Randomization and Associativity in the Design of Placement-Insensitive Caches

Michael Schlansker, Robert Shaw,
Sivaram Sivaramakrishnan
Computer Systems Laboratory
HPL-93-41
June, 1993

cache memory, associative, stride, random, hashing, set lookup, data placement

This paper presents a design for a randomized, placement-insensitive, data cache and analyzes its performance. An address stream is randomized using a hash function which selects a set in an associative cache. The manipulation of large data structures is modeled by traversal of a cyclic sweep address sequence and the miss ratio is accurately determined for this sequence. A purely analytic approach to determine cache performance is developed. Analysis predicts regions in which a placement-insensitive cache operates with very few cache misses. A pseudo-random hash function is presented and used to randomize addresses into cache sets and a counting technique is used to determine miss ratios. Finally, an actual cache is simulated allowing comparison against theory. A matrix multiply program is studied demonstrating a close relationship between analysis and at least one real application.

Internal Accession Date Only

1. Introduction

Modern RISC processors typically reference data through a first level cache. Such caches are optimized for scalar references typical in non-numeric applications. A number of simulation oriented studies of cache behavior have been performed in the literature [1] and the effects of associativity on cache performance has been explored using address traces from real programs [2] [3]. Synthetic traces have been used to model program reference behavior and to stimulate cache models [4]. We will describe a simple, worst case, synthetic address trace which is directly analyzed.

An important design methodology for caches is to use a direct mapped cache supporting concurrent address translation and cache set lookup [5]. While, this is frequently a sound design methodology, there are legitimate uses for second level caches of a radically different, large and highly associative design. In order to demonstrate the use these alternative cache structures, assume that scalar, data references are directed to a conventional first level cache while latency tolerant array references with poor temporal locality are directed to a second level cache using cache bypass instructions [6]. The usage of this second level cache (because it contains no scalar references) is not consistent with assumptions on which more conventional cache designs are based.

Supercomputer and mini-supercomputer processor designs have used latency tolerant processing techniques such as vectorization or software pipelining to gain great advantage from large high bandwidth interleaved memories. However, the supercomputer's high bandwidth highly interleaved memory is very expensive. Microcomputer chips similar in performance to supercomputers will be developed in future VLSI technology. Less expensive scientific computer systems will use large, possibly multi-ported, second level caches to mimic the capability of a supercomputer memory at a fraction of the cost.

When one designs a very large second level cache, some performance is lost due to increases in latency. These losses must be compensated for by increases in performance due to the availability of a large number of sets and a high degree of associativity. A large number of sets supports a large array of cache memory chips and provides the capacity needed to manipulate large arrays. However, if we merely increase the number of sets in a direct mapped cache, then the joint manipulation of two (or more) data arrays when poorly placed in the cache address space causes cache performance to degrade. The use of higher associativity can provide robust and efficient usage of these chips when they hold large data sets placed in a variety of structured and unstructured address patterns. This paper describes important performance characteristics of large highly associative caches.

1.1 Motivation

In set associative cache design, a set is usually selected by taking the address of the cache line containing the data modulo the number of sets in the cache. These caches are subject to rather dramatic cache breakdowns when accessed with regular array-like address sequences. Access into data sets may be regular either because they are matrix structures, or because they have been allocated in a consistent pattern as might occur through repeated calls to a memory allocation procedure requesting records of fixed size.

When caches are physically indexed, virtual to physical translation may provide some element of randomness to alleviate this problem. In some settings, processors have been designed with no virtual memory translation. In other settings processors are designed with very large page sizes in order to minimize translation cache size requirements demanded by large memories. Here, large datasets may occupy only one or a few pages and any benefit due to virtual mapping randomization will be minimal. We assume that virtual address

translation alone is not a sufficient means to alleviate cache breakdown phenomenon and will ignore virtual translation for the rest of the paper.

Prior work has studied the blocking of matrix algorithms for execution on processors with data caches [7]. A cache benefits from a large blocking factor by reducing traffic to a slower main memory. In matrix multiply, for example, blocking algorithms reduce memory traffic in linear proportion (approximately) to a blocking factor. A larger blocking factor requires a larger second-level cache but allows more efficient computation out of a slower main memory. So, for example, using a 200 by 200 block, we reduce main memory data references 200 fold. A large second level cache when used with blocking allows a high performance processor connected to slow and inexpensive main memory to achieve supercomputer levels of performance on large data sets.

Blocking cannot always be used. Many complex programs are difficult to block especially if blocking is performed only by a compiler. In some cases, while blocking is possible, a blocking algorithm is not nearly as efficient as blocking for matrix multiply. In such cases it is especially important for a cache to contain large volume of data. Performance may be satisfactory only up to the point where the data set overflows the cache. Here again, a large cache is backed by inexpensive main memory which supports swapping data objects into the cache either from a single or from multiple applications.

Cache blocking algorithms result in extreme performance variation as a function of the dimension of the manipulated matrices [7] and, certain matrix dimensions should be avoided. When inappropriate data dimensions are selected, data is entirely located within a small fraction of cache sets causing poor cache utilization, and serious performance degradation. Significant performance degradation may occur even when the matrix blocking factor is chosen as a very small fraction of the total cache size if the cache block is placed with inappropriate static dimensionality.

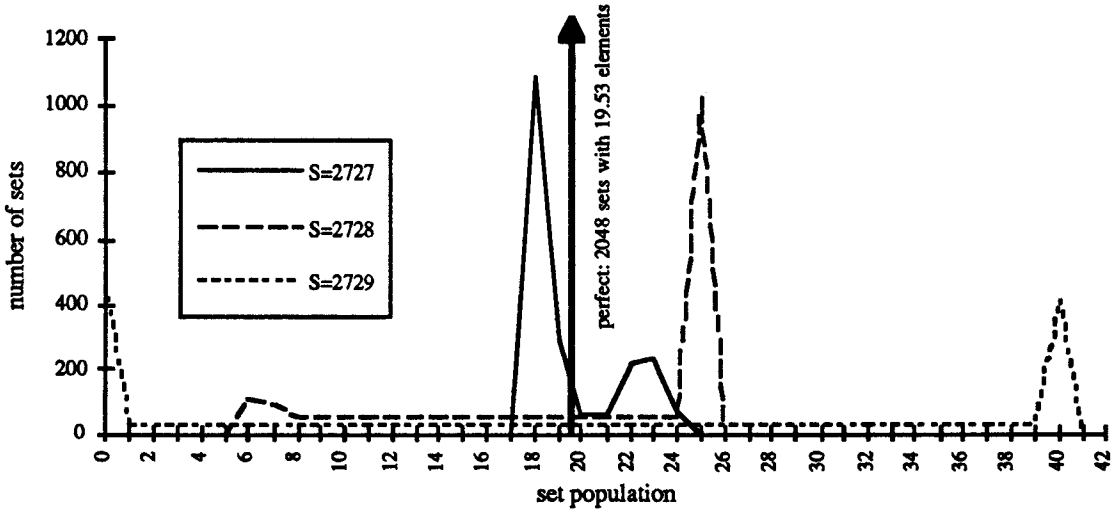

Figure 1: Set population distribution

The problem is demonstrated by traversing a 200 by 200 matrix in a 2048 set 32-way associative cache with one word per line. The experiment is performed three times using a static array dimensions equal to 2727, 2728, and 2729 respectively. A total of 40000 matrix addresses are mapped into the cache modulo the 2048 sets yielding a fixed number of data words within each set. Each stride results in a set population histogram (Figure 1) which plots, for each set population, the number of sets having that population. We see that

the set population for the static array dimension 2727, is quite centrally distributed (between 17 and 25). For this stride, no set holds more than 32 members and, the matrix can be efficiently and repeatedly accessed within the cache. A small change in static dimension to 2729, causes the set population to be more broadly distributed. Approximately 500 sets now have population 40 and over 400 sets are empty. Catastrophic cache breakdown has occurred at associativity 32.

