May 1990 UILU-ENG-90-2214
CSG-122

2

COORDINATED SCIENCE LABORATORY
College of Engineering

SINGLE-PASS MEMORY SYSTEM EVALUATION FOR MULTIPROGRAMMING WORKLOADS

Thomas M. Conte
Wen-mei W. Hwu

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): UILU-ENG-90-2214 CSG-122	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: NSF, NCR, NASA, ONR
6c. ADDRESS (City, State, and ZIP Code): 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code): NSF:1800 G Street, Washington, DC 20552 NCR:Personal Computer Div.-Clemson 1150 Anderson Dr., Liberty, SC 29657
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: same as 7a. / 8b. OFFICE SYMBOL (If applicable): N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: NSF:MIP-8809478 NCR:1-6-41546, NASA: 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)

Single-Pass Memory System Evaluation For Multiprogramming Workloads

12. PERSONAL AUTHOR(S)
Conte, Thomas M. Hwu, Wen-mei W.

13a. TYPE OF REPORT: Technical	13b. TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Year, Month, Day): 1990 May	15. PAGE COUNT: 22

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, cache performance, stack-based method multiprogramming

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

Modern memory systems are composed of levels of cache memories, a virtual memory system, and a backing store. Varying more than a few design parameters and measuring the performance of such systems has traditionally be constrained by the high cost of simulation. Models of cache performance recently introduced reduce the cost simulation but at the expense of accuracy of performance prediction. Stack-based methods predict performance accurately using one pass over the trace for all cache sizes, but these techniques have been limited to fully-associative organizations. This paper presents a stack-based method of evaluating the performance of cache memories using a recurrence/conflict model for the miss ratio. Unlike previous work, the performance of realistic cache designs, such as direct-mapped caches, are predicted by the method. The method also includes a new approach to the problem of the effects of multiprogramming. This new technique separates the characteristics of the individual program from that of the workload. The recurrence/conflict method is shown to be practical, general, and powerful by comparing its performance to that of a popular traditiona cache simulator. The authors expect that the availability of such a tool will have a large

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: 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. NASA Langley Research Center, Hampton, VA 23665
Office of Naval Research, 800 N. Quincy, Arlington, VA 22217

19. impact on future architectural studies of memory systems. (KR)

Single-Pass Memory System Evaluation For Multiprogramming Workloads

Thomas M. Conte Wen-mei W. Hwu

Center for Reliable and High-Performance Computing

University of Illinois

hwu@csg.uiuc.edu

Single-Pass Memory System Evaluation For Multiprogramming Workloads

Abstract

Modern memory systems are composed of levels of cache memories, a virtual memory system, and a backing store. Varying more than a few design parameters and measuring the performance of such systems has traditionally be constrained by the high cost of simulation. Models of cache performance recently introduced reduce the cost simulation but at the expense of accuracy of performance prediction. Stack-based methods predict performance accurately using one pass over the trace for all cache sizes, but these techniques have been limited to fully-associative organizations. This paper presents a stack-based method of evaluating the performance of cache memories using a recurrence/conflict model for the miss ratio. Unlike previous work, the performance of realistic cache designs, such as direct-mapped caches, are predicted by the method. The method also includes a new approach to the problem of the effects of multiprogramming. This new technique separates the characteristics of the individual program from that of the workload. The recurrence/conflict method is shown to be practical, general, and powerful by comparing its performance to that of a popular traditional cache simulator. The authors expect that the availability of such a tool will have a large impact on future architectural studies of memory systems.

1 Introduction

Because of the role they play in the design of cost-effective memory systems, cache memories have occupied a special place in research into computer architecture. In 1986, Smith compiled a bibliography of 380 papers on the topic covering fifteen years of research [1]. A majority of these papers have focused on the performance evaluation for the design of cache memories. Some papers have evaluated cache performance as compared to other alternatives [2, 3]. In either case, cache design and evaluation is largely an empirical procedure. A benchmark set is selected and it is used to evaluate the cache performance for a design space. The performance evaluation technique of preference has been simulation. However, simulation is costly, limiting the design space and the number of benchmarks the designer can consider. Two directions have been undertaken in the literature into alternate cache performance evaluation methods. An analytical approach was introduced by Denning in [4]. A recent example of an analytical approach is presented by Agarwal, *et. al.* [5] and has been used for design space exploration by Przybylski *et. al.* [6].

