The Susceptibility of Programs to Context Switching

Wen-mei W. Hwu, *Member, IEEE*
Center for Reliable and High-Performance Computing
University of Illinois
Urbana IL 61801 USA
e-mail hwu@crhc.uiuc.ecu.

Thomas M. Conte
Department of Electrical and Computer Engineering
University of South Carolina
Columbia SC 29208 USA
e-mail: conte@ece.scarolina.edu

***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, the recurrence/conflict model, and extends the model for recording the effects due to both voluntary context switches (e.g., system calls) and involuntary context switches (e.g., time quantum expiration). Involuntary context switches are modeled using the distribution of lengths between a reference to an address and the re-reference to the same address. The paper makes the assumptions that involuntary context switches are equally likely to occur between each reference, and that one can independently estimate, f_{CS}, the fraction of a cache's contents flushed between context switches. The case where $f_{CS} = 1$ is used to measure the effect of worst-case context switch penalty (the susceptibility) of several members of the SPEC89 benchmark set to context switching. Some empirical results of f_{CS} are presented to illustrate the case where $f_{CS} < 1$. The model is validated against its assumptions by comparing its results with more restrictive methods.*

Index Terms—Multiprogramming, cache, simulation, single-pass algorithm, memory hierarchy, performance analysis, benchmarking, SPEC.

Manuscript received March 25 1991; revised October 1992. This was supported by the National Science Foundation (NSF) under Grant MIP-8809478, AT&T Global Information Solutions, 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-K0656, and an equipment donation from the Hewlett-Packard Co.

IEEE Log Number 9400780.

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

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]-[5]. 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 from a context switch 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 the cache contents that is actually purged by intervening processes is determined by the 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 used to measure the effects of context switching [6]-[14]. The earliest approaches flushed the cache being simulated at fixed intervals in the trace [6], [7]. Shedler and Slutz [8] approached the problem by stochastically merging several memory reference traces. Easton [9] used the average working set size of the memory reference trace to estimate cold-start miss ratios. Haikala [12] 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 [10] to measure multiprogramming effects. Also, measurements of actual multiprogrammed workloads were performed by Clark [11], Agarwal *et al.* [13], and Mogul and Borg [15]. Apart from the approximations of Easton [9] and Haikala [12], 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%–12% degradation in performance [11]–[13], 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. This is the first such study to make the dichotomy between program susceptibility and multiprogramming effects. The measured performance of the method is compared to results from periodic and random flushing of the LRU stack.

II. 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 (i.e., that *miss)* for a cache at a level of the memory system hierarchy over the total number of references made at that level. 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. A given miss ratio can be used to decide whether or not a potential memory element technology will meet the required access time for the memory system [6].

The recurrence/conflict model of the miss ratio is best illustrated with an example. Consider the trace of Fig. 1. The *recurrences* in the trace are accesses *e*, *f*, *g* and *h.* In the ideal case of an infinite cache, the miss ratio may be expressed as

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

Originally published in *IEEE Trans. Computers,* Sept. 1994, pp. 994–1003.

Reference	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
Address	0	1	2	3	1	2	1	0

Fig. 1. An example trace of addresses.

where *R* is the total number of recurrences and *N* is the total number of references. Nonideal behavior occurs due to 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 Fig. 2. (Note that in practice, such a small cache would be impractical to build.) A miss occurs for the recurring reference *e* because reference *d* purges address 1 from the cache due to insufficient cache capacity. Similarly, a miss occurs for recurring reference *h* due to reference *c*. References *d* and *c* represent a dimensional conflict for the recurrences *e* and *h*, respectively. The other misses, *a, b. c* and *d*, occur because these are the first references to addresses 0, 1, 2 and 3, respectively. The following formula can be used for deriving cache miss ratio, p, for a given trace, a given cache dimension:

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

where *D* the total number of dimensional conflicts. (For the example, $\rho = (8 - (4 - 2))/8 = 0.75$.) This is a general model and can be extended to account for other effects. This paper extends this model to address conflicts due to context switching.

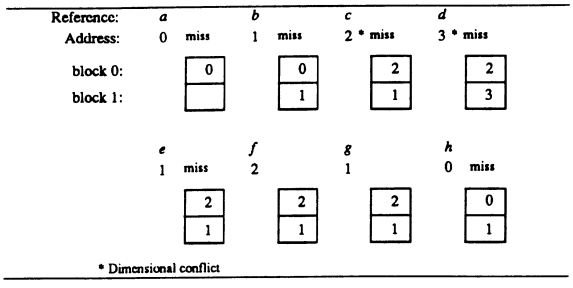

Fig. 2. An example two-block direct-mapped cache behavior.

A multiprogramming conflict is defined as an event which converts a recurrence into a miss due to a context switch. For example, both *f* and *g* are dimensional hits of the cache in Fig. 1. If a context switch occurs between references *e* and *f* which purges addresses 1 and 2 from the cache, two multiprogramming conflicts will occur, one to reference *f* and one to reference *g*. Equation (2) can be extended to account for these multiprogramming conflicts:

$$\rho = \frac{N - (R - D - M)}{N}, \tag{3}$$

where *M* the total number of multiprogramming conflicts.

A. Reference Streams and Cache Dimensions

A formal abstraction of a benchmark's trace is termed a *reference stream.* This is a sequence of references to addresses, *w(k),* of length *N (O ≤ k < N).* When required, the addresses are 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. Note that a reference at *w*(*k*) occurs later than *w(k – 1)* in time, but the parameter *k* does not represent parameterized time since it does not take into account the difference in service times between cache hits and cache misses. For this reason, *k is* referred to as the *reference count.* The trace also contains information about voluntary context switching. *A* reference is called a *voluntary context-switch event* if the benchmark relinquishes the CPU after the reference (e.g., a system call is 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. The term *set size is* used to mean associativity level, or the number of blocks per set. *Cache size is* the total number of bytes per cache. *Block size* has been called *line size* elsewhere [10]. 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 l-KB direct-mapped cache with a block size of 64 bytes; and, a cache of dimension (21, 10, 11) (alternately, (21, 10, ∞)) is of size 2 MB with l-KB-length blocks and it is fully-associative. A dash is substituted for an entry in the triple to indicate all caches of that dimension: (– , 5 ,1) are all caches with block size of 32 bytes and having two-way associativity. Caches are assumed to use LRU replacement and map addresses into sets using bit selection [3].