It is often suggested that a programmer or a compiler should control data layout so as to avoid cache performance degradation. This represents an important approach, but also has significant limitations. Specific cache parameters become part of the user visible architecture introducing significant complexity into the program tuning process. Application programmers typically develop programs which are intended to run on many computer systems. Data dimensions which are selected to optimize one computer system may not be optimized for another. In some cases, especially where dynamic allocation is used, a programmers is not intimately aware of the data layout and will have difficulty in writing new or rewriting existing applications so as to avoid performance loss due to poor data layout.

One solution would be to design a compiler to ensure that the machine dependent data layout constraints are satisfied, insulating the user from these headaches. If the manipulation of large data sets were always done within contiguous memory locations, cache behaviour could be significantly improved. However, trends in computing have been in exactly in the opposite direction. Virtual memory and dynamically allocated structures have been used to simplify issues regarding the memory map for both the programmer and the compiler. It is unlikely that this trend will reverse and data will be systematically copied into contiguous address space.

In languages such as FORTRAN, the data layout within a program is visible to the programmer and data reorganization can conflict with language standard compliance. The optimization of data layout often requires analysis and optimization across multiple procedures. In the classic environment of separately compiled FORTRAN modules this joint analysis may not be possible. Even when entire applications are legitimately subject to joint analysis, the technical difficulty of adjusting the actual data layout and maintaining language standard compliance is significant. The use of pointers and dynamically allocated structures in the C language makes the problem even more difficult. These issues pose difficult application tuning and compiler optimization problems which will not be immediately solved in complex situations.

This paper investigates methods to eliminate difficult to predict and catastrophic cache breakdown. The addition of associativity alone does not solve this problem. However, we will demonstrate techniques where by using pseudo-random hashing and significant associativity, we eliminate hard to predict cache breakdown behavior. Higher associativity is the price paid to gain robust and efficient usage of the large second level cache memory.

Randomization has been traditionally used in software hashing techniques in order to rapidly index into symbols as required, for example, within a compiler [8 , 9]. Randomization has also been used to solve the problem of memory interference due to the presence of structured data access within an interleaved memory. It has been demonstrated that randomization can be used to make interleaved memories insensitive to the stride of matrix problems[10-12]. Here, we extend this work to explore the use of hash functions for the selection of cache set membership resulting in cache structures which are insensitive to data placement.

Cache memories can use hashing to provide insensitivity to data layout. This minimizes data layout responsibility for both programmer and compiler. Data layout insensitivity is achieved by hashing addresses when presented to the cache so to randomize the placement

of data across sets. The strength of this approach is that consistently high performance is obtained without highly complex software technology. The weakness lies in the complexity of the requisite hash function, the need for highly associative caches, and a margin of extra memory required to achieve adequate observed performance.

Supercomputing systems are excellent candidates for the use of randomized and highly associative caches. Traditionally supercomputing systems have suffered from very long latency access to memory. This has resulted from the use of extremely fast processor logic in conjunction with large arrays of slow speed memory chips. The time delay for the propagating of signals and cycling of ram chips has resulted in significant memory latencies whose cost was alleviated through techniques such as vectorization. If caches on extremely fast processors are to play a key role in holding large data sets, they too will experience difficulties traditional to supercomputer main memories. The additional time needed for address randomization and associative lookup will be small compared to the time needed to deliver signals and cycle ram chips in a very large second level cache array. The use of large and highly associative caches can be seen in the design of the KSR1 processor which uses a sixteen way associative 32 MB cache on each processor [13].

1.2 Overview

A placement-insensitive cache uses hashing to pseudo-randomly place data in cache sets. We develop an analysis of a placement-insensitive cache which relates cache performance to size and degree of associativity. In this analysis a cache reference pattern is modeled by placing data randomly within sets and traversing data in a reference circuit which we call cyclic sweep. Each subsequent reference is to the least recently referenced datum. Cyclic sweep can be viewed as a worst case temporal reference sequence in that it has no data locality. Cyclic sweep is not intended to be used as a representative of average cache performance. Rather, it illustrates an important component of cache behavior and helps characterize one mode for cache breakdown. An analysis for cyclic sweep is detailed below.

Section two develops an analytic model of random indexing into caches. This relates the cache miss ratio to the volume of data in the cache, the size of the cache, and the degree of associativity of the cache. Analytic models have been presented in prior work [14] but have studied different workloads. We provide a probabilistic model which provides a better understanding of the breakdown of associative caches when stressed by unstructured address sequences having poor locality. While prior work has characterized both cold and warm start cache miss characteristics [15], we focus on warm start.

Section three introduces a multiply-based hashing function and a simple mod placement function which are used to collect experimental data. Data is placed within cache sets using the selected hashing function and histogramming techniques are used to evaluate cache performance. The multiply-based hash function quite accurately obeys the prediction of the statistical model.

Section four uses a placement-insensitive cache simulation model in order to validate the design as well as analysis techniques presented within the paper. The simulation model is calibrated using a cyclic sweep address trace. A simple matrix multiply program is used to demonstrate that the model yields meaningful results on this program.

2. Analytic Treatment of Random Cache Indexing

We define the parameter **s** to be the number of sets within a cache, α to be the degree of associativity and **w** to be the number of words per line. The total number of lines which can be resident within the cache is $s*\alpha$. For the purposes of results within this paper, caches have exactly one word per line (w=1). We expect that caches will be designed with

more than one word per line and results derived here can be extended to estimate the performance of caches having a multiple word line, but this will not be discussed.

Let **d** be the number of words being accessed within the data set . We can define the fill fraction **f**=d/(s*α*w). If the size of the data set exceeds the size of the cache, the fill fraction exceeds and typically results in substantial miss penalties. The fill fraction represents a measure of the fullness of the cache with useful data. Caches which experience fewer misses at higher fill fraction are (in one important measure) better caches.

A cache set lookup function maps the address of each cache line to a set index in the range 0..(s-1). The classical cache set lookup function is: set=MOD_S(address). This function is known to perform very well for sequential data access. A contiguous vector interleaves perfectly among sets allowing for very effective cache utilization. Other regular access patterns may cause cache breakdown. A hashing function which hashes a data address to select a target set is approximated by independent uniformly distributed random trials each selecting a cache set. The performance of this cache can be analyzed using probability theory. The analysis assumes that all words within the data set are touched in turn before any word is re-touched. After all words are touched, the pattern repeats. Cyclic sweep access patterns represent important and problematic components of the actual access pattern within matrix problems.

2.1 Calculation of the Statistical Set Population Histogram

The set population histogram describes the distribution of set membership as a function of the number of sets, the degree of associativity, and the volume of data within the cache. The set population histogram determines the number of misses that the cache experiences using the cyclic sweep access pattern. In order to calculate the set population histogram, we assume that each time a word is statically placed within cache sets, a randomly selected set is chosen using a uniform distribution. A fully associative lookup of members within a selected set is performed. If a miss occurs, both LRU and random replacement strategies will be investigated.

The cache size, the fill fraction f, and the average set population λ, can be calculated:

$$\text{size}=s \times \alpha \qquad f=\frac{d}{s \times \alpha} \qquad \lambda=f \times \alpha$$

Fix on any set within the cache. On a trial mapping of an address into the cache, the probability that the line is mapped into the selected set is p and the probability that the line is mapped into another set is q:

$$p=1/s \qquad q=1-p$$

Let pm(i,s,d) be the probability that the selected set contains exactly i member lines. For i between 0 and d, a binomial density function is used to evaluate pm(i,s,d). This is evaluated as:

$$pm(i,s,d)=p^i\, q^{d-i}\, c_i^d \quad , \quad 0 \le i \le d \qquad c_i^d=\frac{d!}{i!\,(d-i)!}$$

A binomial coefficient is used to count the number of ways that exactly i of the total d cache lines may populate a given set.

