Modeling Multiprogrammed Caches

Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper presents a simple, yet accurate, model for multiprogrammed caches and validates it against trace-driven simulation. The model takes into account nonstationary behavior of processes and process sharing. By making judicious approximations, the paper shows that a very simple expression of the form $u^2(p-1)/tS$ accurately models the multiprogramming component of the miss rate of large direct-mapped caches. In the above expression, t is the context-switching interval, S is the cache size in blocks, p is the number of processes, and u is the number of unique blocks accessed by a process during the interval t.

1 Introduction

Analytical modeling has never really been a popular method for cache analysis. Rather, with the availability of massively large traces for virtually every conceivable workload, trace-driven simulation has become the dominant method for cache performance analysis. Unfortunately, simulations are not only slow, but they lend little insight into the behavior of caches. Statistical methods for trace analysis [1, 2, 3] help reduce the simulation time, but they are still significantly slower than modeling, and offer little additional insight over full simulation. Because caches are amenable to mathematical analysis, we contend that simulations against massively long traces are unnecessary, except in the final phases of the design process.

Analytical models have two major strengths — they help obtain fast estimates of cache performance, and if the models are simple, they offer insights into cache behavior as well. The analytical cache model developed in [4] characterized the *steady-state* cache miss rate as the sum of three components: nonstationary, intrinsic-interference, and multiprogramming miss rates. The *nonstationary* component is due to misses that bring blocks into the cache for the first time. (These correspond to compulsory misses in the terminology of Hill and Smith [5].) The *intrinsic-interference* component results from the misses caused when the blocks associated with the working set of a given process interfere with each other in the cache. (These misses are further divided into conflict and capacity misses by Hill and Smith.) The *multiprogramming* component arises from misses caused by blocks from different processes competing for cache residency. In a multiprocessor, *coherence-related invalidations* introduce an additional miss-rate component. In this paper, we focus on the multiprogramming component of the miss rate. Understanding this component is important because significant evidence exists showing that it is the chief determinant of the miss rate in large caches [6, 7].

While the model for the multiprogramming component of misses developed in [4] can be used to yield fast estimates of cache performance, it is not simple, and hence does not offer much

S	Number of cache sets (or rows)
u	Working-set size (in blocks) of each process
p	Number of processes or the degree of multiprogramming
D	Degree of associativity (or set size)
t	The context-switching interval measured in memory references

Table 1: Definition of terms used in the multiprogrammed cache model. In our analysis $t = 10,000$.

intuition on cache performance. For example, the nature of the variation of the miss rate with working-set size of a process is not obvious from the expression for the miss rate. It turns out that the miss rate expression for commonly used cache types can be simplified greatly. In fact, this paper shows that the miss rate of large direct-mapped caches is proportional to the degree of multiprogramming and to the square of the working set size, and is inversely proportional to the cache size. Thus, for a program that does not exhibit significant nonstationary behavior, its multiprogrammed-cache performance is related to a single workload parameter – its working-set size.

Furthermore, in recent experiments with the model, we found that its predictions for some of our programs did not correlate well with simulations. We have traced this lack of correlation to nonstationary behavior in programs. The model in [4] made the assumption that when a process is rescheduled after being switched out, it accesses all the blocks that it left behind in the cache in its previous quantum. The same assumption is made by Thiebaut and Stone [8] in their model for computing the transient number of misses when two processes share a cache. Nonstationary behavior in programs causes many of these blocks to become dead, and hence these blocks are never accessed again. For example, for the SIMPLE trace discussed in Section 6, nearly 75% of its working set is renewed each time quantum. This paper drops this assumption and modifies the model to account for nonstationarity, resulting in much more accurate estimates.

This paper also presents an extension to the model that takes into account data sharing between processes. This extension is necessary when significant portions of run time are devoted to operating-system and shared-library calls. The model for sharing indicates that the gains due to sharing are related to the square of the fraction of data that is shared.

The rest of this paper is organized as follows. Section 2 states our assumptions and defines the terms used in the model. Section 3 builds the multiprogramming cache model. Sections 4 and 5 extend the model to include the effects of nonstationarity and sharing respectively. The predictions of the model for various cache sizes and for various numbers of processes are validated against trace-driven simulations in Section 6.

2 Assumptions and Notation

This paper focuses on obtaining a simple, closed-form expression for the miss rates in direct-mapped caches. We will use the notation presented in Table 1. The parameter S is the number of cache sets; in direct-mapped caches, the product of S and the block size yields the cache size.

The working set of a process is the set of distinct blocks a process accesses within a context-switching interval. The notion of the working set of a process used in this paper is conceptually slightly different from the one used in [4], in which the working set was defined to be the set of blocks in *active use* by a process. The size of the working set was measured over some time interval (not necessarily the same as the context-switch interval), which was chosen to be long enough that the rate of acquiring new blocks had dropped significantly below the initial start-up rate. The two notions of working-sets are identical if the time interval chosen to measure the size of the working set is the same as the context-switching interval.