Originally published in *IEEE Trans. Computers,* Sept. 1994, pp. 994–1003.

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 of each address to zero.

B. Least Recently Used (LRU) Stack Operation

LRU stacks were first introduced by Mattson, *et al.* in [1] as a way to model the behavior of paging systems. 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), it is removed from the stack, then repushed onto the stack. This is illustrated in Fig. 3 for the example reference stream at the beginning of this section (Fig. 1). This stack maintenance policy is specific to a particular block size, as is the discussion below.

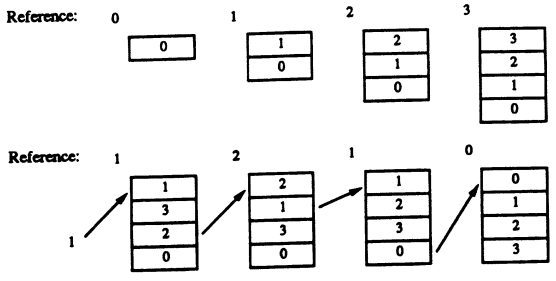

Fig. 3. An example of LRU stack operation.

A 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 $S_B(k) = \{S_B(k)[0], S_B(k)[1], \ldots, S_B(k)[m]\}$' where *m* is the depth of the stack. The following operations are defined for a stack: the **push**( ) function,

push$(S_B(k), \alpha) = \{ \alpha\, S_B(k)[0], S_B(k)[1], \cdot, S_B(k)[m] \}$,

the $\Delta(\cdot)$ function,

$$\Delta(S_B(k), \alpha) = i, \text{ if } S_B(k)[I] = \alpha$$

and, the **repush**( ) function,

repush$(S_B(k), \alpha)$

$$= \begin{Bmatrix} \alpha, S_B(k)[0], S_B(k)[1], \ldots, S_B(k)[\Delta(S_B(k), \alpha) - 1], \\ S_B(k)[\Delta(S_B(k), \alpha) + 1], \ldots, S_B(k)[m] \end{Bmatrix}$$

$\Delta(S_B(k), \alpha)$ 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 use $\Delta = \Delta(S_B(k), \alpha)$. 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 Fig. 4. In Step 1.1, the references between the top of stack and the recurring reference have been referred to as the set $\Gamma = \{\beta i \mid \beta i = S_B(k-1)[i], 0 \le i \le \Delta\}$.

Fig. 4 is applied to $\alpha = w_B(k)$ for all *k*. The LRU policy is essentially a definition for calculating $S_B(k)$ from $S_B(k-1)$ and α. In most situations, $S_B(k)$ is calculated in order to obtain other statistics, such as the stack depth distribution. (Step 1.1 is explained in detail in [16].

1. if $\alpha \in S_B(k-1)$ then
1.1 *determine D from* Γ
1.2 $S_B(k) \leftarrow$ **repush**$(S_B(k-1), \alpha)$,
2. else $S_B(k) \leftarrow$ push$(S_B(k-1), \alpha)$
3. $N \leftarrow N + 1$

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

Originally published in *IEEE Trans. Computers,* Sept. 1994, pp. 994–1003.

C. Types of Context Switching

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. Page faults are treated as involuntary context switches because page faults depend on the interaction of processes in the system, whose interaction is assumed to be pseudo-random in nature.

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 [12]. This probability is denoted, q, and termed the involuntary *context switching intensity*. Separation of the system's characteristics from the characteristics of the benchmark allows many different systems to be considered without resimulating 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 III-A 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 f_{CS} 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 f_{CS} highly dependent on several volatile variables and therefore difficult to measure. (Several empirical estimates of f_{CS} are presented in Section III-E.) Some virtual memory system implementations force a cache flush to eliminate problems with page sharing of writable pages [13]. Also, it has been shown that for small cache sizes, a context switch effectively flushes the cache, therefore $f_{CS} = 1$ [10]. For larger caches, this provides an upper bound for the effects of context switching.

D. 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 of the context switch. The numbers of each type of potential victims are defined as $X_V[C, B, S]$ and $\overline{X}_I[C, B, S, q]$, for all voluntary and involuntary context switches, respectively. $X_V[C, B, S]$ is the total number of potential victims due to voluntary context switching for caches of dimension (C, B, S). $\overline{X}_I[C, B, S, q]$ 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,

$$M[C,B,S,q] \equiv f_{CS}\,(X_V[C,B,S] + \overline{X}_I[C,B,S,q]). \qquad (4)$$

The equation for the miss ratio (2) can be modified to take into account the new conflicts,

$$\begin{aligned}\rho &= \frac{N-(R-D-M)}{N} \\ &= \frac{N-(R-D-F_{CS}(X_V[C,B,S]+\overline{X}_I[C,B,S,q]))}{N}.\end{aligned} \qquad (5)$$

Determining the multiprogramming conflicts involves measuring X_V and $\overline{X}_I$ from the reference stream. The measurement can be done by extending the recurrence/conflict single-pass technique. The miss ratio is then calculated by first calculating $M[C, B, S, q]$ using Equation 4 for a value of f_{CS}, then using the result to complete Equation 5.

E. Multiprogramming Extensions to LRU Stack Operation

The extensions required to the recurrence/conflict single-pass technique measure X_V and $\overline{X}_I$ are shown in Fig. 6. The procedure for determining X_V $[C, B, S]$ is illustrated in Fig. 5. The procedure operates as follows: When α is processed, if it is not a recurring reference (i.e., the test of Step 1 of Fig. 6 fails), then it cannot be a victim since it cannot produce a hit. However, if α is a voluntary context switch event, it is marked as such when it is pushed on the stack in Step 2 (marked references are shown using asterisks in Fig. 5).