The function pm(i,s,d) can be approximated by a Poisson distribution as described in [16]. This excellent approximation, even for small d, is precise in the limit as d goes to infinity.

$$pm(i,s,d) \approx pa(i,\lambda) = \frac{e^{-\lambda}\ \lambda^{i}}{i!}, \quad \lambda=\frac{d}{s}$$

The Poisson density function $pa(i,\lambda)$ depends only on i (the set population), and λ (the average number of members per set). The Poisson density function is shown in Figure 2. The parameter $\lambda=f*\alpha$ is varied in steps of $.2*\alpha$. This helps illustrate the relationship between the cache fill fraction and the cache set population distribution.

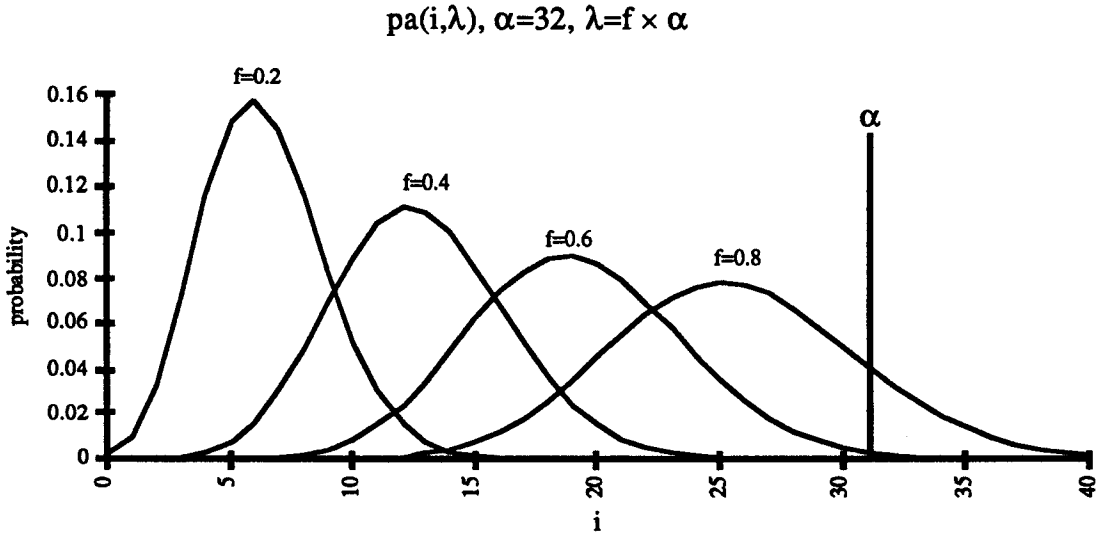

Figure 2: Probability of occupancy versus set population i

Define the normalized density function $pn(j,\lambda)$ so that caches with differing degree of associativity can be better compared:

$$pn(j,\lambda)=pa(\alpha*j,\lambda)$$

The normalized density function is shown in Figure 3. We have fixed the cache fill fraction at .7 and varied the degree of associativity from 1 to 64. The argument j of the normalized density function $pn(j,\lambda)$ represents a set fill fraction. A full set has $i=\alpha$ members in the pa distribution corresponding to j=1 members in the normalized pn distribution and a fill fraction of 1 for that set. Note that for the topmost three data plots of Figure 3, the points at which the function evaluates j corresponding to an integral number of sets are indicated by markers. These points correspond to j representing integral multiples of $1/\alpha$, the smallest unit of adding a single line to the set.

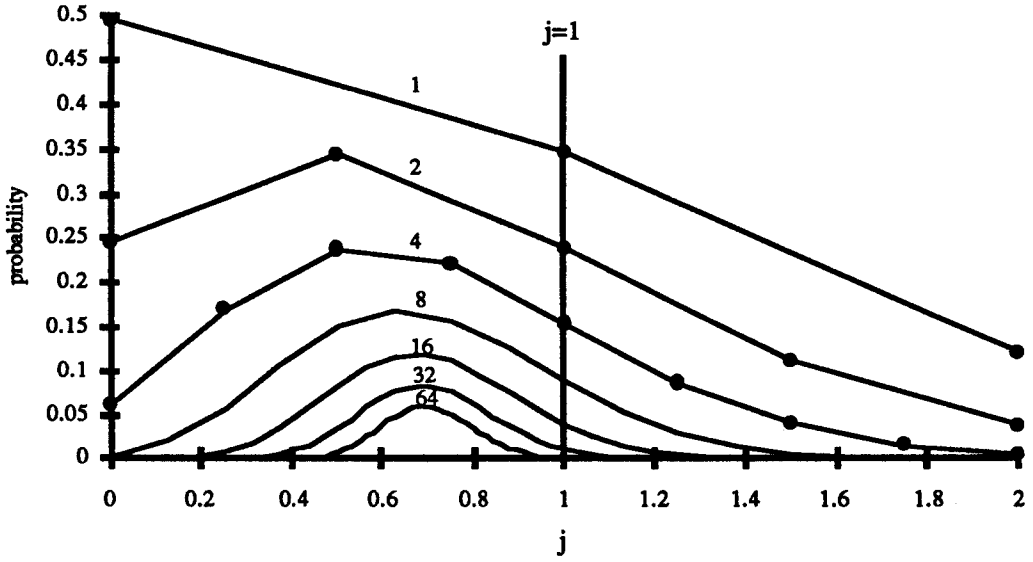

Figure 3: normalized occupancy distribution

2.2 Probabilistic Histogram-Based Miss Calculation

Given a distribution for the set population, the steady state cache miss penalty is evaluated for a cyclic pattern of data access touching data in a cycle spanning all elements. In cyclic sweep any data element is touched, then a second, then a third, each distinct from all previous until finally the first member is re-touched and the pattern repeats. Cyclic sweep is interesting in that it represents a worst case mode for cache behavior and it represents an important mode frequently visible when caches break down. Problems such as matrix multiply may traverse a full M×M result matrix , or blocked matrix multiply may traverse an M×M sub block of a larger matrix. Such problems are frequently coded so that they sweep data access in such a circuit. Of course, in real applications there is significant deviation from the perfect cyclic sweep described above.

For each set which has population count less than its degree of associativity, once data is faulted into the cache, no further misses occur and the steady state cache penalty is zero. In the normalized density plot of Figure 2, this corresponds to points to the left of j=1 (inclusive) on the horizontal axis. Conversely, points to the right of j=1 represent sets which have overflowed the associativity. Sets within this "miss region" experience an ongoing penalty whose value depends on the replacement strategy.

The fill fraction represents the ratio of the amount of data manipulated within the cache to the total cache size. As shown here:

$$f = \frac{\sum_{i=0}^{i=\infty} i\, pa(i,\lambda)}{\alpha}$$

We sum over all i the probability that a set has i elements times i to calculate a mean number of elements per set. Divided this by α, yields the fill fraction. For very low fill fraction, the area under and within the miss region of the population histogram is negligible, and so is the cache miss ratio. As the fill fraction increases, the area within the miss region increases and so does cache miss. The area within the miss region must be properly weighted to calculate the cache miss.

The normalized density distribution becomes more sharply peaked as the degree of associativity is increased. At constant fill fraction, the miss region shrinks with increased associativity and thus, high associativity allows the efficient use of nearly full randomly accessed caches. The effectiveness of hashing references into sets increases with high associativity. This narrowing of the population distribution is illustrated in Figure 3. In the extreme case, a fully associative cache has a single set with precisely d elements and no statistical deviation. This cache experiences no steady state cyclic sweep misses until the fill fraction for the entire cache exceeds one.

