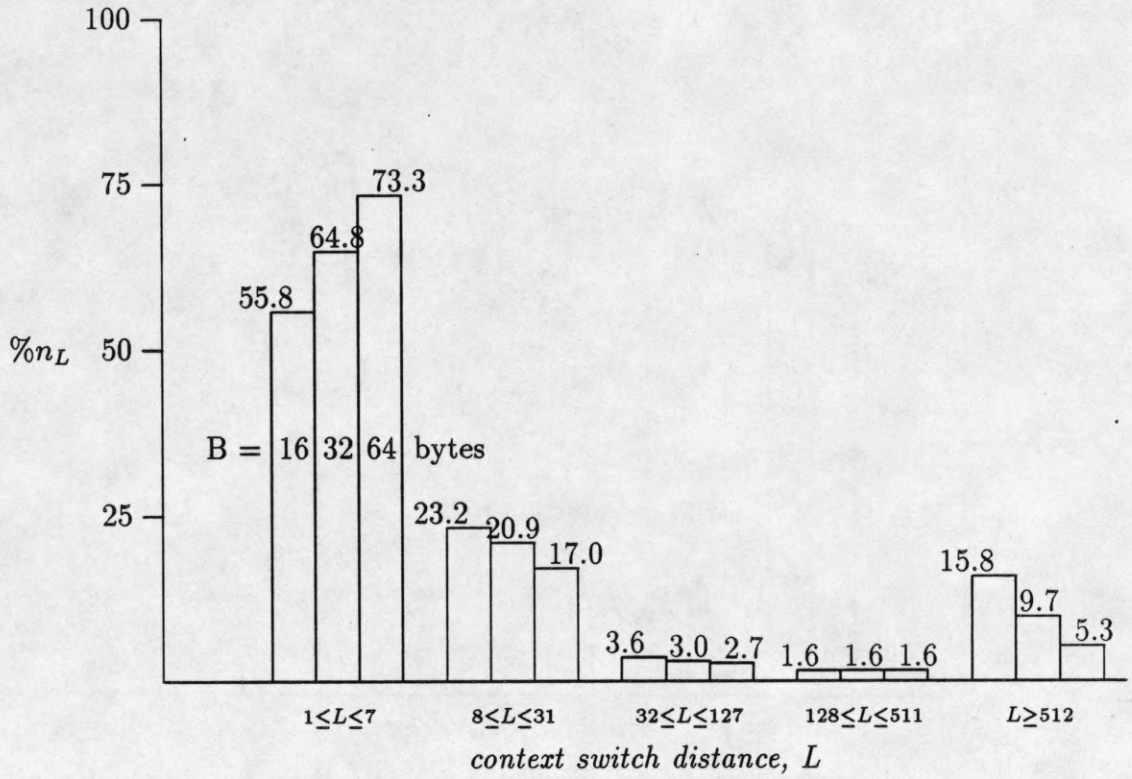

Figure 12: Distribution of n_L vs. L for espresso.

larger for gcc than for spice2g6 or espresso for 16-byte and 32-byte block sizes. This explains the higher dependence on block size for espresso over gcc.

The middle categories of L decline relatively gradually for gcc compared with espresso. Espresso has a sharp decrease in n_L between sizes $8 \leq L \leq 31$ and $32 \leq L \leq 127$. This can be seen in the transition between $q = 0.1$ and $q = 0.01$ and explains why the ordering of gcc and espresso reverses for block size 32-bytes (Figure 9). Similar effects are apparent in Figures 8 and 10. In general, larger block sizes are less susceptible to context switching effects, although the differences are less pronounced than the differences between the benchmarks themselves.

It is useful to select one of the benchmarks that has relatively average behavior and use this benchmark as a representative for the benchmark set. From the graphs of $\Delta\rho$ and n_L