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

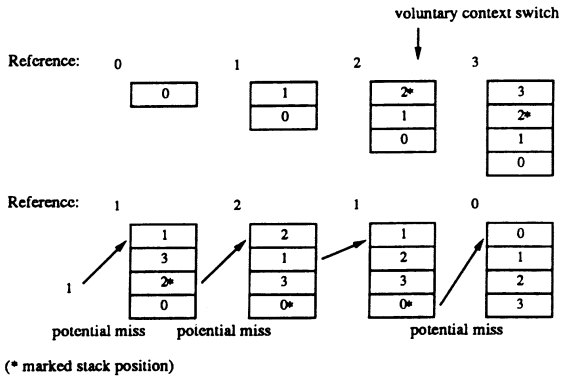

Fig. 5. An example for voluntary context switch of the modified LRU stack operation.

1. if $\alpha \in S_B(k-1)$ **then**
1.1 $vol\_cs \leftarrow$ **false**
1.2 $L \leftarrow 1$
1.3 for $i \leftarrow 0$ to Δ do
1.3.1 $\beta_i \leftarrow S_B(k-1)[i]$
1.3.2 **if** β_i *marked as a voluntary contest match event* **then**
1.3.2.1 $vol\_cs \leftarrow$ **true**
1.3.3 $L \leftarrow L + c_I(\beta_i)$
1.4 $L \leftarrow L + 1$
1.5 **for all** (C, B, S) without *a dimensional conflict* **do**
1.5.1 $n_L[C, B, S] \leftarrow n_L[C, B, S] + 1$
1.5.2 if vol_cs **then**
1.5.2.1 $X_V[C, B, S] \leftarrow X_V[C, B, S] + 1$
1.6 **if** α *marked as a voluntary context switch event* **then**
1.6.1 *mark* $\beta_{\Delta+1}$,
1.6.2 *unmark* α
1.7 $c_I(\beta_{\Delta-1}) \leftarrow c_I(\beta_{\Delta-1}) + c_I(\alpha)$
1.8 $c_I(\alpha) \leftarrow 1$
1.9 $S_B(k) \leftarrow$ **repush**$(S_B(k-1),\alpha)$,
2. **else**
2.1 $c_I(\alpha) \leftarrow 1$
2.2 $S_B(k) \leftarrow$ **push**$(S_B(k-1),\alpha)$

Fig. 6. An LRU stack method modified for context switching.

If α is a recurring reference, $X_V[C, B, S]$ *is* conditionally incremented if a marked reference is encountered when the dimensional conflicts are calculated. $X_V[C, B, S]$ is only incremented for all dimensions in which α does not have a dimensional conflict. If $X_V[C, B, S]$ were incremented for all dimensions, a reference might be counted more than once as a conflict, once as a multiprogramming conflict and once as a dimensional conflict. Notice that the references immediately below repushed, marked references inherit the marking in Fig. 5 (Step 1.6 and its substeps of Fig. 6). This is done to insure all subsequent recurring references that cross the context switch event are subject to a voluntary context switch.

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

The procedure for determining $\overline{X}_I[C, B, S, q]$ using an LRU stack is somewhat more complicated than that for determining $X_V[C, B, S]$. 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 events for the recurring reference α at reference count 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,

$$p_L = \sum_{j=1}^{L} \binom{L}{j} q^j (1-q)^{L-j}. \quad (6)$$

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

$$\overline{X}_I[C, B, S, q] = E\left[n_L[C, B, S]\right] = \sum_L p_L n_L[C, B, S]. \quad (7)$$

Equation 7 expresses the expected number of potential victims due to involuntary context switching.

The equation 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.

Figure 7 shows an example of calculating $\overline{X}_I[C, B, S]$. The figure shows that a counter of the number of context switch events affecting α is kept, defined as $c_I(\alpha)$. Initially and after a recurring reference is repushed, $c_I(\alpha) \leftarrow 1$ (Step 2.1 and 1.8 of Fig. 6). In Step 1.3 and its substeps and Step 1.4, L is computed from one plus the sum of the counters of entries above α on the stack. (Notice that $c_I(\alpha)$ is not part of the calculation of L, Fig. 7 illustrates this). In Step 1.5 and its substeps, $n_L[C, B, S]$ 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 $S_B(k-1)$. As a bookkeeping step, $C_I(\beta_0)$ is incremented by $c_I(\alpha)$ (Step 1.7). 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 algorithm shows $n_L[C, B, S]$ being maintained for all values of L. Not all values of L must be recorded using $n_L[C, B, S]$. Rather, power-of-two sized categories can be retained. The scheme used for the simulations that is presented below uses 14 categories. The first category contains $n_L[C, B, S]$ for $1 \le L < 4$, following this, the ith category contains $n_L[C, B, S]$ for $2^{(i+2)} \le L < 2^{i+3}$. This quantization scheme is based on observations of the distribution of $n_L[C, B, S]$ vs. L. The scheme does however produce error for small q, and this is commented on in the following section.

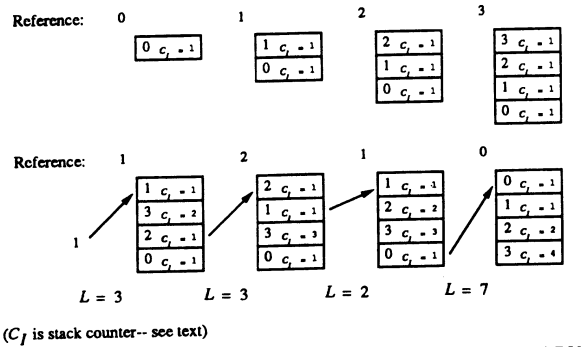

Fig. 7. An example for involuntary context switching of the modified LRU stack operation.

Notice that the calculation of $n_L[C, B, S]$ is independent of the context switching intensity distribution assumptions. The function used to calculate p_L in (7) need not be (6). It is possible to substitute other context switching intensity distributions into (7) without altering the presented single-pass method. The impact of this observation is that the method is more general than the assumption of uniformly-distributed involuntary context switching of (6).