Our working-set parameter u is similar to the *footprint* parameter used by Thiebaut and Stone [8] in their model for the number of cache misses suffered by two processes executing on a processor in a round-robin fashion, but with one notable difference. Thiebaut and Stone's footprint parameter measures the total number of distinct blocks touched by a process during its entire execution. Our working-set parameter is the total number of distinct blocks touched by a process during an uninterrupted sojourn on the processor, that is, during the interval t. When the working set is defined thus, the model remains valid even if a process renews portions of its working set each time quantum, provided we make the corrections for nonstationarity outlined in Section 4.

The model for multiprogrammed caches discussed in this paper makes the following assumptions. Many of these assumptions are not strictly necessary, but help simplify the expression for the miss rate, so that insights into cache performance can be derived without necessarily plotting curves for a wide range of parameters.

- Blocks have a uniform probability of mapping to any cache set. Furthermore, the mappings for different blocks are independent of each other. When a processor accesses contiguous locations in memory, with bit selection mapping, the mapping of adjacent blocks in memory is no longer independent. Miss rates tend to be slightly underpredicted in this case. Conversely, certain portions of the address spaces of various processes tend to map systematically to the same cache sets. In this case, miss rates are generally overestimated. Section 6 assesses the validity of these assumptions.

- To simplify the analysis, this paper assumes that all processes have the same working-set size u. The analysis is a little more complicated when the values of u for each process are different; we will mention the changes to the analysis under these circumstances.

- We assume that all context-switching intervals are of the same length t. (In this paper, the terms context-switching interval and time quantum mean the same thing.) See [4] for a model that does not make this assumption. Briefly, the *number* of multiprogramming-induced misses suffered by a process during any time quantum will not change if the intervals are of different sizes, but the *miss rate* will be inversely proportional to the average size of the context-switching interval, provided the intervals are long enough that each process can reference its entire working set.

- We assume t is long enough for each process to access *all* its live blocks in this interval. In other words, if a block is accessed during the nth interval, then it will be accessed in every subsequent time interval m, until the block becomes dead. Conversely, if a previously accessed block is not referenced in a given time interval, then it is never accessed in a future time interval. Simply put, we ignore *resurrections.* Allowing blocks to be resurrected in later time intervals introduces significant complexity in the model; our experiments show that resurrections materially change the results only when context-switch intervals are very small (say, less than few hundreds of cycles) [9].

- Processes switch in a round-robin fashion. Round-robin switching results in worst-case miss rates, but simplifies the model.

3 A Multiprogrammed Cache Model

This section derives from first principles a very simple expression for the miss rate in multiprogrammed caches. The same expression can also be derived from the model in [4] after making a few simplifying assumptions. The next two sections extend this model to include the effects of nonstationary behavior and sharing.

To facilitate discussion, let us first define two terms – the *cached set* and the *carry-over set* of a process. Consider a process i with working-set size u. As mentioned before, u is the number of unique blocks the process references between context switches. At any given time, some fraction of the working set of process i will be resident in the cache. The *cached set* of process i is the set of blocks from its working set that are resident in the cache. Because multiple blocks can map into the same cache location, the size of the cached set is generally smaller than the size of the working set. Clearly, when a process switches out, the number of its blocks left behind in the cache is the size of its cached set.

When the process i is rescheduled for execution on the processor, it is likely that it will access some or all of the blocks that it left behind when it switched out. Let us define the *carry-over set* of a process as the set of blocks that the process accesses on its return out of those previously left behind in the cache. The term "carry over" comes from the intuition that if a block is left behind in the cache by a process on a context switch, and if the same block is reused when the process returns, then the block can be considered to have been carried over to the next time quantum of the process. In other words, the carry-over set size is the number of cache-resident blocks of a process that the process reuses after a context switch.

A process suffers misses attributable to multiprogramming when the $p-1$ processes scheduled between invocations of process i purge some fraction of the carry-over set of process i. We emphasize that multiprogramming-related misses occur only when blocks that comprise the carry-over set of a process are purged, and not necessarily when blocks from the cached set are purged. The cached set can contain dead blocks, which do not contribute to misses if purged by intervening processes.

The multiprogramming miss rate can now be readily computed by estimating the fraction of the carry-over set of a process that is purged as a function of the number of intervening processes. If the cache uses physical addresses, the virtual-to-physical translation ensures de-aliasing of the addresses of each process; for virtual-address caches, adding process identifiers (PIDs) to the address tag ensures proper de-aliasing. Clearly, if the entire cache is flushed on a context switch, the number of misses the process will suffer on each switch is the size of the carry-over set.

Let us assume, for now, that a process accesses all the blocks in its cached set after a context switch, i.e., no blocks die. (Section 4 discusses the modifications necessary when this assumption is not satisfied.) When all blocks in the cached set are accessed after a context switch, the size of the carry-over set is the same as the size of the cached set.