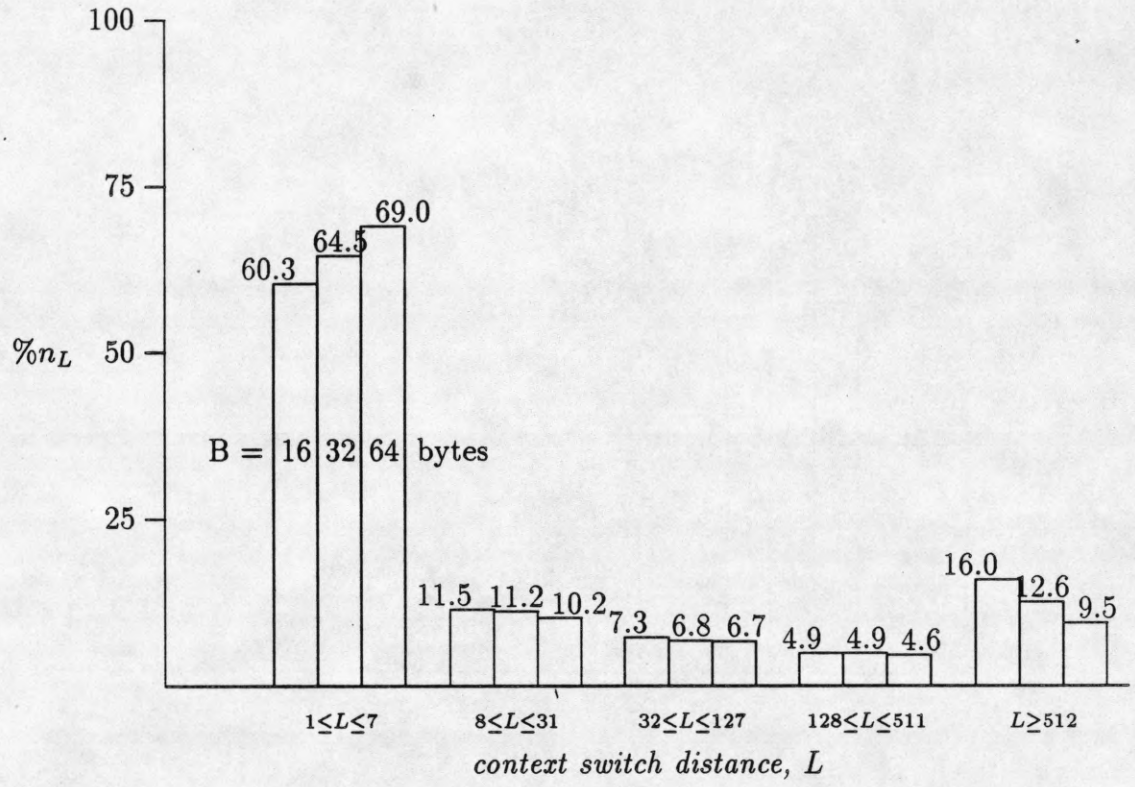

Figure 13: Distribution of n_L vs. L for spice2g6.

vs. L, gcc has the median context switching susceptibility. Therefore, gcc will be used in several of the discussions below.

3.2 Voluntary context switching susceptibility

The susceptibility of benchmarks to voluntary context switching effects is relatively small

Table 2: Voluntary context switching susceptibility vs. block size.

	Block size (bytes)		
Benchmark	16	32	64
gcc	3.1%	2.0%	1.4%
espresso	0.03%	0.02%	0.01%
spice2g6	5×10^{-6}%	3×10^{-6}%	2×10^{-6}%

compared to the involuntary effects. This can be seen in Table 2 presents the voluntary

susceptibility ($\Delta\rho$) for fully-associative caches of the largest dimension. A previous study that measured the occurrences of voluntary context switches found that they rarely occurred for these benchmarks [18]. This may well be a quirk of the benchmarks and should not be taken as a general statement that voluntary context switches do not matter.

3.3 Dimensional conflict effects

Thus far, the dimensional conflicts have been excluded from consideration to isolate the effects of context switching. The relative importance of dimensional conflicts to multiprogramming conflicts is interesting because it might indicate that some cache designs are more resilient to context switching than others. Consider caches of size 1K bytes: this is a fairly small size and therefore should experience a high percentage of dimensional conflicts. Figure 14 shows $\Delta\rho$ vs. q for gcc using caches of 1K-bytes and several set-associativities. Calculating the miss ratios for the uniprogrammed case for gcc reveals a variation of 18% for $(10, 4, 0)$ to 15% for $(10, 4, \infty)$ (this data is not shown in the figure). However, there is much less variation in $\Delta\rho$ apparent in Figure 14. This same effect is apparent from the data collected for the other two benchmarks.