The LRU replacement strategy is used to exploit data references having significant temporal locality and is particularly poor for data sets having no locality. In the case of the cyclic data reference, the LRU strategy ensures that for all sets having higher population than degree of associativity, each reference to a new line in the set misses. When there are exactly $\alpha+1$ lines in an α-way associative cache set, each reference is to precisely the line which was most recently displaced, yielding full miss. In order to calculate the expected number of misses in a single sweep through the data circuit, we count for all $i>\alpha$, the probability that a set has i lines times i , and multiply this product by the total number of sets s. To calculate a miss ratio, we normalize this product by dividing by the total number of data elements in all sets ($s*\lambda$). The miss ratio can be simplified to:

$$\text{lru\_miss}(f,\alpha) = \frac{\sum_{i=\alpha+1}^{\infty} pa(i,\lambda)*i}{\lambda}, \quad \lambda = f*\alpha$$

This lru_miss function is shown in Figure 4. In order to randomly access a cache having a high fill fraction (>.5) and still maintain a low miss ratio, we must employ a large degree of associativity. High associativity such as 64-way allow efficient use of the cache up to a level of about 70% full.

The use of the set histogram to calculate cache miss ratios can be extended to other cache replacement strategies such as most recently used (MRU) and random. The analysis of random replacement is presented below. If i elements were resident within a single set, then MRU replacement would incur a penalty of $(i-\alpha)$ for each set where i is greater than α. This represents an optimal strategy for cyclic sweep. There is a problem in reaching this steady state because an MRU cache might not overcome cold start. It would continually replace recent data rather than populating the cache with newly traversed data. An initially empty cache would overcome this problem by not invoking MRU replacement until a set is full.

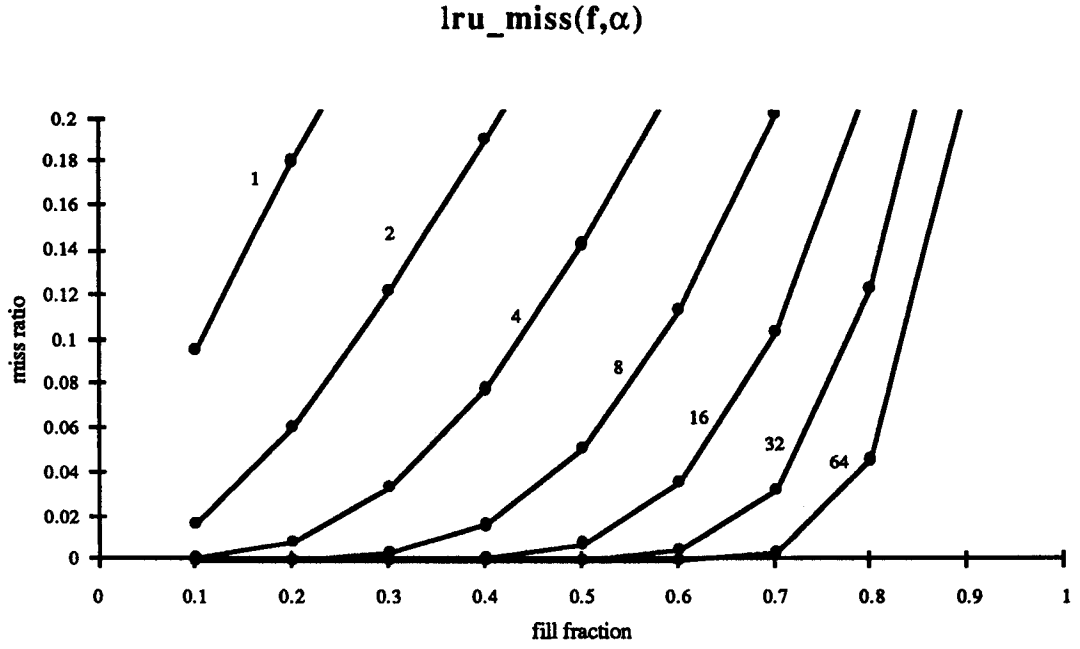

Figure 4: Miss ratio versus fill fraction and associativity for LRU replacement.

2.3 Random Replacement

Random replacement results in fewer misses than LRU replacement for cyclic sweep. Again, in the case of the random replacement, for all sets having i <= α lines, the number of steady state misses is zero. In the miss region, we weight the number of sets having population i by i*mr(α,i). We define mr(α,i) to be the steady state miss fraction observed when i data elements are cyclically referenced within a single cache set with degree of associativity α. We can calculate mr(α,i) analytically through a Markov state analysis which is informally described, or through random event simulation. Values for mr(α,i) presented in the plot of Figure 5 were obtained through simulation.

A Markov analysis for mr(α,i) defines states which uniquely identify cache set membership history. Transition probabilities connect each state to successors which assure that during a miss, a randomly selected member within the set is replaced. When a hit occurs, each state has a unique successor; when a miss occurs, each state has α successors each with transition probability $1/\alpha$ corresponding to all replacement choices. The Markov state analysis produces closed form results for mr(α,i) but has number of states which grow combinatorially. Evaluating this Markov analysis is not practical for large values of α and i. For small values of α and i, results from Markov state analysis closely match those obtained through simulation .

mr(α,i)

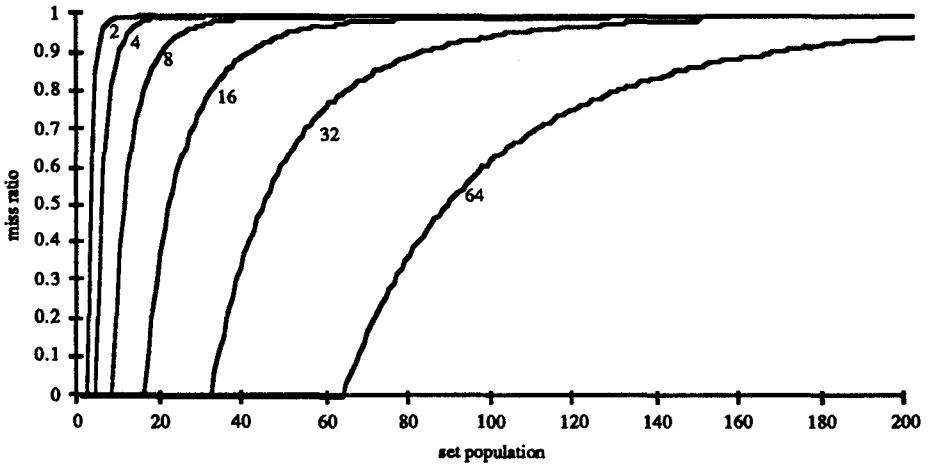

Figure 5: Single set miss ratio versus population and associativity for random replacement.