Let us now compute the size of the carry-over set and then determine the fraction lost due to multiprogramming-related interference. Let us denote the probability that d blocks map into a cache set (or cache row) as $P(d)$. When a block maps to any cache set with equal probability, $P(d)$ has a binomial form. If a cache has S sets, and if the set size is D, then the probability a block maps into a given cache set is $1/S$. Similarly, the probability that d blocks map into the given cache set is $(1/S)^d$. The probability that the remaining $u-d$ blocks do not map into this set is given by $(1-1/S)^{u-d}$. Since we can choose d blocks out of u blocks in $\begin{pmatrix} u \\ d \end{pmatrix}$ ways, we have

$$P(d) = \begin{pmatrix} u \\ d \end{pmatrix} \left(\frac{1}{S}\right)^d \left(1 - \frac{1}{S}\right)^{u-d} \tag{1}$$

When context-switch intervals are large enough to allow complete replenishment of the cached set (and hence that of the carry-over set), the size of the carry-over set of a process is independent of the number of intervening processes, and is completely determined by the parameters of the cache and the parameters of the process.[1] Thus the size of the carry-over set (which has been assumed to be the same as that of the cached set) can be determined by estimating the number of blocks of process i resident in the cache after the process has accessed all the blocks in its working set. We compute this number by counting d blocks for each set that has D or less blocks mapped to it, and D blocks for sets with more than D blocks mapped. This yields,

$$\textit{Cached set size} = \textit{Carry over set size} \; = \; S \sum_{d=0}^{d=D} d\, P(d) \; + \; S \sum_{d=D+1}^{d=u} D\, P(d) \tag{2}$$

It turns out that using the above expression to compute the misses due to multiprogramming leads to slightly pessimistic miss rates. The reason is that not all blocks in the carry-over set purged by intervening processes contribute to multiprogramming misses. When multiple blocks of the same process map to the same cache set, there is a high likelihood that the blocks will be replaced by other blocks of the same process. Whether a block is purged on account of a conflict with another process or due to a conflict with a block of the same process is irrelevant insofar as the overall miss rate of a process is concerned, but identifying the source of a miss is important if we are interested in separating the multiprogramming component from the uniprocessing miss rate.

Therefore, to compute the multiprogramming miss rate, we must estimate the size of the *safe carry-over set*, denoted $v(D)$. The safe carry-over set is simply the set of blocks in the carry-over set that suffer no self interference. We can estimate $v(D)$ by dropping the second term in Equation 2 because the second term corresponds to blocks of the process that conflict with each other. Thus we obtain

$$v(D) = S \sum_{d=0}^{d=D} d\, P(d) \tag{3}$$

Although we expect that using the safe carry-over set will underestimate the multiprogramming miss rate slightly, in practice, we have found through simulations that the results are virtually indistinguishable from those computed with the carry-over set, because the second term in Equation 2 is very small compared to the first term for reasonable cache sizes. As in

[1]When the context-switching interval is smaller, as in multithreaded caches, the analysis must take into account partial replenishment of the cached set of a process [9].

the multiprogrammed model derived by Thiebaut and Stone [8], when self interference is not explicitly considered, the carry-over set can be used instead.

We can simplify the expression for $v(D)$ when $D = 1$ as

$$v(1) = SP(1) = u\left(1 - \frac{1}{S}\right)^{u-1} \tag{4}$$

When $S >> 1$ and $u >> 1$, we can simplify the above expression for $v(1)$ as

$$v(1) \approx ue^{-\frac{u}{S}} \tag{5}$$

The number of multiprogramming misses that process i will suffer is the difference between $v(D)$ and the number of blocks of process i that are still valid when i returns. This number depends on the number of blocks of the intervening processes that map to a set containing valid blocks of process i. When there are $p-1$ intervening processes, we can derive the probability, $P'(d)$, that d blocks from the combined working sets of these processes map into any given cache set as follows. The number of unique blocks referenced by the $p-1$ processes is

$$u' = (p-1)u \tag{6}$$

assuming that all processes have the same properties, and that blocks are not shared between processes. If the working-set sizes are different, we derive u' as the sum of the individual working-set sizes of the intervening processes, i.e.,

$$u' = \sum_{j=1, j\neq i}^{j=p} u_j \tag{7}$$

where u_j denotes the working-set size of process j.

We now obtain

$$P'(d) = \begin{pmatrix} u' \\ d \end{pmatrix} \left(\frac{1}{S}\right)^d \left(1 - \frac{1}{S}\right)^{u'-d}$$

We can now derive the number of misses suffered by a process on its return from a context switch. We will focus on deriving a simple and accurate formula for direct-mapped caches.[2] Because they are simple and fast, direct-mapped caches are popular with processor architects [10, 11]. In a direct-mapped cache ($D = 1$), a block of process i resident in some cache set is purged if one or more blocks from the intervening processes map to that set. The probability one or more blocks of the intervening processes map to a given cache set is one minus the probability zero blocks of the intervening processes map to the given cache set, and is given by

$$1 - P'(0)$$

[2]A more complicated, but general, expression for the miss rate of set-associative caches is derived in [4] as

$$m(p) = \frac{S}{t}\sum_{d=0}^{d=D} P(d) \sum_{e=0}^{e=u'} MIN(d, e+d-D)P'(e)$$

for LRU replacement, where e is the number of blocks of the intervening processes that map into the given set, and $MIN(d, e+d-D)$ evaluates to the minimum of d and $e+d-D$ when both arguments are positive, and evaluates to zero if $e+d-D$ is negative.

The probability a cache set has a block of process i that belongs to the safe carry-over set is simply the probability exactly one block of process i maps to that set, which is $P(1)$. The product of these two probabilities is the probability of a miss in that set. Since the cache has S sets, the total number of misses suffered by process i on being rescheduled is

$$SP(1)(1 - P'(0))$$

Alternatively, the above expression for the number of misses can be viewed as the product of the size of the safe carry-over set of the process i (which is $SP(1)$) and the fraction lost due to collisions with other processes $(1 - P'(0))$.