III. Empirical Results of Program Susceptability

The validity of the single-pass method of the previous section is discussed below by comparing the method's results with the results from other techniques that have similar assumptions. The results from the model are presented and discussed for members of the SPEC89 benchmark set presented in Table I (from [18]).

The dimensional conflicts that occur due to different cache sizes are discussed in Section 3.4 to compare their performance degradation with that of context switching. Empirically observed values of the parameter f_{CS} are also presented. It is found that $f_{CS} < 1$ for moderate multiprogramming loads, confirming the observation that $f_{CS} = 1$ produces overly-pessimistic results.

A. The Validity of the Single-Pass Method

It is important to question whether the single-pass method extended to measure context switching produces performance estimates that are consistent with the assumptions made in Section II-C. (Whether these assumptions are valid themselves is beyond the scope of this study.) The approach used in testing the validity of the method is to compare its predictions against methods used for traditional cache simulators.

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

Table I. Benchmark Set (From SPEC89 Benchmark Suite).

Benchmark	Description
doduc	Monte Carlo simulation of the time evolution of a thermohydraulical modelization for a nuclear reactor .
eqntott	Generates truth table from logic equations.
espresso	Performs PLA optimization.
gcc	GNU C compiler, version 1.35.
matrix300	Performs 300 x 300 matrix multiply.
Xlisp	Lisp interpreter (the application) executing the Nine-Queens problem (the recursive benchmark).

One commonly-used simulation technique to measure the effects of context switching is to flush the state of the simulation at context switching events [6],[7],[10]. It is clear that in this case, $f_{CS} = 1$ is assumed. The decision of when to flush the stack for voluntary context switch events is known since these are present in the trace. The decision of when to flush the cache for involuntary context switch events is done by distributing involuntary context switch events throughout the trace uniformly. This random-interval simulation flushes the contents of the stack based on a uniformly-distributed random number with mean q. Note that the random-interval simulation requires a simulation for each value of q. The single-pass method does not have this restriction since it measures the effects of all q in one pass over the trace.

The random-interval simulation method approximates the assumptions of Section II-C, except that the simulation produces results for one particular random distribution of context switch events across the trace. The single-pass method measures the average effect of all distributions. This discrepancy can be eased by averaging the results of several random-interval simulations. Random-interval simulations are performed iteratively until the results converged.

Figure 8 present the difference between the random-interval simulation and the single-pass method for the benchmarks expressed as the absolute error of the miss ratio, for $q = 0.01$ and $q = 0.001$, respectively. Only the absolute errors for fully-associative caches are shown in the figure for brevity. Smaller levels of associativity were found to have lower error.

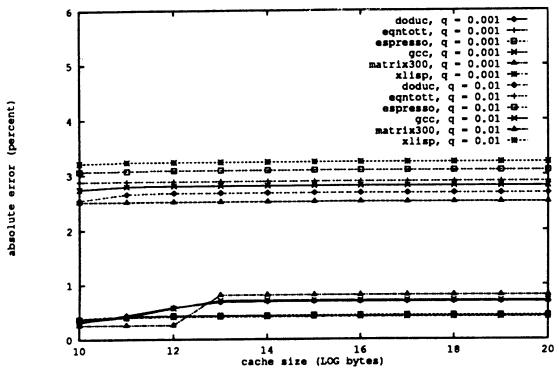

Fig 8. Absolute error of the miss ratio for random-interval simulation vs. Single-pass method, $q = 0.01$ and $q = 0.001$

The figure demonstrates that the difference between the simulation and the model is approximately 2.5% to 3.3% for $q = 0.01$ and less than 1% for $q = 0.001$. The increase in error for larger q values is due to the quantization of L discussed in the previous section. This error of 2.5% to 3.3% is large for $q = 0.01$, however empirical evidence suggests that q values are typically in the range of $q = 0.001$ to $q = 0.0002$ (calculated from [19] and [20] and assuming one reference made every five instructions). The single-pass technique using the power-of-two quantization of L produces results within 1% of the random-interval simulation for practical q values. This evidence suggests the single-pass method produces results that are consistent with its assumptions.

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

B. Involuntary Context Switching Susceptibility

It is useful to define $\Delta\rho = M[C, B, S, q]/N$ as a measure of benchmark susceptibility to context switching. This is the difference between the uniprogramming and multiprogramming miss ratios. Figure 9 presents $\Delta\rho$ for gcc and xlisp for block size 16 bytes. The results in the figure are for the cache (31,4, ∞), to eliminate the effects of dimensional conflicts. Section III-D discusses the effects of dimensional conflicts. The figure considers only involuntary context switching. The effects of voluntary context switching are discussed in Section III-C. Also, complete cache flushing *(f_{CS} = 1)* is used to emphasize the worst case context switching penalty (Section III-E discusses other values of f_{CS}*)* The figure demonstrates that when the intensity of context switching, q, is small, $\Delta\rho$ approaches zero such that context switching has little effect for $q \leq 0.0001$. This value of q corresponds to an average context switching interval of 10000 references. The gcc benchmark is slightly more susceptible to context switching than the xlisp benchmark for $q = 0.1$. This situation reverses itself and xlisp becomes more susceptible for $q > 0.01$. This phenomenon can be explained with the cumulative distribution of n_L vs. L, which is plotted in Fig. 10. The figure has a logarithmic axis for the independent variable, L. From the figure, it is apparent that xlisp has a higher number of recurrences for $L \leq 2^5$: 85% for gcc vs. 87.4% for xlisp. This would imply that a context switch frequency of greater than every $2^5 = 32$ references would impact xlisp more than gcc. This explains the behavior observed in Fig. 9.