An alternative to an analytical model is a hybrid approach described collectively one-pass or stack algorithms. These were introduced by Mattson *et. al.* in [7]. They function by exploiting properties of stacking replacement policies to evaluate all fully-associative cache sizes in one pass over the trace. A recent example of the evolution of these ideas is in Thompson and Smith [8], where one-pass algorithms are presented for fully-associative buffers for realistic policy decisions such as write back and sector mapping. Traiger and Slutz [9] present a method that addresses various levels of set associativities and block sizes in one pass, but the amount of collected information required to reconstruct the cache performance is large. Due to this large storage requirement, their technique is impractical.

Even when using the trusted simulation techniques for evaluation of cache memories, the issue of approximating operating system effects is troublesome. Multiprogramming has the effect of partially- or completely flushing a buffer at arbitrary instances during execution. One approach to this problem used in [10] was to systematically flush the cache at fixed intervals. Using this technique, the designer can randomly insert context switches into a simulation, but to get stable results requires increasing significantly the number of simulations. Also, it has been shown that assuming fixed context switching intervals is overly optimistic [11, 12]. Another approach is to estimate the cache performance using cold-start miss ratios, but an assumption of no-saved context after a context switch has been shown to not be true for large caches [12, 10]. Combining several reference streams into a stochastically merged stream solves this problem, but at the cost introducing workload choice (e.g., sets of benchmarks) into the evaluation problem [13]. Lastly, none of these approaches consider separately the voluntary context switching that occurs when a program makes a request of the operating system.

This paper presents the recurrence/conflict method of evaluating the performance for fully-associative, set-associative and direct mapped cache organizations for all cache and block sizes exactly using one pass over the trace. This method is based on the work of Mattson *et. al.*, Traiger and Slutz, and Thompson and Smith, but alters the statistics collected to accommodate a new model for miss ratio calculation [7, 8, 9]. This model for the miss ratio reduces substantially the traditional storage requirements of the collected information, making the method practical. A portion of the information collected can be used to reconstruct multiprogramming effects due to both voluntary and involuntary (preemptive) context switching. Preemption frequency and partial flushes of the buffer are parameterized to separate workload considerations from benchmark considerations. To evaluate the method's practicality, the run time of its implementation is compared against a popular traditional cache simulator. Results from a set of benchmarks are presented to demonstrate the method's operation.

2 A Method for Memory System Evaluation

A cache memory is a familiar concept. The dimension of a cache can be expressed as a three-tuple, (C, B, S), for a cache of size 2^C bytes, with block size 2^B blocks, and 2^S blocks for each associativity set Note that $C \geq B + S$. For example, a cache of dimension $(10, 6, 1)$ is a 1KB direct-mapped cache with a block size of 64 bytes. A cache of dimension $(21, 10, 11)$ is of size 2MB with 1KB-length blocks and it is fully-associative (such a cache models a modern virtual memory system). The notation (C, B, ∞) is an abbreviation for fully-associative caches $(S = C - B)$.

Common metrics of cache performance are miss ratios and traffic ratios. One method of calculating the miss ratio, ρ, is to count the number of instances that a miss occurred in N references. This number is the *miss count*, M, and the miss ratio is then,

$$\rho = \frac{M}{N}, \tag{1}$$

The *traffic ratio*, σ, a measure of the traffic on the memory bus generated by the cache, can be expressed as $\sigma = 2^B \rho$.

2.1 The recurrence/conflict model

Because the traffic ratio is derived from the miss ratio and the block size, the miss ratio suffices to characterize the performance of a cache memory. One method of calculating ρ is to use Equation 1. Another method of calculation is based on the observation that all hits occur due to recurring references. For example, consider the following string of references:

Reference number	1	2	3	4	5	6	7	8
Address	100	101	102	103	102	101	104	101

References to addresses 100, 103, and 104 occur only once and result in a miss regardless of the cache organization. References to 101 and 102 occur more than once and hence have potential for a hit, dependent upon the cache organization. Such references are called *recurring references*, and there are three of them in the example (references 5, 6, and 8). The references between recurring references are termed the *intervening references*. For example, references 3, 4 and 5 are the intervening references between the first recurring reference to address 101. There is a chance that for a given cache organization the intervening references will remove the recurring reference from the cache, resulting in a miss instead of a hit. Such a situation is termed a *conflict.* If there are R instances of recurring references and K instances of conflicts, the miss ratio can be expressed as,

$$\rho = 1 - \frac{R-K}{N}. \tag{2}$$

This expression is termed the recurrence/conflict model for the miss ratio.