Because context switches are assumed to happen at intervals of length t, the multiprogramming component of the miss rate is given by

$$m(p) = \frac{1}{t} SP(1)(1 - P'(0)) \tag{8}$$

Substituting for $P(1)$, $P'(0)$, and u',

$$m(p) = \frac{S}{t}\frac{u}{S}\left(1 - \frac{1}{S}\right)^{u-1}\left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u}\right]$$

In the above equation, $\frac{u}{S}\left(1 - \frac{1}{S}\right)^{u-1}$ is the probability a set has *exactly one* block of process i mapped into it in a direct-mapped cache, $\left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u}\right]$ is the probability that at least one of the blocks of the intervening $(p-1)$ processes maps into that cache set purging the block of process i. The product of the above two probabilities and S yields the number of blocks of process i purged.

For typical values of S and u, we can ignore the minus one in the exponent $u - 1$, and write

$$m(p) \approx \frac{u}{t}\left(1 - \frac{1}{S}\right)^{u}\left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u}\right] \tag{9}$$

We can simplify the above expression for the miss rate when the number of sets, S, is large compared to one, which is virtually always true. When $S >> 1$,

$$m(p) \approx \frac{u}{t} e^{-\frac{u}{S}}\left[1 - e^{-\frac{(p-1)u}{S}}\right] \tag{10}$$

Furthermore, when $S >> (p-1)u$, we can ignore the second-order terms in u in the binomial expansion of the product terms in Equation 9 and write

$$m(p) \approx \frac{u}{t}\left(1 - \frac{u}{S}\right)\frac{(p-1)u}{S}$$

Finally, if we can ignore u/S relative to 1, we can write,

$$m(p) \approx \frac{u^2}{t}\frac{(p-1)}{S} \tag{11}$$

This approximation will be henceforth termed the *large-cache approximation.*

The above simple expression for the miss rate indicates that when the cache is much larger than the combined working sets of the individual processes, the miss rate is proportional to the square of the individual working-set sizes and the number of processes, and inversely proportional to the cache size.

The expression also indicates that the multiprogramming miss rate decreases as t increases. Although u increases slightly with the time interval t used to measure it, the rate of increase is generally significantly smaller than linear, which makes the miss rate always decrease with t. There is one circumstance under which the rate of increase of u with t can be significant. When the process exhibits nonstationary behavior, the change in u with t can be non-negligible. However, the nonstationary model (developed in the next section) essentially subtracts the nonstationary component from u, making the miss rate insensitive to the increase in u.

In the domain where the cache is small compared to the combined working-set sizes of all the processes, that is, when $(p-1)u >> S$, the multiprogramming component of the miss rate can be approximated as

$$m(p) = \frac{u}{t} e^{-\frac{u}{S}} \tag{12}$$

The above *small-cache approximation*, obtained by simplifying Equation 10, says that the entire carry-over set of a process, computed as $ue^{-\frac{u}{S}}$ (see Equation 5), is purged when the degree of multiprogramming is very high. We emphasize that the above miss rate is the *component* due to multiprogramming.

Finally, as mentioned earlier, if the entire cache is flushed on a context switch, the miss rate (denoted $m_{flush}(p)$ to distinguish it from the multiprogramming miss rate of physical caches or the miss rate of caches with PIDs) is the size of its safe carry-over set divided by the context-switching interval, and is given by

$$m_{flush}(p) = \frac{S}{t} \sum_{d=0}^{d=D} d\, P(d) \tag{13}$$

which, as one would expect, is independent of the multiprogramming level. When $S >> 1$, we can simplify the above for direct-mapped caches ($D = 1$) as follows.

$$m_{flush}(p) = \frac{u}{t} e^{-\frac{u}{S}}$$

When $S >> u$, we can simplify the expression for the miss rate further, as

$$m_{flush}(p) = \frac{u}{t} \left(1 - \frac{u}{S}\right)$$

Finally, if we can ignore u/S relative to 1, we get $m_{flush}(p) = u/t$, which simply says that the cache can hold the entire working set of the process, but loses it on each context switch.

4 Including the Effects of Nonstationarity

An important source of error with the model discussed in the previous section was our assumption that *every* block left behind in the cache when a process relinquished the processor is reused by the process on its return. Consequently, if the working set changed significantly between

u_{ns}	Number of blocks fetched for the first time during a context-switch interval
u	Working-set size of each process (as defined previously)
U	Total number of distinct blocks in the trace
n	Number of time intervals in the trace

Table 2: Definition of terms used in the multiprogramming cache model to compensate for nonstationarity.

context switches, the model would count as misses purged blocks that are actually dead, thereby overestimating the multiprogramming miss rate. Note that the blocks that are fetched for the first time are counted in the category of nonstationary misses.