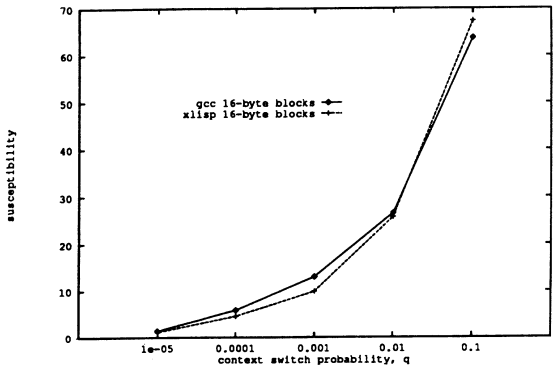

Fig 9. $\Delta\rho$ (involuntary) of gcc and xlisp vs q for block size 16 bytes cache dimension (31, 4, ∞)

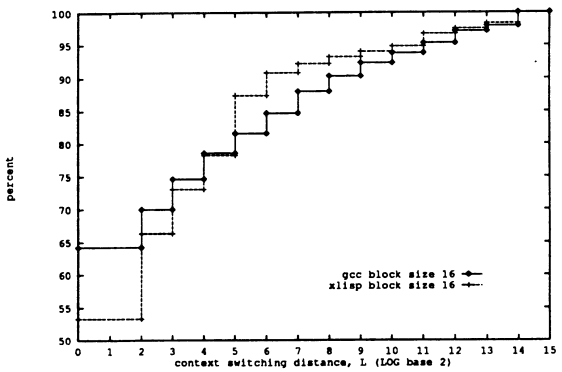

Fig 10 Cumulative distribution of n_L vs. L for block size 16 bytes, cache dimension (31, 4, ∞)

Figures 11 and 12 presents $\Delta\rho$ for gcc, espresso and xlisp for block sizes 32 bytes and 64 bytes, respectively. The data for all the benchmarks is presented in Table II. The corresponding cumulative distribution functions of n_L vs. L are presented in Figs. 13 and 14, respectively. Together, Figs. 9–12 demonstrate that the susceptibility to context switching decreases as block size increases. From Table II, for $q = 0.01$, the difference in the miss ratio is 21.2% for block size 16 bytes and 13.9% for block size 32 bytes, on average. One possible reason for this is that the block reference streams for larger block sizes have smaller context switching distances. This occurs since more references occupy the same cache block for larger block sizes than for smaller block sizes.

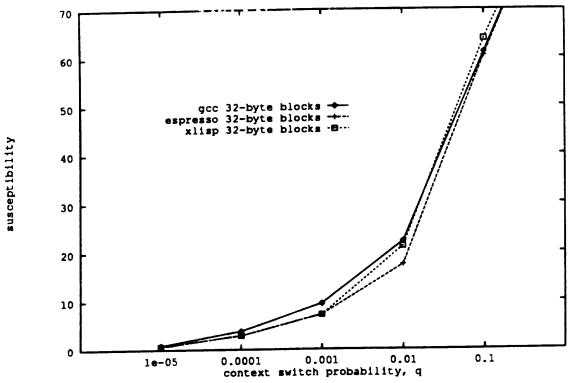

Fig. 11. $\Delta\rho$ (involuntary) of gcc, espresso and xlisp vs. q for block size 32 bytes, cache dimension (31, 5, ∞).

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

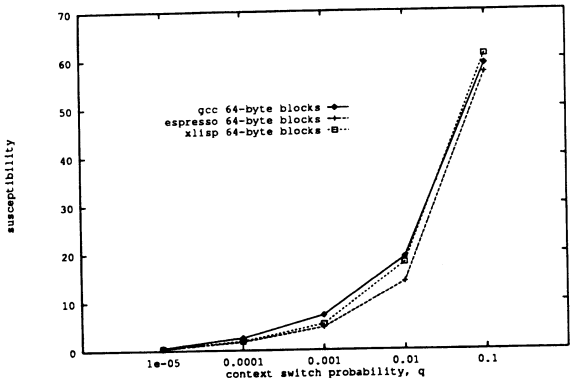

Fig. 12. $\Delta\rho$ (involuntary) of gcc, espresso and xlisp vs. q for block size 64 bytes, cache dimension (31, 6, ∞).

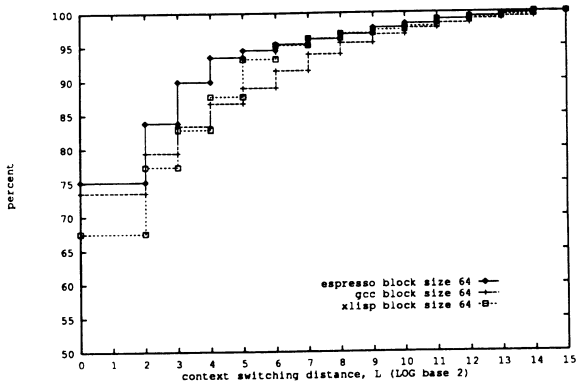

Fig. 14. Cumulative distribution of n_L, vs. L for block size 64 bytes, cache dimension (31, 6, ∞).

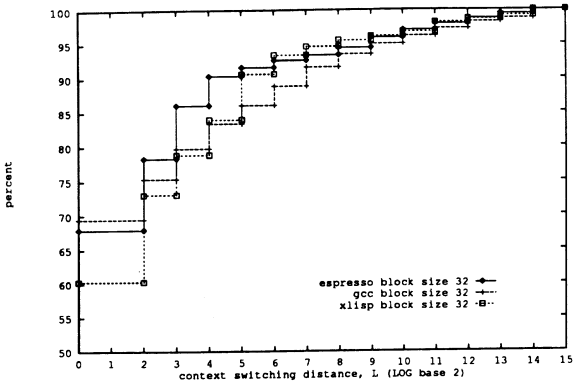

Fig. 13. Cumulative distribution of n_L vs. L for block size 32 bytes, cache dimension (31, 5, ∞).

Comparison between the benchmarks reveals significant variance in susceptibility. The matrix300 and eqntott benchmarks have the lowest change in the miss ratio, whereas benchmarks such as doduc and xlisp are quite sensitive to the value of q. For q = 0.01, $\Delta\rho$ = 37.8% for doduc compared to 5.98% for xlisp. This confirms that susceptibility is a characteristic of the benchmark and that workload choice influences the observed effects of context switching.