The miss ratio under random replacement is calculated much as the LRU miss ratio, but each term is weighted:

$$\text{rand\_miss}(f,\alpha) = \frac{\sum_{i=m+1}^{\infty} pa(i,\lambda)*i*mr(\alpha,i)}{\lambda}, \quad \lambda=f*\alpha$$

The rand_miss function is shown in Figure 6. Note that the primary difference between LRU and random replacement is a systematically lower miss rate for random replacement.

rand_miss(f,α)

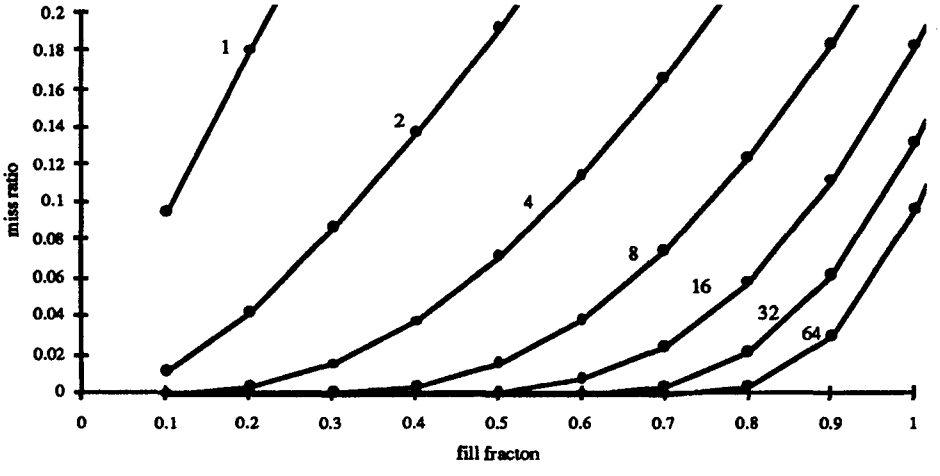

Figure 6: Miss ratio versus fill fraction and associativity for random replacement.

3. Deterministic Histogram-Based Miss Calculation

In deterministic histogram-based miss calculation, we use a synthetically generated address trace and a real hash function to distribute address sequences into sets. Each address is hashed into a target set whose count is incremented. To calculate the overall miss penalty we sum over all sets adding a miss penalty obtained using each set's exact population count. Each set is weighted according to LRU and random weighting methods described above. We present measurements on two hash functions: mod and square.

The mod function is a traditional hash which selects the remainder of the cache line address modulo the number of sets in the cache. Square represents an attempt to provide a random hash based on squaring the address and multiplying by a constant. A middle square method for hashing has been described upon which the square hash presented here is based [17, 8, 9]. A large constant serves to intermingle bits from across the width of the product. A C program provides a description of the square hash function in Table 1.

```
unsigned int  in_addr,sets, selected_set
/*
in_addr=address of cache line to be hashed
sets=number of sets
program assumes 32-bit word size
*/
selected_set=(((in_addr*in_addr*174773)>> 21)%sets);
```

Table 1 Square Hash Function

A cache is modeled having 2048 lines with 32-way associativity. We define an outer matrix dimension S representing a static FORTRAN dimension (which we informally term stride) within which array access is performed. We operate within this static array with an M by M subarray representing the actual data set. The operation performed is to sweep the subarray elements in a cyclic pattern with no locality. The actual access order makes no difference as long as all elements are traversed in a cycle.

Experiments are performed at a number of distinct fill fractions. For each fill fraction an average is taken over a number of stride experiments. Given fixed fill fraction and stride, a hash function is used to place all elements of the M by M array corresponding to the fill fraction. The resulting set population histogram determines a miss ratio. Each fill fraction averages 2048 distinct stride experiments with static outer dimension S ranging from 2000 to 4047. The use of 2048 strides integrates over a single period of a periodic miss function which is periodic in srtride.

Figure 7 illustrates miss ratio as a function of fill fraction for a 32-way associative 2048 set LRU replaced cache. Data is presented for the analytically calculated Poisson model and two hash functions: square and mod. Data is collected at matrix sizes for M ranging from 100 to 260 in steps of 10 corresponding to 17 fill fractions along a horizontal axis. Three data series corresponding to three hash functions are shown. The square hash function and Poisson distribution are indistinguishable on this plot while mod has very different characteristics. Note that at modest fill fraction, the average miss ratio of the placement-insensitive cache is significantly less than that of the conventional modulo cache because, the conventional cache is subject to periodic cache breakdowns which are summed into the average. Figure 8 repeats the experiment of Figure 7 with random replacement.

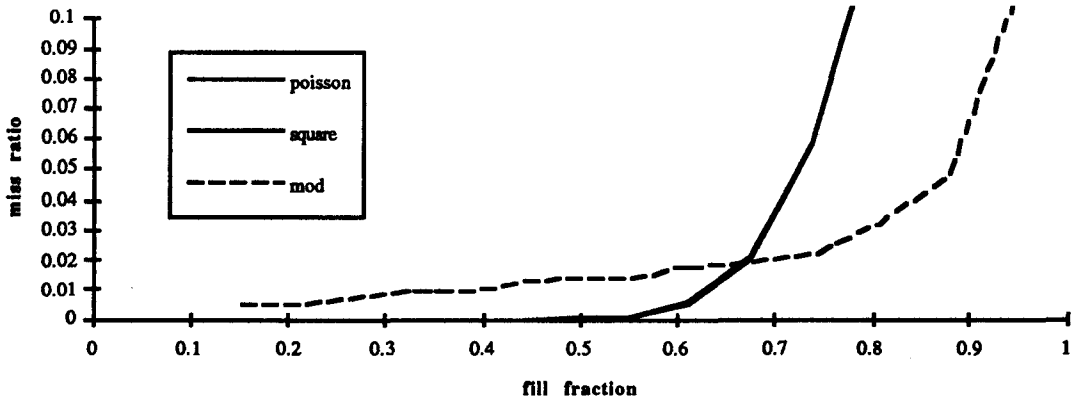

Figure 7: Miss ratio versus fill fraction f for: 32 way 2048 set LRU replaced cache; averaged over 2048 strides.

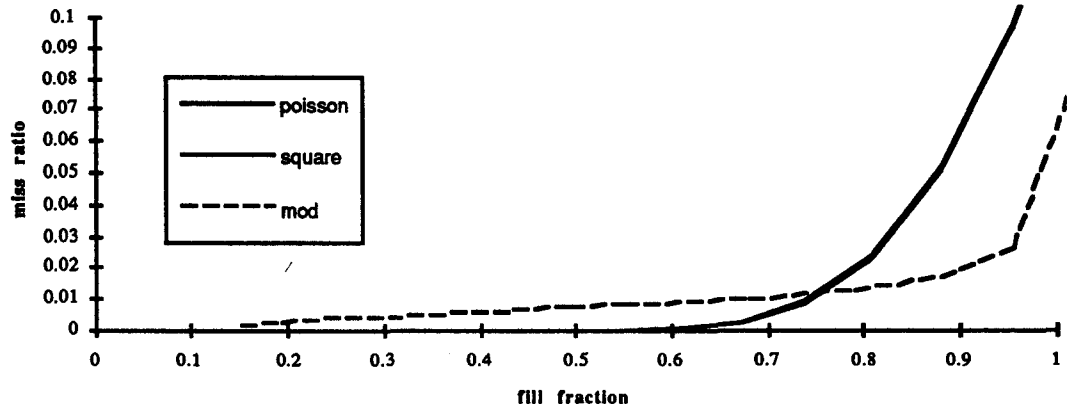

Figure 8: Miss Ratio versus fill fraction for: 32 way, 2048 set, random replaced cache; averaged over 2048 strides.

Figure 9 repeats the experiment of Figure 8 but plots the maximum miss ratio corresponding to the worst case S. This shows a complete breakdown of modulo access and a modest degradation of square access when selecting a worst case stride. The best case miss ratio for mod placement (not plotted) would be exactly zero up to fill fraction 1. This is far better than the best case miss ratio for square hash placement at large fill fraction. Excellent best case performance for mod demonstrates the ability of mod placement to exploit sequential access .