We can drop our assumption of full reuse of the cached set and estimate the size of the safe carry-over set $v(D)$ more accurately. We can estimate $v(D)$ more accurately by reducing its value computed in Equation 3 by the fraction of nonrenewed blocks. That is, we multiply $v(D)$ computed in Equation 3 by $(u - u_{ns})/u$, where u_{ns} is the average number of blocks renewed by a process during a context-switch interval. In other words, the safe carry-over set size of process i, which includes nonstationary effects, is estimated as

$$v(D) = \frac{(u - u_{ns})}{u} S \sum_{d=0}^{d=D} d\, P(d)$$

Using the notation in Table 2, the number of blocks that are fetched during each context-switch interval for the first time (or the number of blocks that are renewed, u_{ns}) can be estimated as the total number of unique blocks in a trace (U) divided by the number of time intervals (n). In other words,

$$u_{ns} = \frac{U}{n} \tag{14}$$

Note that we are ignoring resurrections (as defined in Section 2) in computing u_{ns}. In other words, we are assuming that the blocks renewed by the process are truly first-time fetches and not fetches of blocks referenced in some earlier time quantum.

The expressions for u' and $P'(D)$ in the expression for the miss rate remain unchanged. Accordingly, the multiprogramming component of the miss rate for direct-mapped caches can be derived by substituting the new expression for $v(1)$ (which is $\frac{u_{ns}}{u} SP(1)$) in place of $SP(1)$ in Equation 8 as shown below.

$$m(p) = \frac{1}{t} \frac{(u - u_{ns})}{u} SP(1)(1 - P'(0)) \tag{15}$$

The above equation simplifies to

$$m(p) = \frac{S}{t} \frac{(u - u_{ns})}{S} \left(1 - \frac{1}{S}\right)^{u-1} \left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u}\right] \tag{16}$$

We can further simplify the above equation if $S >> (p-1)u$.

$$m(p) \approx \frac{(u - u_{ns})}{t} \frac{(p-1)u}{S} \tag{17}$$

4.1 An Alternate Method of Computing u_{ns}

While our previous analysis used a fixed value of u_{ns} for each time quantum, considerable amount of empirical evidence indicates that u_{ns} diminishes with time. Several studies have found that the cumulative number of unique blocks accessed by a program grows less than linearly with time [12, 13, 14, 6]. Specifically, Belady and Kuehner [12], Kobayashi and MacDougall [13], and Thiebaut [14] have suggested that the total number of unique data blocks referenced by a program grows as some power function of time. If r denotes the time since the program started executing, the total number of unique blocks can be represented as

$$ar^b$$

where a and b are some constants. Differentiating with respect to r, we obtain the rate at which new blocks are being referenced at time r as

$$abr^{b-1}$$

Because typical values reported for a range from about 1 to 10, and those for b are about 0.5, the rate at which new blocks are added clearly diminishes with time. If n time quanta have elapsed since the process commenced execution, recalling that the size of a time quantum is t, its rate of fetching new blocks can be rewritten as

$$ab(nt)^{b-1}$$

Denoting $u_{ns}(n)$ as the number of blocks renewed during the nth time quantum, and ignoring resurrections as before (see Section 2), we can now estimate $u_{ns}(n)$ as the product of the rate of fetching new blocks and the size of a time quantum

$$u_{ns}(n) = ab(nt)^{b-1}t$$

Because $b-1$ is negative, it is clear that when n becomes large (in the steady-state) the rate of addition of new blocks is very small, and we can ignore nonstationary effects in computing the multiprogramming component of the miss rate. Thus, nonstationary effects will be significant during start-up and negligible in the steady state.

5 Effect of Shared Data

Portions of process working sets that are shared are immune to interference from other processes. Let u_{priv} be the number of blocks that are private to a given process out of the u blocks in its working set. Let us also assume that when a piece of data is shared, it is shared by all processes. Ignoring the effect of nonstationarity for now, the multiprogramming miss-rate component is easily derived by making the following changes in Equation 8:

1. reducing the size of the safe carry-over set of process i used in computing the miss rate by the fraction of blocks that are private, i.e., by multiplying $v(1) = SP(1)$ by u_{priv}/u,

2. and using u_{priv} in place of u in computing the probability that one or more blocks of the intervening processes map to any given cache set.

Making the appropriate substitutions and assuming $D = 1$, we have

$$m(p) = \frac{1}{t}\frac{u_{priv}}{u} SP(1)(1 - P'(0)) \tag{18}$$

where we use u_{priv} in place of u in computing $P'(0)$. Simplifying, we get

$$m(p) = \frac{u_{priv}}{t}\left(1 - \frac{1}{S}\right)^{u-1}\left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u_{priv}}\right] \tag{19}$$

Furthermore, when $S >> (p - 1)u$,

$$m(p) \approx \frac{u_{priv}^2}{t}\frac{(p-1)}{S} \tag{20}$$

The situation is a little more complicated when the process suffers nonstationary misses as well. First, the number of private blocks out of the safe carry-over set (previously computed as $(u_{priv}/u)v(1)$) must be reduced as shown in Section 4. That is, $(u_{priv}/u)v(1)$ for the process whose multiprogramming miss rate is desired must be replaced by

$$\frac{u_{priv}}{u}\frac{(u_{priv} - u_{priv_{ns}})}{u_{priv}}v(1) = \frac{(u_{priv} - u_{priv_{ns}})}{u}v(1)$$

where, $u_{priv_{ns}}$ denotes the number of private blocks renewed each time quantum, and can be estimated as

$$u_{priv_{ns}} = u_{priv}\frac{u_{ns}}{u}$$

As before, $u_{ns} = U/n$, denotes the number of new blocks fetched during a time quantum. The above expression for the private portion of the working set affected by nonstationary fetches assumes that each of the private and shared blocks is equally likely to be replaced by a new block of the same type.