The above confirms that dimensional conflicts dominate over context switching effects for small caches. To quantify this, the ratio C_M/C_D can be used as a measure of the relative impact of multiprogramming conflicts. This ratio is plotted against q using caches of dimension $(10, 4, -)$ and $(13, 4, -)$ for gcc and the results are shown in Figure 15. The figure demonstrates that for small q, dimensional conflicts dominate. The two kinds of conflicts have equal effect (i.e., $C_M/C_D = 1.0$) for $q \approx 0.02$ with caches $(10, 4, -)$ and for $q \approx 0.00003$ with caches $(13, 4, -)$. From a relative standpoint, the performance of caches with

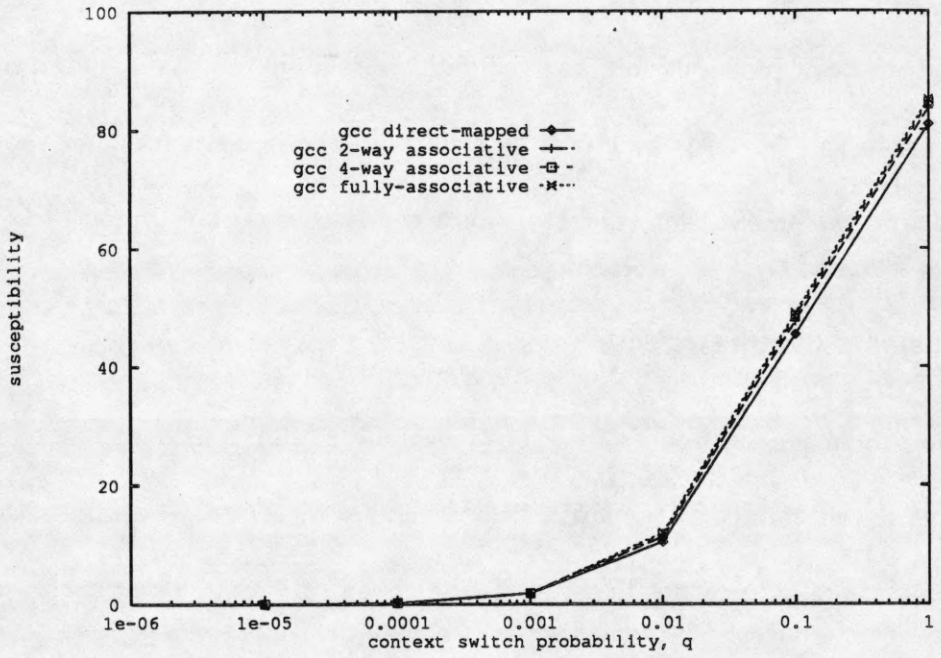

Figure 14: $\Delta\rho$ (involuntary) of gcc for caches $(10, 4, -)$.

higher associativity depends more on the multiprogramming effects. Also, the importance of associativity increases with overall cache size. This implies that when associativity is used, multiprogramming effects can decide the cache size, which is similar to the observations of [12] concerning associativity.

To show the effects observed are not an artifact of the test cache sizes of 1K and 8K bytes, Figure 16 presents C_M/C_D ratios for various cache and block sizes. Any value of q would have been sufficient to demonstrate the general relationship between C_M/C_D and C. The data from Figure 15 was used to select $q = 0.02$ for Figure 16. Since in this region the effects of associativity are relatively minor, the associativity is fixed at 2-way associative (e.g., all caches $(-, -, 1)$). (Note that here, unlike the earlier figure, C_M/C_D is presented using a logarithmic scale). From the figure, it is immediately apparent that the relative impact of multiprogramming (i.e., C_M/C_D) increases linearly with cache size. Also, as a