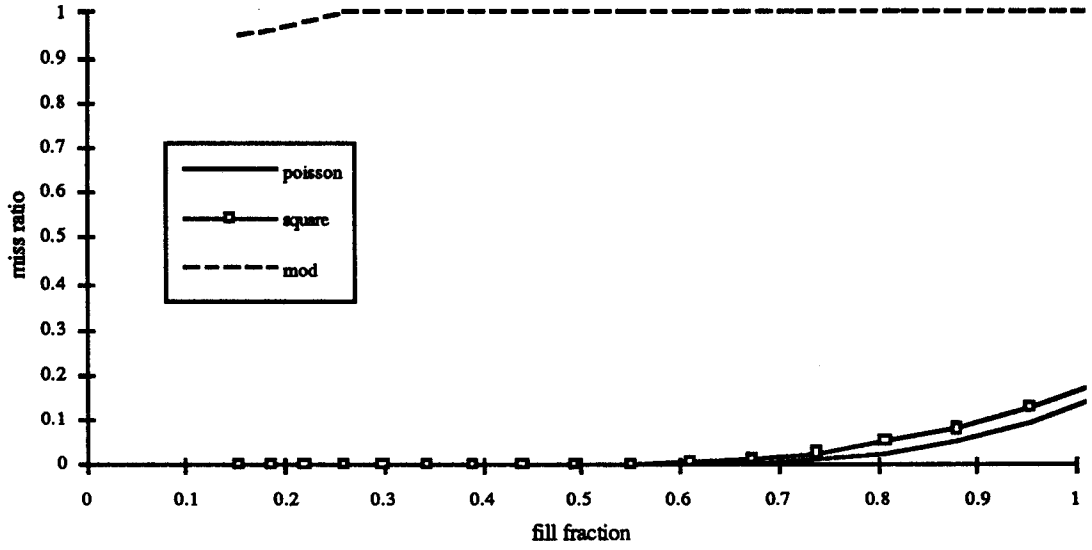

Figure 9: Worst case miss ratio versus fill fraction for: 32 way , 2048 set, random replaced cache; selected from 2048 strides.

Figure 10 illustrates miss ratio versus static array dimension S for a 61% full (M=200), 32 way associative cache. Only the mod function is shown. Square when shown on the same plot hovers above zero. Strong periodic spikes measure full and partial cache breakdown at specific strides. A much deeper understanding of the structure of strided cache breakdown is presented in [18]. Full breakdown occurs at all multiples of 2048 while partial breakdown occurs at strides which when evaluated modulo 2048 are close to fractions 2048/2, 2048/3, 2048/4, etc. Note that a sequence of such strides is measured with dashed arrows originating at 2048. The length of each arrow measures a fraction of 2048 corresponding to a peak miss ratio in the series.

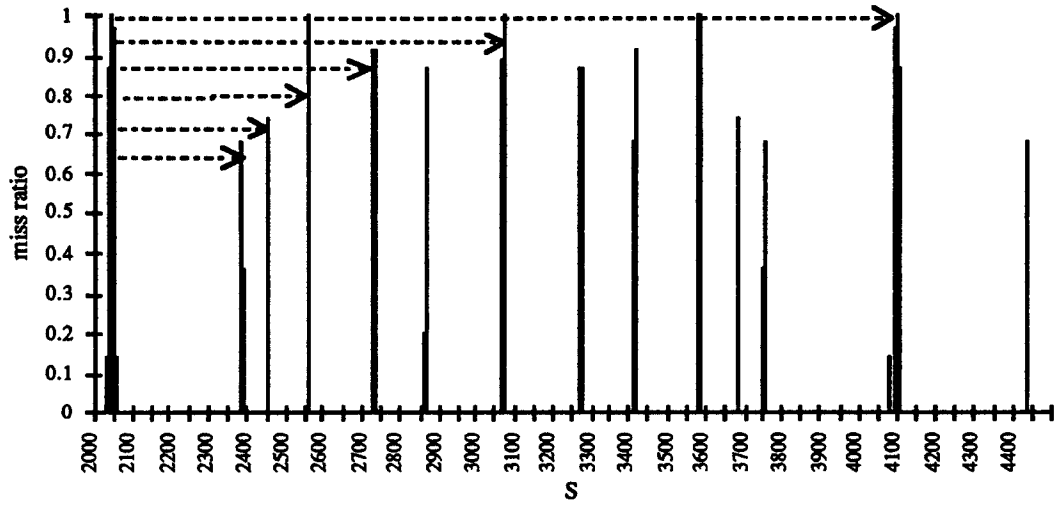

Figure 10: Miss ratio versus array declaration S for 61% full, 32 way , 2048 set, LRU replaced cache using mod hash function.

Figure 11 illustrates miss ratio versus static array dimension S for a 61% full 32 way associative cache using a square hash square function with LRU replacement. A highly magnified axis is used to observe the pseudo-random miss ratio swept out by the square function. The mean of this distribution is approximately centered about a mean predicted by the Poisson miss analysis at .0059.

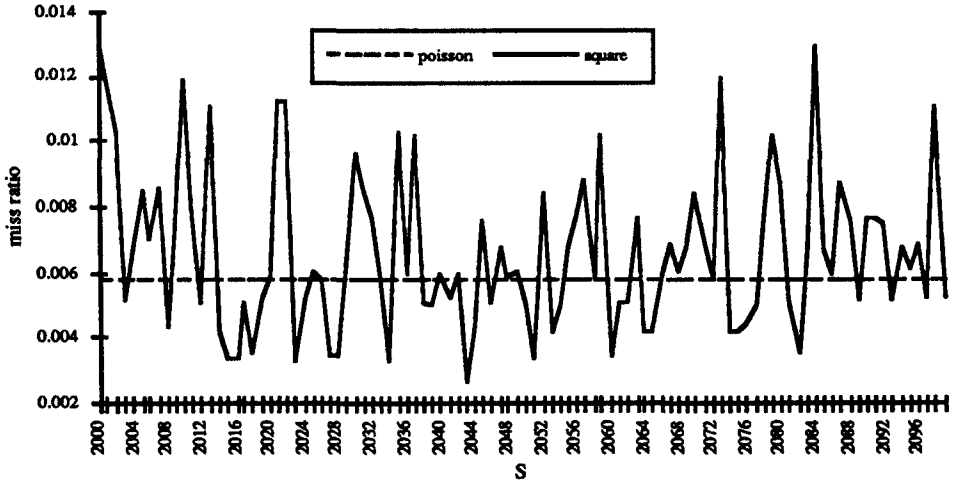

Figure 11: Miss ratio versus array declaration S for 61% full, 32 way, 2048 set , LRU replaced cache using square hash function.

Figure 12 illustrates a cumulative set population distribution obtained for the experiments of Figures 7, 8, and 9. The analytic Poisson and the square are plotted both as simple lines and are essentially superimposed. The mod function, plotted as a dashed line, is more sharply peaked near the mean set population of 32*.61=19.5. This again illustrates the ability of the modulo address distribution to exploit sequential address placement.

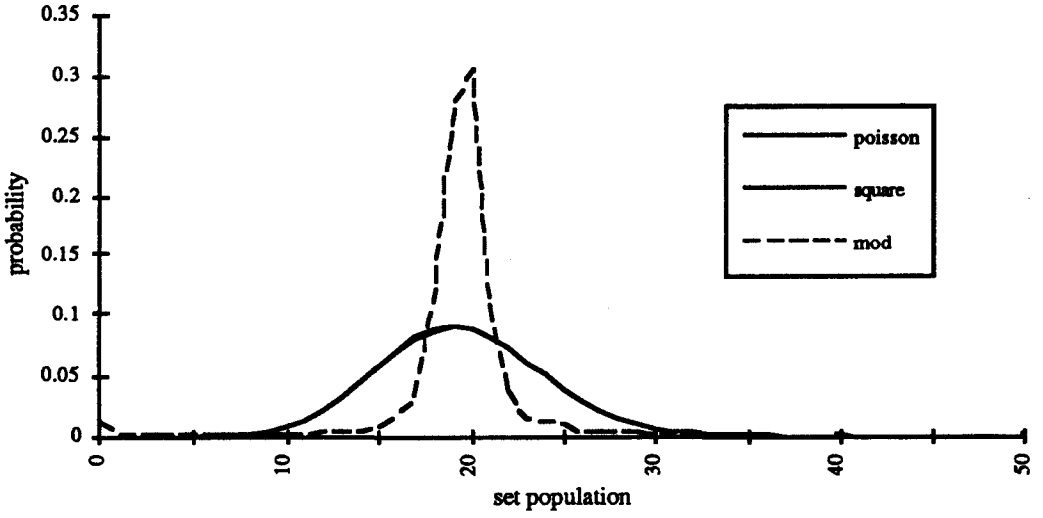

Figure 12: Set population distribution for a 61% full, 32 way , 2048 set cache

It appears in Figure 12 that the sharply peaked mod distribution, when thresholded against a miss region starting at associativity 32, would cause far fewer misses than the broader square and Poisson distributions. This simple set population histogram is not properly weighted to reveal true miss costs. For LRU replacement, cost contributions to miss are proportional to the number of members (distance from vertical axis) in any set having over 32 members.