Second, the cost of fetching shared blocks into the cache for the first time must be amortized over all the processes. When a process has sole use of the processor, the miss-rate component m_{ns} corresponding to first-time fetches of blocks in the steady state (ignoring the initial start-up effects) is the average number of first-time fetches in a quantum (u_{ns}) divided by the size of the time quantum t. Thus, for the single-process case,

$$m_{ns}(1) = \frac{u_{ns}}{t} \tag{21}$$

When p processes share the cache, each with u_{priv} private blocks and $u - u_{priv}$ shared blocks, the cost of first-time fetches of shared blocks are amortized over all the processes that share these blocks, while the cost of first-time fetches of private blocks are incurred by each individual process. If some fraction of the shared blocks are renewed, and if all p processes access these blocks, the per-process nonstationary miss rate with p processes is

$$m_{ns}(p) = \frac{1}{t}\left(u_{ns}\frac{u_{priv}}{u} + u_{ns}\frac{(u - u_{priv})}{u}\frac{1}{p}\right) \tag{22}$$

If we know the exact number of processes that share a given fraction of the renewed blocks, then we can obtain a more accurate estimate of m_{ns} by replacing p in the second term of the above equation by this number. Alternatively, if the entire set of shared blocks are stationary, then m_{ns} for multiprogramming is the same as that for a single process (Equation 21).

The difference between the two values of m_{ns} computed in Equations 21 and 22 is the change in the miss rate due to multiprogramming, and is given by the following expression.

$$-\frac{u_{ns}}{t}\frac{(u - u_{priv})}{u}\frac{(p-1)}{p}$$

Thus the overall miss-rate component due to multiprogramming becomes

$$m(p) = \frac{(u_{priv} - u_{priv_{ns}})}{t}\left(1 - \frac{1}{S}\right)^{u-1}\left[1 - \left(1 - \frac{1}{S}\right)^{(p-1)u_{priv}}\right] - \frac{u_{ns}}{t}\frac{(u - u_{priv})}{u}\frac{(p-1)}{p} \quad (23)$$

The first term in the above equation is the context-switching component as before. Because shared cache blocks are immune to interference from context switching, the safe carry-over component is replaced by the corresponding private component. The second term represents the decrease in the miss rate when all processes share the cost of fetching common blocks into the cache for the first time.

When the second term is larger than the first, multiprogramming can actually result in a decrease in the miss rate experienced by a single process. This phenomenon helps explain the anomalous behavior of the miss rate in multiprogrammed workloads observed in [6]. For large caches, the miss rate for multiprogrammed workloads was observed to be lower than the single processor case. (In the single process experiments, each process was run with a clean cache.) In large caches, the nonstationary component comprises a significant fraction of the miss rate, and amortization of the misses for the shared operating-system code and data over several processes resulted in the lower observed cache miss rates for a single process.

The above expression for the miss rate can be simplified when $S >> (p-1)u$ yielding

$$m(p) \approx \frac{(u_{priv} - u_{priv_{ns}})}{t}\frac{u_{priv}(p-1)}{S} - \frac{u_{ns}}{t}\frac{(u - u_{priv})}{u}\frac{(p-1)}{p} \quad (24)$$

6 Comparison with Simulations

We conducted several experiments to verify that our approximations were indeed valid. We compare the predicted cache miss rates with those obtained from trace-driven simulation for various cache sizes and with different degrees of multiprogramming. Trace driven simulation [15] is a popular technique for cache evaluation, especially since address traces for various workloads are publicly available [16, 7]. This section first describes our simulation methodology and then analyzes the results.

6.1 Simulation Methodology

Our experiments with the model will use both real traces of multiprogrammed workloads and replicated traces. Replicated traces are like multiprogrammed traces, but they are synthesized from a single-process trace as explained below.

We use real traces of multiprogrammed workloads to observe the variation of cache miss rate with cache size. These traces are obtained using ATUM [16] and described in detail in [6]. The multiprogramming traces include MUL6, MUL9 and MUL12, which correspond to VMS workloads for multiprogramming levels of six, nine, and twelve respectively. Examples of processes active in these traces include compilers, SPICE, and Jacobi relaxation.

The multiprogramming cache model yields the miss rate of a *given* process in a task-switching environment from the parameters of that process and those of the other active processes in the trace. We designed our simulation experiments to yield the individual miss rates of the constituent processes in the multitasking traces. The results that we present are the averages obtained over 10 processes selected at random from the three traces. We used only user references in this experiment. The number of *user* references in MUL6, MUL9 and MUL12 are 1.3 million, 1.3 million, and 0.9 million respectively.