The method of calculating the miss ratio for a large class of cache organizations accurately is based on the recurrence/conflict model. The method involves the calculation of two arrays, $r[B]$ and $\kappa[C, B, S]$, using one pass over the reference string. An algorithm to perform these computations is presented in Figures 1 and 2. The procedure, `recurrence_conflict(`a`)`, is applied in-turn to each referenced address. The array **stack**$[B]$ is an array of stacks managed by the routines **push**$(\cdot)$, **topofstack**$(\cdot)$, **depth**$(\cdot)$, and **repush**$(\cdot)$. The items kept on a stack are addresses. Note the two functions, **zero_out_lsb**$(\cdot)$ and **count_trailing_zeros**$(\cdot)$ The function **zero_out_lsb**(a, B) returns a with B least significant bits set to zero (i.e., the block address of address a). The function **count_trailing_zeros**$(\cdot)$ returns the number of trailing zeros in the binary representation of a number. The algorithm explores a predefined maximum design space, delimited by the parameters $C_{\mathbf{max}}$, $B_{\mathbf{max}}$, and $S_{\mathbf{max}}$. Also, a special column is maintained in $\kappa[C][B][S]$ for conflicts in fully-associative caches, $\kappa[C, B, \infty]$. Note that any conflict that occurs in a cache of size (C, B, S), also occurs in a cache of size $(C-1, B, S)$. The **while** loop of procedure **process_cycle** (Figure 2) implements this observation. The miss ratio for a cache of dimension (c, b, s) can be calculated from $r[B]$ and

```
recurrence_conflict($a$, voluntary_cs):
begin
    $N \leftarrow N+1$
    for $B \leftarrow 0$ to $B_{\max}$ do
    begin
        block_addr←zero_out_lsb($a, B$)
        if on_stack(stack[$B$], block_addr) then
            $d$←depth(stack[B],block_addr)
            process_cycle($B$, $d$, block_addr)
            repush(stack[$B$], block_addr)
        else
            push(stack[$B$], block_addr)
        end
        cs_count(block_addr) ← 1
        unmark_voluntary_cs(block_addr)
        if voluntary_cs then mark_voluntary_cs(block_addr)
    end
end
```

Figure 1: Driver routine for the recurrence/conflict algorithm.

$\kappa[C][B][S]$ using Equation 3.

$$\rho(c,b,s) = 1 - \frac{1}{N}\left(r[b] - \sum_{j=c}^{C_{\max}} \kappa[j,b,s]\right). \tag{3}$$

Since the recurrence/conflict algorithm is based on the LRU stack algorithm presented by Mattson, *et. al.* in [7], it is of complexity $O(N \lg N)$ on average. The space storage required for the resulting $r[B]$ and $\kappa[C,B,S]$ is $B_{\max} + (C_{\max}+1) \times (B_{\max}+1) \times (S_{\max}+2) + 1$ words (typically, a C language long). Also, two arrays, $M_I[C][B][S]$ and $M_V[C][B][S]$ are calculated to account for multiprogramming (see Section 2.2 below). Hence, for a design space of size $C_{\max} = 31$ (2GB), $B_{\max} = 12$ (4KB), and $S_{\max} = 3$ (up to 8 ways, and fully associativity), the space storage requirements are approximately 6K words. This allows the statistics to be readily stored as a disk file. Since the arrays tend to be sparse, the disk file is smaller than this upper bound, in practice.

```
process_cycle($B$, $d$, block_addr):
begin
    $r[B] \leftarrow r[B] + 1$
    if top_of_stack(stack[$B$]) = block_addr then return
    Let $a \in$ stack[$B$] and depth(stack[$B$], $a$) $= d - 1$
    cs_count(a) ← cs_count(a)+cs_count(block_addr)
    if marked_voluntary_cs(block_addr) then mark_voluntary_cs(a)
    num_unique ← 0
    cs_points ← 0
    voluntary_cs ← false
    for $a \in$ stack[$B$] and depth(stack[$B$], $a$) $< d$ do
    begin
        dist←count_trailing_zeros(|$a$ − block_addr|)
        $p$[dist]←$p$[dist] + 1
        max_dist ← max(max_dist, dist)
        num_unique ← num_unique + 1
        if marked_voluntary_cs(block_addr) then voluntary_cs ← true
        cs_points ← cs_points + cs_count(a)
    end
    FA_cachesize ← ⌊lg num_unique⌋
    if FA_cachesize > $C_{max}$ then FA_cachesize ← $C_{max}$
    $\kappa$[FA_cachesize, $B$, $\infty$] ← $\kappa$[FA_cachesize, $B$, $\infty$] + 1
    $M_I$[FA_cachesize][$B$][$S$]←$M_I$[FA_cachesize][$L$][$S$] + cs_points
    dist←max_dist
    $i \leftarrow 1$
    sum←0
    for $S \leftarrow 0$ to $S_{max}$ do
    begin
        while dist $\geq 0$ and sum $< i$ do
        begin
            sum←sum + $p$[dist]
            dist←dist − 1
        end
        $C_{min,conf} \leftarrow 0$
        if sum $\geq i$ then
            $C_{min,conf}$←dist + $S$ + 1
            $\kappa[C_{min,conf}][B][S] \leftarrow \kappa[C_{min,conf}][B][S] + 1$
        end
        $M_I[C_{min,conf}][B][S] \leftarrow M_I[C_{min,conf}][B][S]$ + cs_points
        if voluntary_cs then $M_V[C_{min,conf}][B][S] \leftarrow M_V[C_{min,conf}][B][S] + 1$
        $i \leftarrow 2 \times i$
    end
end
```