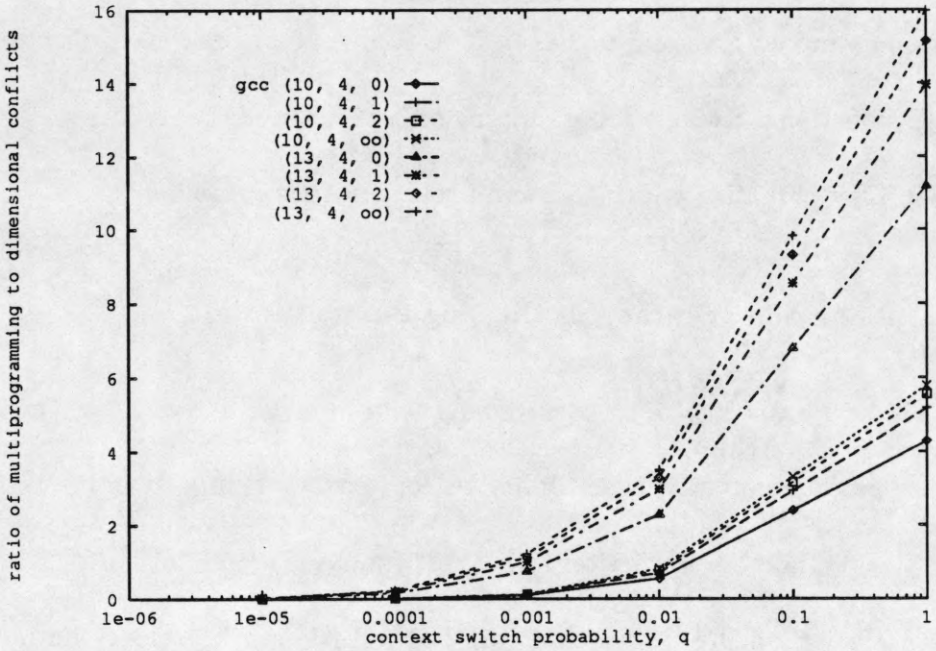

Figure 15: C_M/C_D vs. q for gcc, caches $(10, 4, -)$ and $(13, 4, -)$.

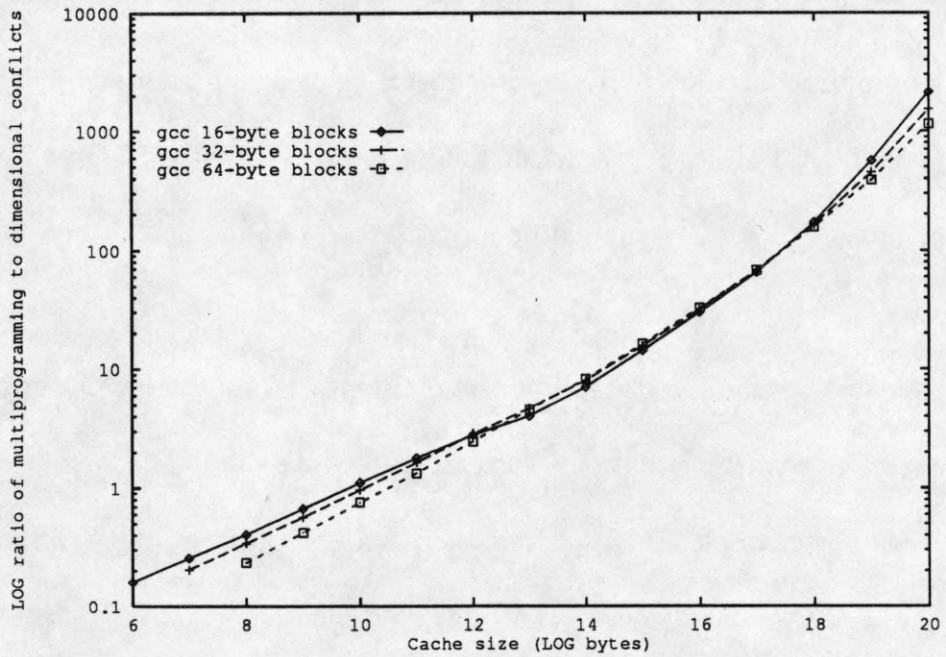

Figure 16: $\log(C_M/C_D)$ vs. cache size, for various block sizes ($q = 0.02$).

refinement of the observations made in Section 3.1, block size is inversely proportional to multiprogramming impact for small caches (less than 2K bytes). However, block size appears to be directly proportional to multiprogramming impact for moderately-large cache sizes (4K bytes up to 256K bytes), after which the trend reverses itself again.