We use *replicated traces* to assess the accuracy of the model as the degree of multiprogramming is changed. A replicated trace is created from a single-process trace by round-robin scheduling p instances of the single-process trace to simulate a multiprogramming level of p, assuming a context-switch interval of 10,000 references. Each instance of the trace is assigned a distinct, random number as a process identifier (PID). The PID of each constituent process is hashed in with the addresses to randomize the locations in the cache occupied by corresponding references of the p subprocesses. The PID hashing scheme has also been suggested as a way of improving the performance of virtual-address caches [6].

While traces of real multiprogramming workloads are clearly more accurate, the replicated traces are easier to use in validation experiments because they can be created with unlimited, varying numbers of threads that have the same statistical properties. Replicated traces can also be created easily with single-processor traces. In our replicated traces, the single-process trace is roughly 300,000 references long, and the replicated traces are longer by a factor p, where p is the degree of multiprogramming.

The replicated traces for experiments with multiprogramming levels use the following constituent traces:

LocusRoute: An eight-processor, physical-address trace of a parallel global router for VLSI standard cells. The trace was obtained using the VAX trap bit with round-robin scheduling of processes. A single processor trace is extracted from this trace for replication.

SIMPLE: A 64-processor, virtual-address trace of a program modeling hydrodynamic and thermal behavior of fluids in two dimensions. The trace was created by a post-mortem scheduling scheme at IBM. A single processor trace is extracted from this trace as well for replication.

IVEX: A single-processor, virtual-address trace of a DEC program, Interconnect Verify, checking net lists in a VLSI chip (under VMS). This trace includes operating system references.

6.2 Parameter Extraction

The multiprogramming model depends on a single process-specific parameter, u, which is the working-set size. We obtain u for a trace by measuring the number of unique blocks it accesses in each time quantum t and taking the average value over the whole trace. The time quantum size t is taken to be $10,000$ since in most of the VAX traces that seemed to be the context-switching interval. The working-set sizes for the IVEX, LocusRoute, and SIMPLE traces are summarized in Table 3. All our experiments use a block size of 16 bytes.

Trace	u	u_{ns}
IVEX	675	294
LocusRoute	295	53
SIMPLE	576	427

Table 3: Working-set sizes and the nonstationary components for the three replicated traces. Working-set sizes are measured in blocks of size 16 bytes.

We compute u' for a replicated trace by multiplying the u obtained for the original trace by $(p-1)$ as suggestion by Equation 6. However, u' for the real multiprogramming traces were obtained using Equation 7 since the various processes in the trace have different working-set sizes.

6.3 Validation

Figure 1 summarizes the cache performance of multiprogramming workloads as the cache size is changed. Cache sizes smaller than 4K bytes are not shown because all the curves are very similar to each other. The circle symbol corresponds to the multiprogramming miss rate assuming that each process is assigned a unique process identifier (PID) that is appended to the tag portion of the address. The PID scheme allows blocks of multiple processes to co-reside in a virtual-address cache. The PID scheme also approximates the miss rate of a physical-address cache. The square symbol depicts the miss rate of a cache flushed on every context switch. The dashed lines show the corresponding model estimates. The accurate model shown in Equation 8 is used for this figure because we are interested in a wide range of cache sizes, including those that are smaller than the working-set size of the process.

It is clear that the model successfully predicts the miss rate for both types of caches. As is clear from the figure, a virtual-address cache that is flushed on every process switch performs poorly relative to the PID scheme for cache sizes greater than 16K bytes. This is because a significant fraction of the blocks of a process are reused across process switches in our traces when the cache size is larger than the working set of the process. All the schemes perform the same when the cache is smaller than the working-set size of a process, because the number of blocks that can be reused across task switches (the carry-over set) is bounded by the cache size.

The form of the multiprogramming-induced component of the miss rate for the PID scheme shows a maximum at a cache size of around 32K bytes, and decreases for caches smaller than this size and for caches larger than this size. The reason for the low value in small caches, despite the high probability of the process blocks being purged, is that the size of the carry-over set is small. Conversely, in large caches, although the carry-over set is large, the probability of blocks being purged is small because large caches can simultaneously hold the working sets of multiple processes.

We can estimate the cache size for which the multiprogramming miss rate is a maximum from Equation 9. Differentiating with respect to the number of cache sets S and equating to zero, we derive S which yields the maximum value for the miss rate as,

$$S = \frac{(p-1)u}{ln(p)}$$

The model underestimates slightly the miss rate for large caches. This inaccuracy results because our assumption of uniform and independent mapping of addresses to cache blocks is

Figure 1: The multiprogramming component of the cache miss rate versus cache size for the PID scheme and for flushing on every miss. Block size is 16 bytes and caches are direct-mapped. Solid lines correspond to simulation results and dashed lines correspond to model estimates.

not entirely true. We found that several addresses from the various processes that constituted the multiprogramming trace systematically collided with each other in the cache. For example, the low address space is used by all processes for instructions. Similarly, the user stack top of all processes map into the same cache regions. To randomize the mapping of a process's blocks with respect to the blocks of other processes, we hash the PID in with the set selection bits, and plot the resulting miss rates in Figure 2. Hashing reduces the multiprogramming miss rate in large caches (256K to 1M byte) and provides a better match with model results.