Figure 2: The process_cycle procedure to calculate $r[B]$ and $\kappa[C, B, S]$.

2.2 Multiprogramming effects

Estimating the effects of multiprogramming on cache performance is a well-known problem [10]. Several techniques have been employed to approximate these effects. The cold miss ratio vs. warm miss ratio technique was examined by Easton in [12, 14]. Examples of statistical approaches can be found in [11, 13]. This paper presents a method based on the recurrence/conflict model. Multiprogramming is divided into two categories: voluntary context switching and involuntary context switching. These categories are explained below.

Voluntary context switching

A process performs a voluntary context switch when the continuation of its execution depends on a system service which may take a long time to finish. The frequency and timing of a voluntary context switch is solely a characteristic of the benchmark. The number of processes executed before a process returns from a context switch is, however, a function of the system load and the operating system scheduling policy. For example, the working set of a process may have been purged from the cache before it re-enters the run state after a context switch. This results in a degraded cache performance as compared to an ideal execution of the same benchmark without any context switching.

There are two pieces of information that are associated with context switching. One is the number of potential *victims*, defined as the number of non-conflicting recurring references which may be converted from a hit to a miss. This information is a function of the benchmark and the cache dimension. The method presented in this paper provides this information exactly. However, the fraction of the potential victims which are actually converted to misses is a function of the system's load and the operating system's scheduling policy. Hence, this fraction is modeled as a parameter, ξ. The designer can vary the parameter value between 0% and 100% to examine the changes in design decisions based on information collected in only one pass. This feature distinguishes this method from most of the previous ones where varying this parameter requires a re-simulation.

The total number of potential victims of all voluntary context switches is measured using the recurrence/conflict method. Each voluntary context switch point is marked in the trace, and the potential victims of this context switch point are identified as those non-conflicting recurring references which occur across the context switch point. This is implemented by marking references that occur immediately before voluntary context switch points.

The array, $M_V[C][B][S]$, is used to record the number of potential victims of voluntary context switching. The method of updating $M_V[\cdot]$ is included in Figure 2. If $M_V[c][b][s]$ is equal to n at the end of the execution, it indicates that for all caches (c', b, s), $c' \geq c$, n of all the hits can be potentially converted to misses due to voluntary context switches. Given a percentage of preserved context across context switches, ξ, one can expect to find ξn of the hits to be converted to misses. The miss ratio for a cache of dimension (c, b, s) in the presence of voluntary context switching becomes,

$$\rho(c, b, s) = 1 - \frac{1}{N}\left(R[b] - \sum_{j=c}^{C_{max}} K[j][b][S] - \xi \sum_{j=0}^{c} M_V[j][b][s]\right). \qquad (4)$$

Involuntary context switch

Involuntary context switching occurs due to external events such as timer-implemented preemption and I/O device interrupts. The frequency and occurrance of involuntary context switching is a function of the system load and the operating system's scheduling policy, but not a characteristic of the program. Therefore, it is assumed that an involuntary context switch has an equal probability of occurring after any reference. With this assumption, the recurrence/conflict method derives the average number of potential victims, V_I, due to each involuntary context switch. A parameter, Q, is defined as the effective quantum (average preemption interval). Hence, N/Q is the total number of involuntary context switches expected for the entire reference string. Therefore, the total number of hits that are converted to misses is $(\xi N V_I)/Q$. Like ξ, one can vary Q over an arbitrary range to observe the impact of involuntary context switching frequency on the design decisions.

To derive the average number of potential victims due to each involuntary context switch, one can sum the number of potential victims for all possible switching points in the reference string and divide this sum by the number of possible switch points (N). This is given in the following formula,

$$V_I = \frac{1}{N}\left(\Sigma_{(\text{all switching points})}\text{Number of potential victims for a switch point}\right). \qquad (5)$$

By exchanging the roles of the context switches and the potential victims, Equation 5 can be rewritten in the following form,

$$V_I = \frac{1}{N}\left(\Sigma_{(\text{all potential victims})}\text{Number of switching points affecting potential victim}\right). \qquad (6)$$

Equation 6 fits naturally into the recurrence/conflict method.