Table II. Involuntary context switching susceptibility ($\Delta\rho$) for caches (31, –, ∞).

	$\Delta\rho$								
Bench-mark	$q = 10^{-2}$	$B = 4$ $q = 10^{-3}$	$q = 10^{-4}$	$q = 10^{-2}$	$B = 5$ $q = 10^{-3}$	$q = 10^{-4}$	$q = 10^{-2}$	$B = 6$ $q = 10^{-3}$	$q = 10^{-4}$
doduc	37.8%	16.1%	5.34%	28.9%	10.9%	3.31%	23.9%	8.03%	2.22%
eqntott	5.98%	2.41%	1.27%	5.21%	1.98%	1.04%	4.60%	1.57%	0.79%
espresso	23.0%	11.2%	4.88%	17.5%	7.23%	2.94%	14.04%	4.71%	1.71%
gcc	26.6%	13.0%	5.91%	22.2%	9.52%	3.85%	19.1%	7.22%	2.59%
matrix300	7.78%	5.76%	3.54%	4.91%	3.01%	1.79%	3.46%	1.63%	0.91%
xlisp	25.8%	9.89%	4.56%	21.3%	7.21%	2.91%	18.0%	5.39%	1.89%
Average	21.2%	9.73%	4.35%	16.7%	6.64%	2.64%	13.9%	4.76%	1.69%
Std. Dev.	12.2%	4.516%	1.66%	9.72%	3.52%	1.04%	8.24%	2.72%	0.71%

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

C. Voluntary Context Switching Susceptibility

The susceptibility of the benchmarks to voluntary context switching effects is relatively small compared to the involuntary effects. This can be seen in Table III, which presents the voluntary susceptibility ($\Delta\rho$) for fully-associative caches of the largest dimension for gcc and espresso. The largest-dimensional fully-associative caches were selected so that $\Delta\rho$ would be at its maximum since no dimensional conflicts occur. The occurrences of voluntary context switches are rare for these benchmarks as well as for other members of the SPEC89 set [17]. This explains the small susceptibility due to voluntary context switches. This may well be an artifact of benchmark selection and should not be taken as a general statement that voluntary context switches do not have much effect. One of the benchmarks, gcc, is selected to serve as an example for the discussions that follow to illustrate the behavior of the susceptibility model.

D. Dimensional Conflict Effects

The dimensional conflicts have been excluded from consideration thus far by considering large, fully-associative caches to isolate the effects of context switching. The relative importance of dimensional conflicts to multiprogramming conflicts is interesting because some cache designs may be more resilient to context switching than others due to the influences of dimensional conflicts. Consider caches of size l-K bytes: it is selected as a worst case since it should experience a high percentage of dimensional conflicts due to its extremely small size. Figure l5 shows $\Delta\rho$ vs. q for gee using caches of l-K-bytes and several set-associativities.

The above data suggests that dimensional conflicts dominate over context switching effects for small caches. To quantify this, the ratio of the multiprogramming conflicts to the dimensional conflicts, $M[C, B, S, q]/D[C, B, S]$, can be used as a measure of the relative impact of multiprogramming conflicts to dimensional conflicts. This ratio is plotted against q using caches of dimension (10, 4, -) and (13, 4, -) for gcc and the results are shown in Fig. 16.

Table III. Voluntary Context Switching Susceptibility vs. Block Size.

	Block size (bytes)		
Benchmark	16	32	64
doduc	0.07%	0.04%	0.03%
espresso	0.03%	0.02%	0.01%
eqntott	$5.6\text{x}10^{-3}$%	$53.8\text{x}10^{-3}$%	$52.4\text{x}10^{-3}$%
gcc	3.1%	2.0%	1.4%
matrix300	$1.9\text{x}10^{-3}$%	$1.8\text{x}10^{-3}$%	$1.7\text{x}10^{-3}$%
xlisp	$6.0\text{x}10^{-3}$%	$6.0\text{x}10^{-3}$%	$6.0\text{x}10^{-3}$%

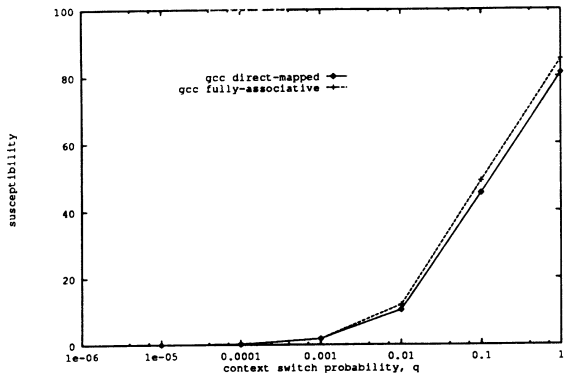

Fig. 15. $\Delta\rho$ (involuntary) of gcc for caches (10, 4, -).

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

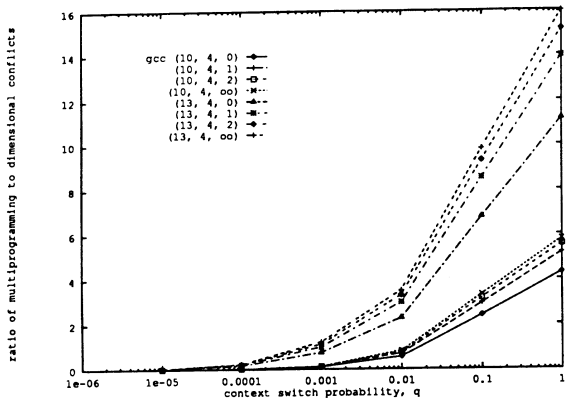

Fig. 16 *M/D* vs. *q* for gcc, caches (10, 4, -) and (13, 4, -).