Figure 3 displays the ranges of cache sizes over which the approximate expressions for cache miss rates are valid. We choose $u = 4000$ blocks, $D = 1$, and $p = 8$. The block size is 16 bytes as before. The small-cache approximation is virtually indistinguishable from the accurate model for caches less than 128K bytes in size, providing further evidence that multiprogramming purges the entire working set of processes in small caches. The large-cache approximation is accurate for caches larger than approximately 2M bytes in size.

Given these observations, how can we choose the right approximation? We suggest using the working-set size as a guide in selecting an approximation. For the above example, the large-cache approximation is suitable when the cache size is greater than twice the product of u and $(p-1)$ ($2 \times 4000 \times 7 \times 16$ bytes). The small-cache approximation is suitable when the cache is smaller than twice the working-set size of an individual process ($2 \times 4000 \times 16$ bytes).

We now turn to the use of replicated traces to analyze the behavior of the model as the degree of multiprogramming is increased. Figure 4 compares the multiprogramming miss rate for the model and simulations with the IVEX, LocusRoute, and SIMPLE traces for the following cache parameters. Block size is 16 bytes, number of cache sets S is 4K, and caches are direct mapped. These three traces were chosen because they have widely different working-set sizes. The results

Figure 2: The multiprogramming component of the cache miss rate with PID hashing.

Figure 3: Assessing the ranges of validity of the simplified cache models. Solid curves are obtained using the accurate model, the dashed curve represents the small-cache approximation, and the dotted curve corresponds to the large-cache approximation.

Figure 4: Increase in the miss rate due to multiprogramming as the level of multiprogramming is increased. Solid lines correspond to simulation and dashed lines correspond to the model.

here include the correction due to nonstationarity, because significant portions of the working sets of IVEX and SIMPLE are renewed each time quantum. It is easy to see that the model compares well with simulations in the entire range from low to high levels of multiprogramming.

Figures 5 through 7 assess the range over which the approximate linear model is valid. As suggested in Section 3 the linear approximation is valid only when $S >> (p-1)u$, that is, when the sum of the individual working-set sizes of the individual processes is much smaller than the cache size. See Table 3 for the working-set sizes (in blocks) and the nonstationary components of each trace.

7 Conclusions

This paper presented a model for multiprogrammed caches and validated it against trace-driven simulation. The model takes into account the effects of nonstationary program behavior and process sharing. The paper shows that the multiprogramming miss rate of large direct-mapped caches, which are commonly used in todays workstations, can be characterized by the following simple expression: $u^2(p-1)/St$, where t is the context-switching interval, S is the cache size in blocks, p is the number of processes, and u is the number of unique blocks accessed by a process during the interval t. Furthermore, when the cache size is small the multiprogramming component of the miss rate is captured by the expression $(u/t)e^{-u/S}$.

Figure 5: The linear approximation for multiprogrammed caches for LocusRoute. The solid lines depict simulation results, the dashed lines correspond to the accurate model and the dotted lines correspond to the linear approximation.

Figure 6: The linear approximation for multiprogrammed caches for IVEX.

Figure 7: The linear approximation for multiprogrammed caches SIMPLE.

8 Acknowledgments

I wish to thank Charlie Selvidge, Jonathan Babb, and Donald Yeung for providing useful feedback on a draft of this paper. I am also grateful to Harold Stone for encouraging me to look into simplifying the model for multiprogrammed caches, and for suggesting possible approaches. The trace data was made available to us by DEC and IBM. This research is funded in part by NSF grant # MIP-9012773, in part by DARPA contract # N00014-87-K-0825, in part by a NSF Presidential Young Investigator Award.

References

[1] Thomas R. Puzak. *Analysis of Cache Replacement Algorithms.* PhD thesis, University of Massachusetts, Department of Electrical and Computer Engineering, February 1985.

[2] Subhasis Laha, Janak H. Patel, and Ravishankar K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on Computers*, 37(11):1325–1336, November 1988.

[3] Anant Agarwal and Minor Huffman. Blocking: Exploiting Spatial Locality for Trace Compaction. In *Proceedings of ACM SIGMETRICS 1990*, May 1990.

[4] Anant Agarwal, Mark Horowitz, and John Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

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

[6] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating Systems and Multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[7] Jeffrey C. Mogul and Anita Borg. The Effect of Context Switches on Cache Performance. In *ASPLOS-IV Proceedings*, pages 75–84, April 1991.

[8] Dominique Thiebaut and Harold S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[9] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. 1992. To appear in IEEE Transactions on Parallel and Distributed Systems. Available as MIT Laboratory for Computer Science Technical Report TR-501, April 1991.

[10] Mark D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25 – 40, December 1988.

[11] Steven A. Przybylski. *Performance-Directed Memory Hierarchy Design.* Technical Report CSL-TR-88-366, Stanford University, Stanford, CA, September 1988.

[12] L. A. Belady and C. J. Kuehner. Dynamic Space-Sharing in Computer Systems. *Communications of the ACM*, 12(5):282–288, May 1969.

[13] Makoto Kobayashi and Myron H. MacDougall. The Stack Growth Function: Cache Line Reference Models. *IEEE Transactions on Computers*, 38(6):798–805, June 1989.

[14] Dominique Thiebaut. On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss ratio. *IEEE Transactions on Computers*, 38(7):1012–1026, July 1989.

[15] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[16] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 119–127, IEEE, New York, June 1986.