Due to the large number of context switching points involved, a counter, cs_count($\cdot$), is kept for each element on the stack. Each time a new stack element is created, this counter is set to 1. When a recurring reference is processed, the context switching count of its stack element is accumulated into that of the element above it before it is promoted to the top of the stack. In this way, all the references originally below the element will see the same number of context switching points above them. The context switching count of the promoted element then is reset to one. (See Figures 1 and 2.)

The array, $M_I[C][B][S]$, is used to record the total number of potential victims of all involuntary context switching points. If $M_I[c][b][s]$ is equal to n at the end of the execution, it indicates that for all caches (c', b, s), $c' \geq c$, n of all the hits will potentially be converted to misses due involuntary context switching. The average number of potential victims per involuntary context switch for a cache of configuration (c, b, s) is,

$$V_I = \frac{1}{N} \sum_{j=0}^{c} M_I[j][b][s]. \tag{7}$$

The miss ratio for a cache of configuration (c, b, s) under multiprogramming is expressed in Equation 8.

$$\rho(c, b, s) = 1 - \frac{1}{N}\left(R[b] - \sum_{j=c}^{C_{\max}} K[j][b][S] - \sum_{j=0}^{c} M_V[j][b][s] - \frac{\xi N V_I}{Q} \right). \tag{8}$$

2.3 Trace collection

A method of collecting the trace of a benchmark program is to annotate executable with special probe instructions. As these probes are executed, local-scope dynamic behavior is recorded. Such a method is termed, ***keyhole experimentation***, to emphasize that it is a dynamic dual of retargetable "peephole" techniques used for local optimization [15]. Keyhole experimentation has been used to generate profiles of the programs' behavior, although the potential for more than just profile information gathering exists. The keyhole probes can be placed by the compiler (e.g., GPROF [16]), the assembler (e.g., TRAPEDS [17]), or a separate object-code modifier (e.g., PIXIE [18]). Using keyhole experimentation at the compiler level is of greatest use to architects, since the compiler possesses information about the program's data and instruction structure before optimization. The Architects Workbench (CARA), created by Flynn at Stanford [19], is one such tool. The *System Parameter Independent Keyhole Experimenter* (SPIKE) is a compiler-independent tool similar to CARA, constructed by the authors. The current version of SPIKE has been fitted into the GNU CC compiler [20], since GNU CC is capable of producing code for a variety of architectures.

3 Experimental results

The success of a cache performance evaluation method depends on its practicality. To

Table 1: The benchmark set.

Benchmark	No. references	Description
grep	4.1M	The grep program from Unix, used for a search through `/usr/dict/words`
tex	2.7M	The TeX typesetter, using the 'TripTeX' diagnostic input
yacc	722K	The LALR(1) parser-generator from Unix, with the grammar from *make)* as input

investigate the practicality of the recurrence/conflict method, a set of benchmark programs was compiled for the MC68020 and their instruction reference behavior was instrumented using SPIKE. The benchmarks are summarized in Table 1.

The run time for the recurrence/conflict method was compared against Dinero III, a reliable public-domain cache simulator constructed by Mark Hill. These results are presented in Table 2. The minutes of (user-mode) run time were collected for each benchmark using an unloaded Sun 3/280. Note that although *tex* had approximately 1.2M less references than *grep*, it took longer to run. This is due to the nature of stack algorithms: the less locality present in a program, the larger the average stack depth. The worst slowdown was

Table 2: Running time versus Dinero III.

Benchmark	Time			Average ratio RCM/Dinero
	Recurrence/ Conflict model	Dinero III		
		$(21,4,\infty)$	$(21,4,0)$	
grep	1:38	0:21	0:20	4.8
tex	4:35	0:17	0:16	16
yacc	0:53	0:03	0:03	18

by a factor of 18. However, given that the design space explored contained approximately $31 \times 10 \times 5 = 1500$ cache dimensions for each level in the memory system's hierarchy, the recurrence/conflict model has a great advantage over conventional simulation.

Since *tex* had the most interesting locality, results of the miss ratio for *tex* are presented in Figure 3. Set associativity is represented as a solid line for $S = 0$, a dotted line for $S = 2$, and a dashed line for $S = \infty$. (Because of its high performance, $S = \infty$ is only visible in the graph of $B = 3$.) After the execution of the recurrence/conflict method, the time required to generate the entire set of miss ratios for Figure 3 was under a second of user time. As many points as feasible were checked using Dinero, and all agreed with 100% accuracy.