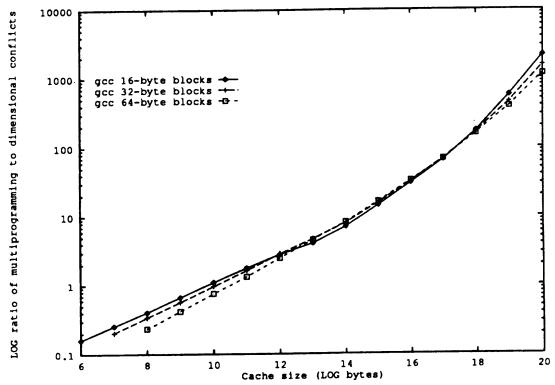

Fig. 17. log(*M*/*D*) vs. cache size, for various block sizes ($q = 0.02$).

The figure demonstrates that for small *q*, dimensional conflicts dominate. The two kinds of conflicts have equal effect (i.e., M/D = 1.0) for $q \approx 0.02$ with caches (10, 4, -) and for $q \approx 0.00003$ with caches (13, 4, -). As associativity increases, the performance depends more on the multiprogramming conflicts than dimensional conflicts. 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 [13] concerning associativity.

To show the effects observed are not an artifact of the test cache sizes of l-K and 8-K bytes, Fig. 17 presents *M/D* ratios for various cache and block sizes.

Any value of *q* would have been sufficient to demonstrate the general relationship between *M/D* and *C*. The data from Fig. 16 was used to select $q = 0.02$ for Fig. 17. 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, *M/D* is presented using a logarithmic scale). From the figure, it is immediately apparent that the worst-case relative impact of multiprogramming (i.e., *M/D)* increases approximately linearly with cache size (both axes are logarithmic). Also, as a refinement of the observations made in Section III-B, block size is inversely proportional to program susceptibility for small caches (less than 2-K bytes). However, program susceptibility appears to be directly proportional to block size for moderately-large cache sizes (4-K bytes up to 256-K bytes), after which the trend reverses itself again.

E. Some Empirical Measurement of Fraction of Cache Flush (f_{CS})

The majority of this section has assumed $f_{CS} = 1$ in order to measure the worst-case susceptibility of programs to context switching. This section presents some empirical estimates of the f_{CS} parameter. Note that these measurements are presented as some examples of f_{CS} values. They should not be used to make general conclusions about f_{CS}. The data is intended to illustrate the role of f_{CS} in the model of context switching used in this paper.

The method used is to simulate a multiprogrammed system using a trace composed of interleaved sections of traces taken from several benchmarks. At each reference, the interleaver makes a decision whether to continue processing the current trace or to switch to another waiting trace. This probability is assumed to be uniformly distributed with mean *q*.

The values of f_{CS} for each benchmark can be derived by comparing the number of misses for a uniprogramming cache to the observed number of misses for the multiprogramming cache. Any additional misses in the multiprogrammed case must be due to context switching. Let $D_u[C, B, S]$ represent the dimensional misses from the uniprogramming trace and $D_m[C, B, S]$ represent the dimensional misses measured from the multiprogramming trace. Let *M* represent the estimated multiprogramming conflicts. Then,

$$\hat{M}[C,B,S,q] = D_m[C,B,S] - D_u[C,B,S]. \qquad (8)$$

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

Note that it can always be assumed that $D_m[C, B, S] > D_u[C, B, S]$ since any dimensional conflicts for a benchmark trace must come as a result of the interleaving of traced events. Using (4) and neglecting the X_V term as justified by the experimental results of Section III-C, then,

$$\hat{f}_{CS} = \frac{D_m[C,B,S] - D_u[C,B,S]}{X_i[C,B,S,q]}, \qquad (9)$$

where $\hat{f}_{CS}$ is the experimental value for f_{CS}. Equation (9) requires knowledge of X_I, which would require use of the algorithm of Fig. 7.

The two experiments interleave espresso, gcc, and xlisp with q = 0.01 and q = 0.001. A block sizes of 32 bytes was assumed for these experiments. Values of f_{CS} across cache size and associativities for the espresso benchmark are presented in Figs. 18 (q = 0.01) and 19 (q = 0.001) from the perspective of the espresso benchmark.

It is clear from these two Figs. that f_{CS} *is* a function of cache size, as suggested by Equation 9. For small caches, $f_{CS} \approx$ 13%-18%, whereas for large caches, $f_{CS} \approx$ 8%, when q = 0.01. The effect of associativity on f_{CS} are less pronounced than the effects of cache size for q = 0.01 (Fig. 18). For q = 0.001 (Fig. 19), $f_{CS} \approx$ 0 1arge, fully-associative cache sizes. This is not true for smaller associativities, since dimensional conflicts occur between the references of the three benchmarks regardless of cache size. The effects of associativity also become less noticeable as cache size increases, possibly because less dimensional conflicts occur between the references from espresso and those of gcc and xlisp.

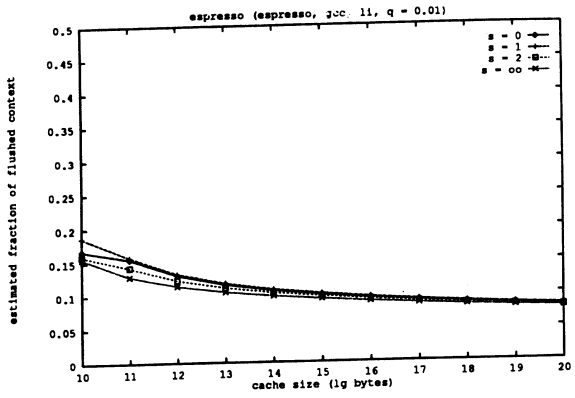

Fig. 18. f_{CS} vs. cache size for espresso, q = 0.01.