3.4 Comments on the accuracy of the single-pass method

Up to now, the emphasis has been placed on the results gathered using the single-pass method to measure benchmark susceptibility to context switching. It is interesting to ask how accurate the method is. Two alternative test methods were chosen for a comparison. The fixed-interval method flushes the contents of the LRU stack every Q number of references. Fixed-interval flushing is similar to the approaches of [5] and [6], among others. The equivalent involuntary context switch probability is $q = 1/Q$. The second test method is to flush the contents of the stack based on a uniformly-distributed random number with mean q. This random-interval method better approximates the single-pass method of this paper, therefore it should yield closer results than the fixed-interval method. Observe that each separate value of Q requires a re-simulation for these two test methods, whereas this is not true for the single-pass method.

Selecting a realistic value of Q for for the test methods guarantees that the error observed has some meaning in realistic situations. Several empirical values of Q have been reported in the literature. For example, $Q = 6418$ for the VAX 11/780 using the VMS OS [21] and $Q = 19353$ for the VAX 8800 also using VMS [22]. As mentioned above, $Q \approx 10000$ for the VAX 11/780 BSD Unix implementation [14]. Hence, the median of $Q = 10000$ was selected for the error analysis as a relatively realistic value.

Table 3 presents the absolute RMS error between the single-pass method and the two test methods calculated across all cache dimensions in the design space. In general, the fixed-

Table 3: Absolute RMS error between single-pass and test methods (gcc, $q = 0.0001$).

	Block size (bytes)		
Test Method	16	32	64
fixed-interval	0.64%	0.37%	0.23%
random-interval	0.11%	0.12%	0.10%

interval method has higher error than the random-interval method. Several other researchers have commented that fixed-interval flushing is overly pessimistic, which may account for this phenomenon [8],[11],[12]. The most striking feature of the table is the relatively small magnitude of the error. To view this error graphically, the miss ratios for the single-pass method and the two test methods are plotted against cache size for a block size of 32 bytes in Figure 17. The figure demonstrates the the error is incurred after the miss ratio has leveled off. This occurs in the larger cache sizes when dimensional conflicts become rare. From Table 1, gcc achieves a miss ratio of $\rho \leq 1\%$ for caches of size $(16, 5, \infty)$, and greater, which corresponds to the location of the knee of the curve in Figure 17.

It is interesting to compare the overall results using the single-pass method with the results of Agarwal, *et at.* which where obtained using a microcode-assisted trace collection technique of actual interactive workloads on a VAX architecture [12]. That paper presents a distribution for the effective Q values, which can be used to determine an average of $q \approx 5.26 \times 10^{-5}$ and $q \approx 6.25 \times 10^{-5}$ for the MUL3 and MUL10 workloads of the paper, respectively. These values were used as the context switching intensities for gcc to generate Figure 18. Of course, a direct match with the results of Agarwal, *et al.* should not be

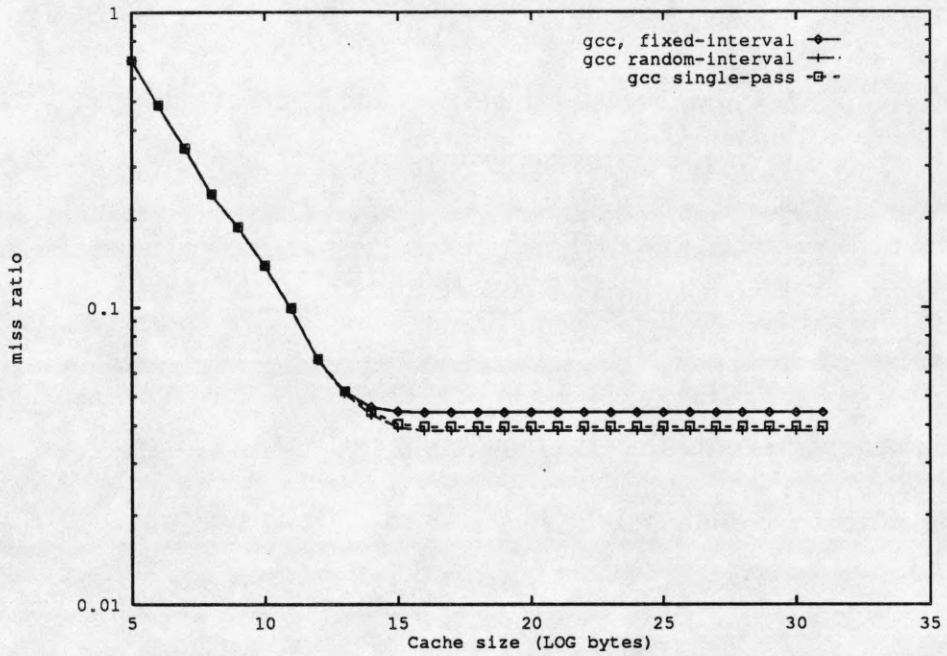

Figure 17: Miss ratio for single-pass and test methods (gcc, $q = 0.0001$) for caches $(-, 5, \infty)$.