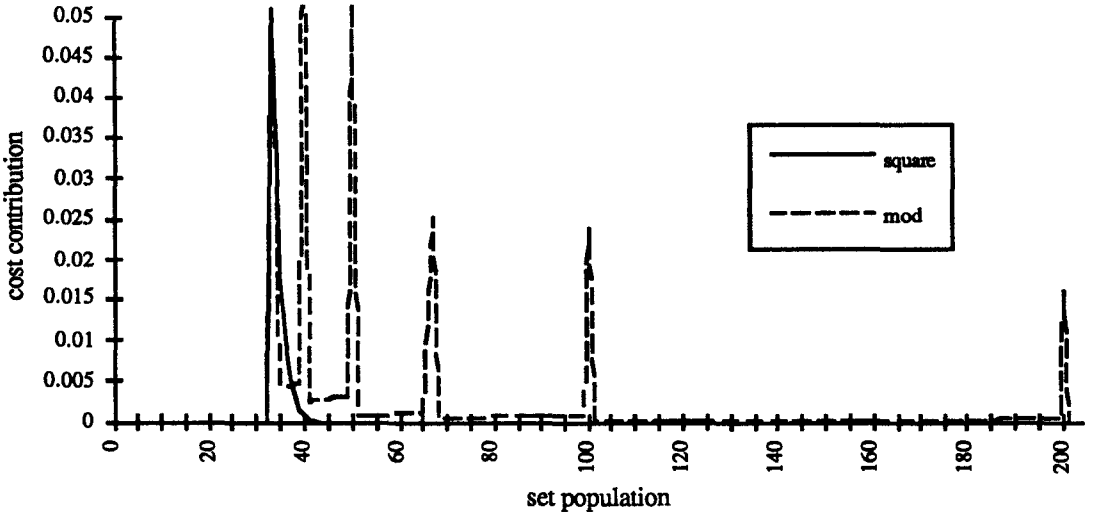

Figure 13: Cost contribution plot for a 61% full, 32 way, 2048 set cache.

In Figure 13, an LRU cost contribution is accumulated across 2048 strides. This cost contribution is calculated by adding the probability a set has achieved a given population times that population. Costs are histogrammed across 2048 experiments into bins corresponding to population, and then plotted. Cost contributions are multiplied by zero for set populations less than or equal to the degree of associativity. In Figure 13, we see clearly defined cost peaks at 200, 100, 66, 50, 40, 33. These represent cost contributions incurred when S MOD 2048 is an integral fraction 2048. Resulting populations at these strides correspond to fractions 1, 1/2, 1/3, ... of the data matrix dimensionality M=200.

The steady state performance of cyclic sweep can be substantially improved through random replacement. Consider the function mr(α,i) of Figure 5 describing the miss ratio of a single set with associativity α processing a cyclic sweep of length i. When random replacement is used, cost contributions in Figure 12 within the miss region (to the right of 32) are multiplied mr(α,i). The penalty for sets whose population slightly exceeds the degree of associativity is greatly reduced by this function while the penalty for sets whose population greatly exceeds degree of associativity is not similarly reduced. When data is randomly placed through hashing (with sufficiently low fill fraction and high degree of associativity), the tail of the Poisson distribution rapidly approaches zero for populations greater than α. Here, sets within the miss region with populations which greatly exceed α are highly improbable. Because of this, random replacement tolerates a higher fill fraction than LRU. In Figure 4, a 32 way LRU cache exceeds 1% miss at a fill fraction of 62%. When using random replacement (Figure 6), the same cache exceeds 1% miss at a fill fraction of 72%.

4. A Placement-Insensitive Cache Simulation

A cache simulation model has been written allowing the collection of cache miss results from address traces. The modeled cache is a 2048 set 32-way associative cache. Two simple trace generators have been designed. The first repeatedly sweeps data within an M by M square in a matrix of static dimension S in order to facilitate calibration with the theoretical model. The second implements a simple M by M matrix multiply. If one has blocked a matrix algorithm, our address trace approximates an M by M matrix sub-block, where addresses within the sub-block are repeatedly accessed. Matrices of size M ranging from 50 to 130 are explored in steps of five. The corresponding fill fraction for each matrix multiply experiment is considered to be the size of the M by M result matrix divided by the total size of the cache. This reflects an approximation: the result matrix is cache resident while input matrices make an insignificant contribution to cache requirements.

Matrix sweep experiments are traversed 50 times with no measurement before a simulated "steady state" measurement begins on the fifty first trial. Matrix multiply experiments are repeated exactly twice. Data collection begins with the second matrix multiply which is considered steady state.

```
for(k=0;k<n;k++)  /*c[i*s+j]+=a[i*s+k]*b[k*s+j]*/
        for(i=0;i<n;i++)
                for(j=0;j<n;j++){
                        cache_access(&c[i*s+j]);
                        cache_access(&a[i*s+k]);
                        cache_access(&b[k*s+j]);
                        cache_access(&c[i*s+j]);
                        }
```

Table 2 Matrix Multiply Address Generator

Figure 14 compares statistically derived miss analyses against actual cache simulations. The Poisson analysis, matrix sweep simulation and matrix multiply simulation are compared all using LRU replacement. An additional plot multiplies by four the matrix multiply miss rate in order to provide a scaled comparison. The Poisson analysis, and the matrix sweep simulation appear essentially superimposed. This validates that the analysis is indeed accurate.

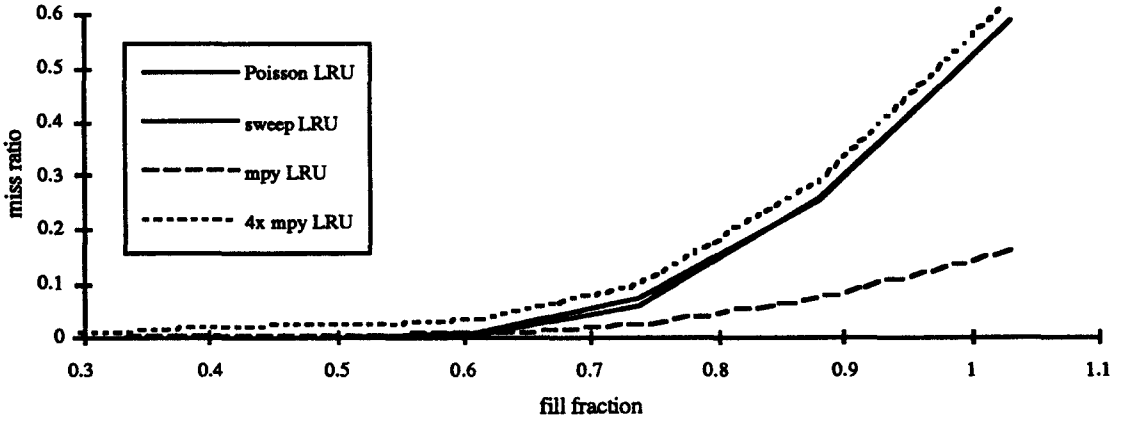

Figure 14: Miss ratio for LRU replacement cache simulation

The matrix multiply algorithm of Table 2 makes only one new reference in four to the large c data array. Three references having much better temporal locality are references to matrices a and b, and a repeated reference to c. When matrix multiplication miss results are multiplied by four we assume that three out of four total references always hit the cache and have no effect on cache miss. Under this assumption, we directly compare the 4× scaled matrix multiply plot to cyclic sweep. This plot is close to but just above the Poisson and matrix sweep plots. Cyclic sweep accurately predicts matrix multiply performance because, the vast majority of misses experienced are due to newly referenced c elements in the loop body. A smaller number of misses occur due to a and b array references causing a small vertical error.