To see the effects of multiprogramming, *tex* was evaluated assuming $B = 3$, for $\xi = 100\%, 90\%, 80\%$ and $Q = 100, 1000$. The results are presented in Figure 4. Unfortunately, voluntary context switching information is not yet available in SPIKE at the time of this writing. Therefore, only the results of involuntary context switching were evaluated. The results are presented as the difference between miss ratios of uniprogramming and of multiprogramming ($\Delta\rho$). Note that the preemption interval dominated for $Q = 1000$, whereas the percentage of flushed context (ξ) had a large effect for $Q = 100$. This implies that, for *tex*, beyond a certain Q saved context has little bearing on instruction cache performance.

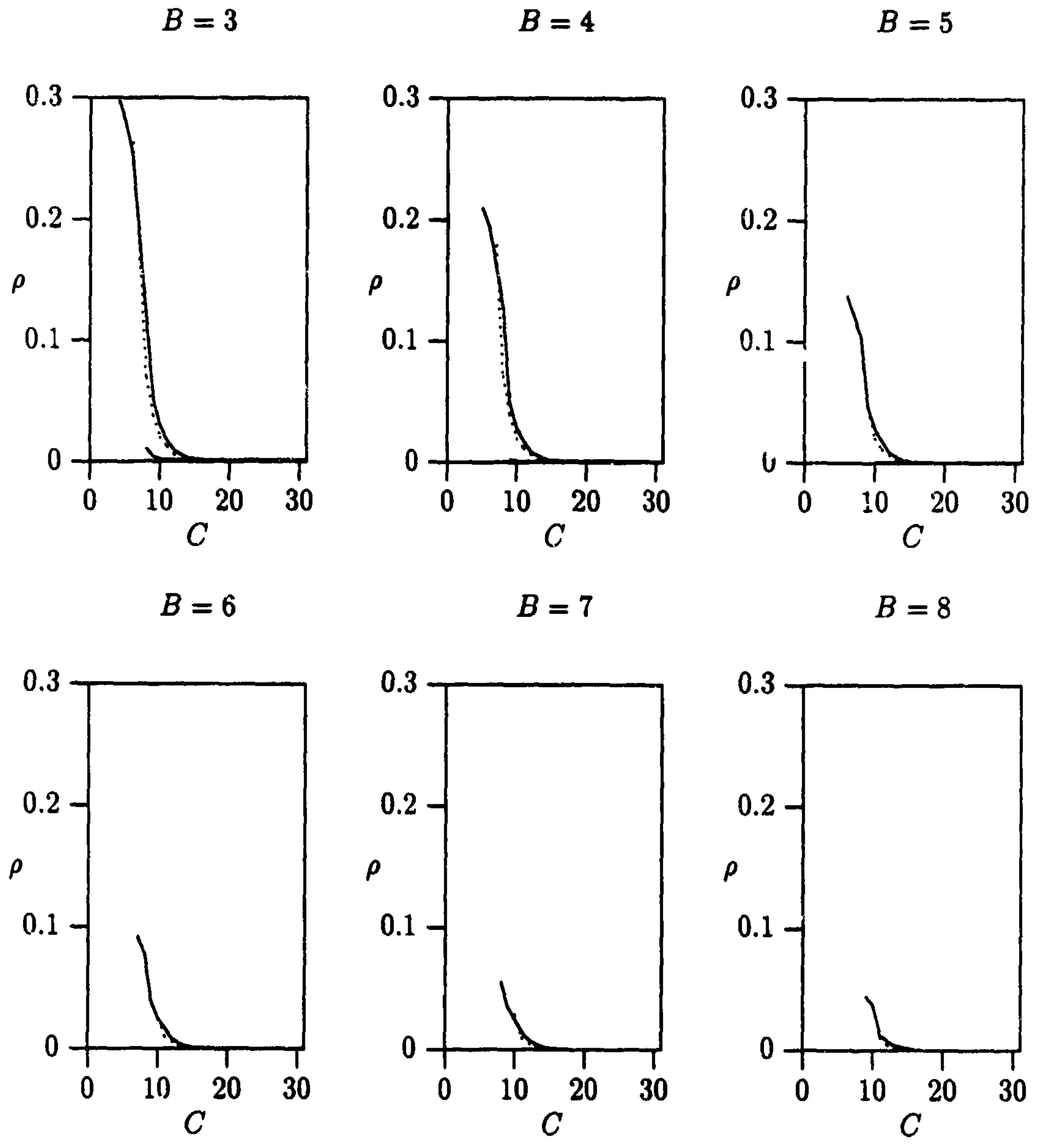

Figure 3: Miss ratios for *tex* for various cache dimensions.