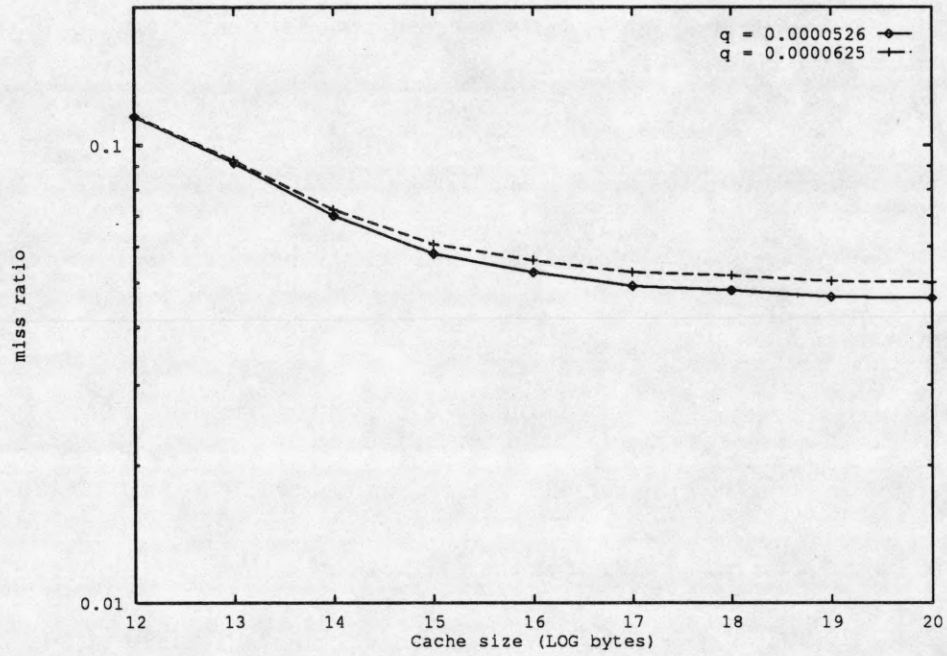

Figure 18: Miss ratio for gcc assuming $q = 5.26 \times 10^{-5}, 6.25 \times 10^{-5}$ for caches $(-, 4, 0)$.

expected since the workloads themselves differ between this study and theirs. However, the curves of Figure 18 appear very similar to those of Agarwal, *et al.* (see Figure 13 parts (a) and (b) ("Purge") of [12]). This is a strong result, indicating the method accurately models the behavior of real multiprogramming environments.

4 Conclusion

This paper presented a method for constructing the susceptibility of a benchmark to the effects of multiprogramming on cache performance. This was done by extending existing single-pass methods to measure the susceptibility in terms of potential victims of context switching. The method removes the single-pass simulation's dependence on involuntary context switching intensity and system load effects, allowing performance to be calculated for values of these parameters without the need for re-simulation. This generalization of single-pass methods extends their usefulness into domains where multiple-pass methods are the only option.

The experimentation performed in this paper revealed that a benchmark's susceptibility to context switching can be minimized by using large block sizes with small and large cache sizes. Interestingly, for medium-sized caches (4K–256K bytes for gcc) small block sizes minimize the impact of context switching.

An increase in context-switching intensity has an roughly-linear effect on a benchmark's susceptibility. For all but extremely high intensities, dimensional conflicts dominate the miss ratio. Since all the benchmarks elicited very small involuntary context switching distances, a relatively high intensity of context switching ($q \geq 0.0001$) was needed to have any noticeable effects at all. Notice that this critical value of $q = 0.0001$ corresponds to $Q = 10000$, a

realistic value according to the literature [21],[22],[14].