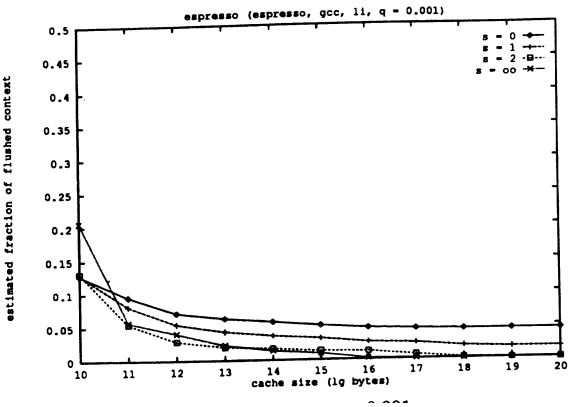

Fig. 19. f_{CS} vs. cache size for espresso, q = 0.001.

These experiments show that f_{CS} = 1 is a pessimistic assumption under moderate load. When workloads are decomposed into workload elements or workload elements are taken from standard benchmark sets such as SPEC89, it is not possible to predict the different combinations of benchmarks that may execute together in the final system. In this situation, a conservative assumption such as f_{CS} = 1 is appropriate. The results for the susceptibility measures presented above for f_{CS} = 1 suggest that the difference in the miss ratio will not change considerably for values of $q \leq 10^4$. If designs are selected to satisfy a required maximum miss redo, this observation suggests that selecting prototypes with f_{CS} = 1 adds degree of tolerance to context switching to the designs with small increase in cost.

IV. Conclusion

This paper presented a method for constructing the worst-case context switching penalty *(susceptibility)* on the cache performance of a benchmark in the presence of multiprogramming. 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 resimulation. 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.

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.

An increase in context-switching intensity has a 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 ($q > 0.0001$) was needed to have significant effects.

It is not true that all workloads will have susceptabilities similar to the SPEC89 benchmark members considered here. However, the method itself is not limited to a specific type of benchmark. Other results are easily generated. The benchmark results were shown to demonstrate the approach's usefulness and validity. It was shown to perform comparable to less-general multiple-pass test methods. Also, the behavior of the multiprogramming miss redo agrees with actual multiprogramming behavior results presented by other researchers, suggesting the results obtained using the single-pass method are reliable for design purposes.

Acknowledgment

The authors would like to thank all members of the IMPACT research group for their support, comments and suggestions.

References

1. R. L. Mattson, et al., "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.*, Vol. 9, no. 2, pp. 78-1 17, 1970.
2. L. Traiger and D. R. Slutz, "One-Pass Techniques for the Evaluation of Memory Hierarchies," IBM Res. Rep. 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. 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, Ctr. for Reliable and High-Performance Computing, Univ. of Illinois, Urbana, IL, May 1990.
5. W.-H. Wang and J.-L. Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis," *ACM Trans. Computing. Sys.*, vol. 9, pp. 222-241, Aug. 1991.
6. K. R. Kaplan and R. O. Winder, "Cache-Based Computer Systems," *Computer*, vol. 6, pp. 3~36, Mar. 1973.
7. W. D. Strecker, "Cache Memories for (PDP-II) Family Computers," *Proc. 3rd Ann. Int'l Symp. Computer Architecture*, 1976, pp. 155-158.
8. G. S. Shedler and D. R. Slutz, "Derivation of Miss Ratios for Merged Access Streams," *IBM J. Res. Develop.*, vol. 20, pp. 505-517, Sept. 1976.
9. M C. Easton, "Computation of Cold-Start Miss Ratios," *IEEE Trans. Computers*, vol. C-27, pp. 404-408, May 1978.
10. A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3. pp. 473-530, 1982.
11. D. W. Clark, "Cache Performance in the VAX-11/780," *ACM Trans. Computing Systems*, vol. I, pp. 24-37 Feb. 1983.
12. J. Haikala, "Cache Hit Ratios with Geometric Task Switch Intervals," *Proc. 11th Ann. Int'l Symp. Computer Architecture*, 1984, pp. 364-371.
13. A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating System and Multiprogramming Workloads," *ACM Trans. Computing Systems*, vol. 6, pp. 393-431, Nov. 1988.
14. D. Thiebaut and H. S. Stone, "Footprints in the Cache," *ACM Trans. Computing Systems*, vol. 5, pp. 305-329, Nov. 1987.
15. J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," *Proc. 4th Int'l Conf Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 75-84.
16. T. M. Conte, "Systematic Computer Architecture Prototyping," Ph.D. thesis, Dept. of Elect. and Comput. Eng., Univ. of Illinois, Urbana, IL, 1992.
17. T. M. Conte and W. W. Hwu, "Benchmark Characterization," *Computer*, pp. 48-56, Jan. 1991
18. "Spec Newsletter," SPEC, Fremont, CA, Feb. 1989.
19. J. S. Emer and D. W. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proc. 1lth Ann. Int'l Symp. Computer Architecture*, 1984, pp. 301-309.
20. D. W. Clark, P. J. Bannon, and J. B. Keller, "Measuring VAX 8800 Performance with a Histogram Hardware Monitor," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, 1988, pp. 176-185.

Wen-mei W. Hwu (S'81-M'87) received the Ph.D. degree in computer science from the University of California, Berkeley, in 1987.

He is an Associate Professor at the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. His research interest is in the area of architecture, implementation, and compilation for high performance computer systems He is the director of the IMPACT project, which has delivered new compiler and computer architecture technologies to the computer industry since 1987. The IMPACT project has been sponsored by NSF, ONR, NASA as well as major corporations such as Hewlett-Packard, Intel, SUN, NCR, AMD, and Matsushita.

In recognition of his contributions to the areas of compiler optimization and computer architecture, the Intel Corporation named Dr. Hwu the Intel Associate Professor at the College of Engineering, University of Illinois in 1992. He received the National Eta Kappa Nu Outstanding Young Electrical Engineer Award for 1993 and the 1994 Senior Xerox Award for Faculty Research.

Thomas M. Conte received the BSEE degree from the University of Delaware, Newark and the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois, Urbana-Champaign.

He is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of South Carolina, Columbia, South Carolina. His interests are in computer architecture and the history of computing.

Dr. Conte is a member of the EKE Computer Society, the ACM, Tau Beta Pi, and Eta Kappa Nu.

Originally published in *IEEE Trans. Computers*, Sept. 1994, pp. 994–1003.