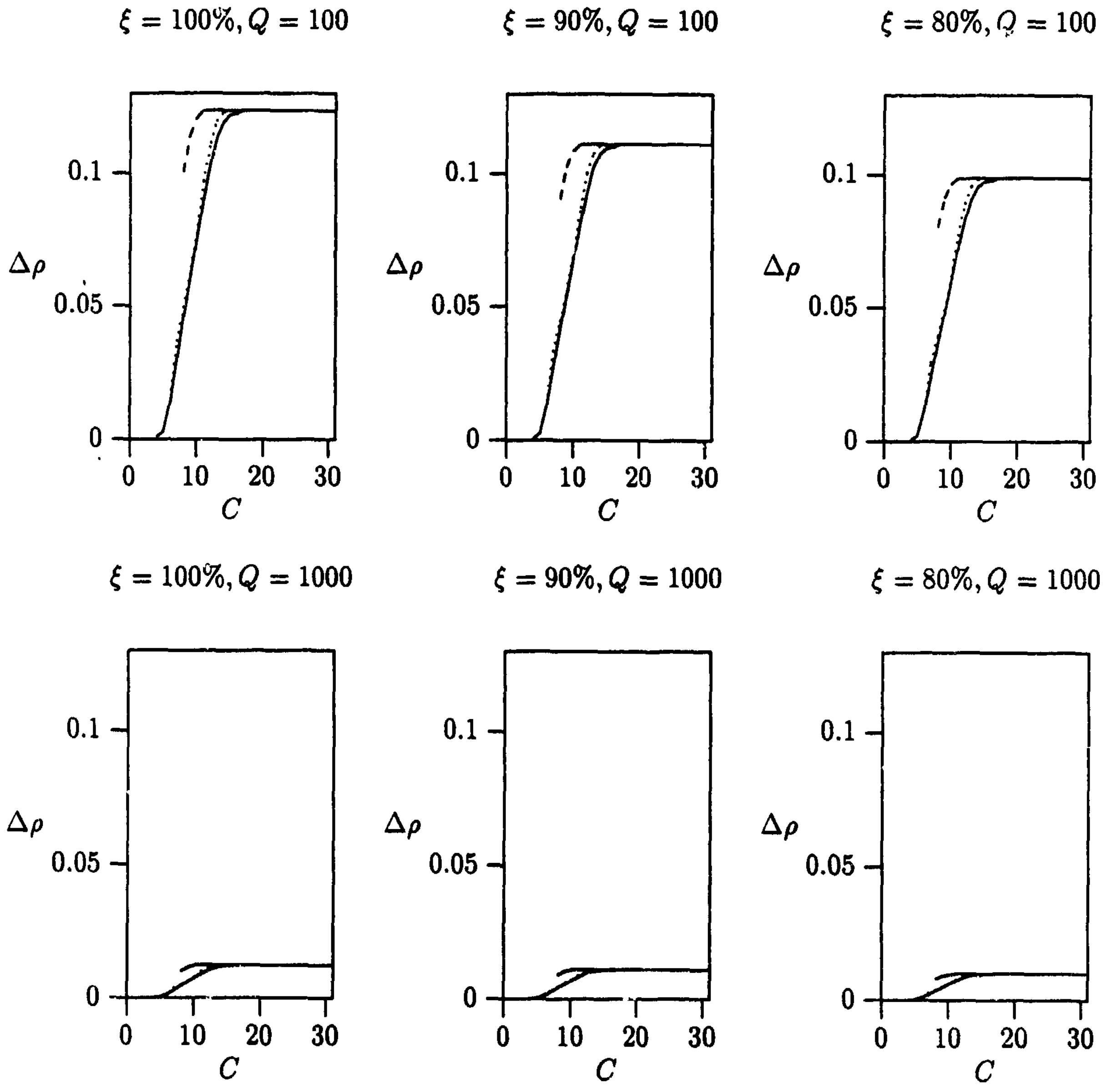

Figure 4: Multiprogramming miss ratios for *tex*.

4 Conclusions

This paper has presented a method to evaluate efficiently a very large design space for cache memories. When used to evaluate cache hierarchies satisfying the inclusion property (see [21]), an entire memory system can be evaluated in one pass. Although the algorithm presented omitted the issues of write-back and sector-mapping for brevity, the stack algorithm extensions of Thompson and Smith are compatible with the recurrence/conflict method [8]. Hence, the method is general. The method was shown to be efficient and hence profitable to use.