Figure 15 repeats the experiment of Figure 14 for random replacement. Once again the Poisson prediction and the matrix sweep simulation experiment are closely superimposed. Now, the 4× multiply plot parallels but is substantially above the Poisson and sweep plots. Cache misses resulting from the steady traversal of the c array are accurately modeled by theory. Cache accesses to the a and b arrays are much more transient in nature. These arrays are accessed with better locality but, rows and columns are faulted into the cache as the solution proceeds. The random replaced cache is particularly inefficient at faulting in new data and may indiscriminately displace recent data as new elements of a and b are imported. When compared to LRU results, the poor transient performance of a randomly replaced cache results in substantially larger vertical error between the Poisson prediction and the 4× matrix multiply plot.

While the steady state analysis presented within this paper represents an important cache access phenomenon, transient effects are also important to performance as indicated by the experiments above. When using the cache simulator, tradeoffs in selecting the replacement policy become apparent. Random replacement provides substantial benefit over LRU in tolerating cyclic sweep because the eviction of recent non-LRU data disrupts the worst case behavior of LRU when processing a reference cycle. However, the randomly replaced cache is substantially slower at incorporating data into the cache when newly addressed data is traversed.

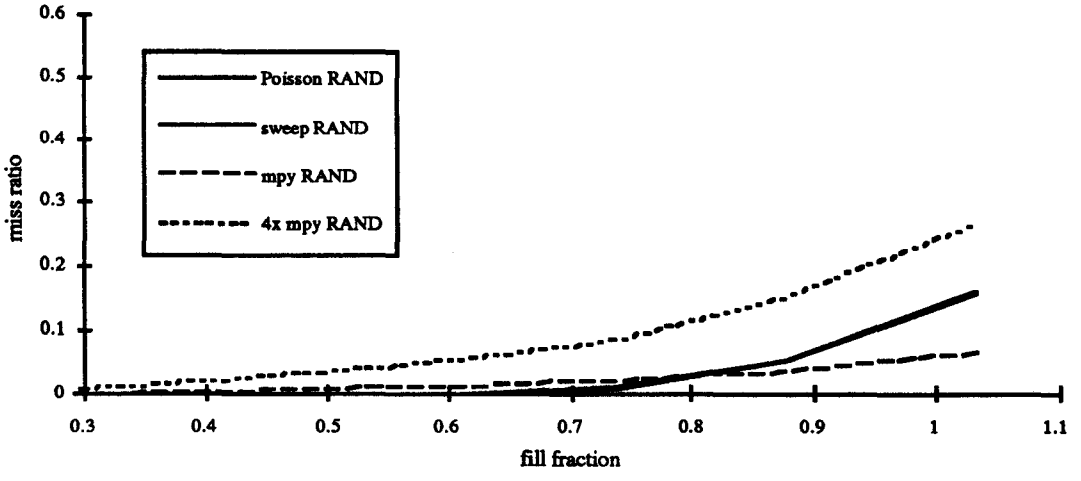

Figure 15: Miss ratio for random replacement cache simulation

5. Conclusions

Placement-insensitive caches can be designed to deliver predictable performance. Access into placement-insensitive caches can be accurately analyzed for cyclic sweep which traverses data having no locality. Cyclic sweep represents an important component of real world data access. Random address streams can be synthesized using address hashing hardware. Hashing techniques result in stable cache performance closely predicted by theory and demonstrate the design of a class of caches which are insensitive to data placement. The variance in cache set population resulting from random placement is masked by providing enough surplus cache memory and high associativity to maintain a predictably low miss ratio. With high associativity, the placement-insensitive cache provides consistently low miss ratios even at high fill fraction. Matrix multiply algorithms operating out of a placement-insensitive cache exhibits characteristic behavior closely tracking theory (for LRU caches).

Acknowledgements

The authors gratefully acknowledge the following individuals. Vinod Kathail contributed valuable suggestions regarding the mathematics of the analysis and the choice of hashing function. Bob Rau provided a better undersanding of the domain of applicability of the work and suggested better means to evaluate the mr(α,i) function. Rajiv Gupta provided a number of comments and suggestions which were of great assistance.

References

1. A. J. Smith. Cache Memories. Computing Surveys 14, 3 (1982), 473-530.

2. M. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. IEEE Transactions on Computers 38, 12 (1989), 1612-1630.

3. S. Przybylski. Cache and Memory Hierarchy Design: a Performance Driven Approach. (Morgan Kaufmann, San Mateo California, 1990).

4. D. Thiebaut, J. L. Wolf, and H. S. Stone. Synthetic Traces for Trace-Driven Simulation of Cache Memories. IEEE Transactions on Computers 41, 4 (1992), 388-410.

5. M. D. Hill. A Case for Direct-Mapped Caches. IEEE Computer 21, 12 (1988), 25-40.

6. C.-H. ;. D. Chi H. Improving Cache Performance by Selective Cache Bypass. Proceedings of the Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. (Kailua-Kona, HI, 1989), 277-285.

7. M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. Proceedings of the ASPLOS-IV (Santa Clara, 1991), 63-74.

8. D. E. Knuth. Sorting and Searching. The Art of Computer Programming, Vol. 3. (Addison Wesley, 1973).

9. W. D. Maurer and T. G. Lewis. Hash Table Methods. Computing Surveys 7, 1 (1975), 5-19.

10. B. R. Rau, M. S. Schlansker, and D. W. Yen. The Cydra 5 Stride-Insensitive Memory System. Proceedings of the International Conference on Parallel Processing (1989), 242-246.

11. R. Raghavan and J. P. Hayes. On Randomly Interleaved Memories. Proceedings of the Supercomputing '90 (1990), 49-58.

12. B. R. Rau. Pseudo-Randomly Interleaved Memory. Proceedings of the 18th International Symposium on Computer Architecture (1991), 74-83.

13. D. Windheiser, *et al.* KSR1 Multiuprocessor: Analysis of Latency Hiding Techniques in a Sparse Solver. Proceedings of the Seventh International Parallel Processing Symposium (Newport Beach, California, 1993).

14. A. J. Smith. A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory. IEEE Transactions on Software Engineering SE-4, 2 (1978), 121-130.

15. M. Easton and R. Fagin. Cold-Start vs. Warm-Start Miss Ratios. Communications of the ACM 21, 10 (1978), 866-872.

16. A. Thomasian. The Structure of Probability Theory with Applications. (McGraw-Hill, New York, 1969).

17. D. E. Knuth. Seminumerical Algorithms. The Art of Computer Programming, Vol. 2. (Addison-Wesley, Reading Masachusetts, 1969).

18. D. H. Bailey, Unfavorable Strides in Cache Memory Systems. Technical Report RNR-92-015. Nasa Ames Research Center, Moffett Field, Ca., 1992.