It is not true that all workloads will have susceptibilities similar to the SPEC benchmark members spice2g6, gcc, and espresso. However, the method itself is not limited to a specific type of benchmark. Other results are easily generated. The benchmark results were useful to demonstrate the approach's validity. It was shown to perform comparable to less-general multiple-pass test methods. Also, the behavior of the multiprogramming miss ratio agrees with actual multiprogramming behavior results presented by other researchers, suggesting the results obtained using the single-pass method are reliable for design purposes.

Acknowledgements

The authors would like to thank all members of the IMPACT research group for their support, comments and suggestions. This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR Corp., the AMD Corp. 29K Advanced Processor Development Division, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS), and the Office of Naval Research under Contract N00014-88-K-0656, and an equipment domation from the Hewlett-Packard Co.

References

[1] R. L. Mattson, J. Gercsei, D. R. Slutz, and I. L. Traiger, "Evalutation techniques for storage hierarchies," *IBM Systems J.*, vol. 9, no. 2, pp. 78–117, 1970.

[2] I. L. Traiger and D. R. Slutz, "One-pass techniques for the evaluation of memory hierarchies," IBM Research Report RJ 892, IBM, San Jose, CA, July 1971.

[3] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Trans. Computers*, vol. C-38, pp. 1612–1630, Dec. 1989.

[4] T. M. Conte and W. W. Hwu, "Single-pass memory system evaluation for multiprogramming workloads," Tech. Rep. CSG-122, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, May 1990.

[5] K. R. Kaplan and R. O. Winder, "Cache-based computer systems," *Computer*, vol. 6, pp. 30–36, Mar. 1973.

[6] W. D. Strecker, "Cache memories for PDP-11 family computers," in *Proc. 3rd Ann. Int'l Symp. Computer Architecture*, pp. 155–158, Jan. 1976.

[7] G. S. Shedler and D. R. Slutz, "Derivation of miss ratios for merged access streams," *IBM J. Research and Development*, vol. 20, pp. 505–517, Sept. 1976.

[8] M. C. Easton, "Computation of cold-start miss ratios," *IEEE Trans. Computers*, vol. C-27, pp. 404–408, May 1978.

[9] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[10] D. W. Clark, "Cache performance in the VAX-11/780," *ACM Trans. Computer Systems*, vol. 1, pp. 24–37, Feb. 1983.

[11] I. J. Haikala, "Cache hit ratios with geometric task switch intervals," in *Proc. 11th Ann. Int'l Symp. Computer Architecture*, (Ann Arbor, MI), pp. 364–371, June 1984.

[12] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Trans. Computer Systems*, vol. 6, pp. 393–431, Nov. 1988.

[13] J. G. Thompson, *Efficient analysis of caching systems.* PhD thesis, Computer Science Division, University of California, Berkeley, California, Oct. 1987. Report No. UCB/CSD 87/374.

[14] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison-Wesley, 1989.

[15] A. Agarwal, M. Horowitz, and J. Hennessy, "An analytical cache model," *ACM Trans. Computer Systems*, vol. 7, pp. 184–215, May 1989.

[16] "Spec newsletter," Feb. 1989. SPEC, Fremont, CA.

[17] J. R. Larus, "Abstract execution: a technique for efficiently tracing programs," tech. rep., Computer Sciences Department, University of Wisconsin–Madison, Feb. 1990.

[18] T. M. Conte and W. W. Hwu, "Benchmark characterization," *IEEE Computer*, pp. 48–56, Jan. 1991.

[19] R. M. Stallman, *Using and porting GNU CC.* Free Software Foundation, Inc., 1989.

[20] S. I. Feldman, D. M. Gray, M. W. Maimore, and N. L. Schryer, "A Fortran-to-C converter," Computing Science Tech. Report 149, AT&T Bell Laboratories, Murray Hill, NJ, June 1990.

[21] J. Emer and D. Clark, "A characterization of processor performance in the VAX-11/780," in *Proc. 11th Ann. Int'l Symp. Computer Architecture*, (Ann Arbor, MI), p. ??, June 1984.

[22] D. W. Clark, P. J. Bannon, and J. B. Keller, "Measuring VAX 8800 performance with a histogram hardware monitor," in *Proc. 15th Ann. Int'l Symp. Computer Architecture*, pp. 176–185, May 1988.