The recurrence/conflict method is applicable to both design and architectural research. Combined with design criteria such as described in [6], there is the potential of an automated memory system design process. Since it evaluates a large memory system design space in one pass, techniques for architectural studies into other interacting system tradeoffs can be simplified and broadened in scope. Hence, there are a large number of future research directions possible using the recurrence/conflict method.

The inclusion of context switching effects into the method is an advance of previous work as it cleanly seperates the behavior of the benchmarks from the multiprogrammed performance characteristics they exhibit. Previous approaches involved measuring snapshots of actual multiprogramming and using these traces for cache simulation. Such approaches are restricted to phenomenological conclusions since the mix of executing processes and the interprocess timings are not adjustable after measurement. This illustrates a powerful feature of the recurrence/conflict model of the miss ratio: external effects that degrade cache performance, such as context switching or coherence protocol invalidations, can be modeled as additional types of conficts, thereby isolating the performance of different design tradeoffs.

[For interested parties, a stable version of the tool written in portable C is freely available from the authors.]

Addendum

This report was presented for review to the International Symposium on Computer Architecture Program Committee in November of 1989. In December of 1989, Mark Hill and Alan Smith published an article in *IEEE Transactions on Computers*, entitled, "Evaluating associativity in CPU caches" (see [22]). Although Hill and Smith did not make the distinction between recurrences and conflicts, the presented algorithm is similar to the RCM method. Since it is common in Science for two distinct research groups to discover an idea, and common also for each group to have different insight, this report is being made available to present our insights into stack-based memory hierarchy analysis. The material in this report discussing evaluation of multiprogramming effects (context switching) is our own and not present in the Hill and Smith paper.

- T. M. Conte and W. W. Hwu, March, 1990

Acknowledgements

The authors would like to thank Sadun Anik, David Griffith and 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, a donation from NCR, 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.

References

[1] A. J. Smith, "Bibliography of readings on CPU cache memories and related topics," *Comput. Architecture News*, vol. 14, pp. 22–42, Jan. 1986.

[2] J. R. Goodman and W.-C. Hsu, "On the use of registers vs. cache to minimize memory traffic," in *Proc 13th Annu. Int'l Symp. on Comput. Arch.*, pp. 375–383, Jan. 1986.

[3] R. J. Eickenmeyer and J. H. Patel, "Performance evaluation of on-chip register and cache organizations," in *Proc. 15th Annu. Int'l Symp. on Comput. Arch.*, (Honolulu, Hawaii), pp. 64–72, May 1988.

[4] P. J. Denning and S. C. Schwartz, "Properties of the working-set model," *Communications ACM*, vol. 15, pp. 191–198, Mar. 1972.

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

[6] S. Przybylski. M. Horowitz, and J. Hennessy, "Characteristics of performance-optimal multi-level cache hierarchies," in *Proc. 16th Annu. Int'l Symp. on Comput. Arch.*, (Jerusalem, Israel), pp. 114–121, June 1989.

[7] 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.

[8] J. G. Thompson and A. J. Smith, "Efficient (stack) algorithms for analysis of write-back and sector memories," *ACM Trans. Computer Systems*, vol. 7, pp. 78–117, Feb. 1989.

[9] 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.

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

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

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

[13] 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.

[14] M. C. Easton and R. Fagin, "Cold-start vs. warm-start miss ratios," *Communications ACM*, vol. 21, pp. 866–872, Oct. 1978.

[15] J. A. Davidson and C. W. Fraser, "The design and application of a retargetable peephole optimizer," *ACM Trans. Prog. Lang. and Systems*, vol. 2, pp. 191–202, Apr. 1980.

[16] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution pr[illegible]r," in *Proc. 1982 SIGPLAN Symp. on Compiler Construction*, pp. 120–126, June 1982.

[17] C. B. Stunkel and W. K. Fuchs, "TRAPEDS: producing traces for multicomputers via execution driven simulation," in *Proc. ACM SIGMETRICS '89 and PERFORMANCE '89 Int'l Conf. on Measurement and Modeling of Comput. Sys.*, (Berkeley, CA), pp. 70–78, May 1989.

[18] MIPS Computer Systems, *MIPS language programmer's guide*, 1986.

[19] C. L. Mitchell and M. J. Flynn, "A workbench for computer architects," *Design & Test*, pp. 19–29, Feb. 88.

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

[21] J.-L. Baer and W.-H. Wang, "Architectural choices for multi-level cache hierarchies," in *Proc. 16th Int'l Conf. on Parallel Processing*, pp. 258–261, Aug. 1987.

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