Optimal Partitioning of Cache Memory

Harold S. Stone, *Fellow, IEEE*, John Turek, *Member, IEEE*, and Joel L. Wolf

***Abstract*—This paper develops a model for studying the optimal allocation of cache memory among two or more competing processes. It uses this model to show that, for the examples studied, the least recently used (LRU) replacement strategy produces cache allocations that are very close to optimal. The optimal fixed allocation of cache among two or more processes is an allocation for which the miss-rate derivative with respect to cache size is equal for all processes.**

The paper also investigates the transient in cache allocation that occurs when program behavior changes, and shows that LRU replacement moves quickly toward the steady-state allocation if it is far from optimal, but converges slowly as the allocation approaches the steady-state allocation. It describes an efficient combinatorial algorithm for determining the optimal steady-state allocation, which, in theory, could be used to reduce the length of the transient. The algorithm generalizes to multilevel cache memories.

For multiprogrammed systems, the paper describes a cache-replacement policy better than LRU replacement. The policy increases the memory available to the running process until the allocation reaches a threshold time that depends both on remaining quantum time and the marginal reduction in miss rate due to an increase in cache allocation. Beyond the threshold time, the replacement policy does not increase the cache memory allocated to the running process.

For all of the questions studied in this paper, the examples shown here illustrate near-optimal performance of LRU replacement, but in the absence of a bound on near optimality the question remains open whether or not LRU replacement is near-optimal in all situations likely to arise in practice.

***Index Terms*—Cache footporint, cache memory, LRU replacement, memory allocation, memory hierarchy, miss rate, miss ratio, multilevel cache memory, power-law model.**

I. Introduction

THIS paper studies the optimal allocation of cache memory among competing processes. Cache memories are high-speed buffer memories whose contents tend to be the most frequently used items accessed by a program. The contents are normally determined by bringing in new items on demand and by using a replacement policy that discards items that are unlikely to be accessed in the near future. When the replacement policy removes the least recently used (LRU) item, caches tend to be very effective, provided that they are large enough to hold the majority of the items that are likely to be active concurrently.

Practical implementations of LRU replacement algorithms do not usually search all entries in a cache, but instead search a small region of the cache that depends on the address reference. The region searched is a called a *set*, and the search is said to be *set associative*. When the region searched contains only a single item, the cache is said to be *direct mapped*. When a new item is brought into the set, some item in the same set is discarded. If the replacement algorithm discards the least recently used item in the set, we designate the replacement algorithm to be an LRU algorithm, and thus both set associative and fully associative caches can be managed by LRU replacement algorithms according to this terminology. Additional information on set-associative and fully associative caches can be found in Smith [13] and Hill [9].

The question of interest is to determine just how well caches behave. We introduce two cache-allocation problems in which processes that have different miss-rate behaviors compete for cache allocation. We find that for neither problem does LRU replacement produce optimal allocations, but the examples in this paper exhibit LRU allocations that are very close to optimal. The data in this paper endorses the almost universal practice of managing cache with LRU replacement. All of the results stated here hold both for set-associative caches and fully associative caches.

The first of the problems studied is the allocation of interlaced data and instruction processes to cache memory. This formulation of the problem was described in Thiebaut, Stone, and Wolf [22]. The paper derives a mathematical model that describes optimal and LRU allocations and gives a validation of the model by means of a trace-driven simulation. Although we are not able to bound the suboptimality of LRU allocations, the evidence presented indicates that they are very good. Our approach is to develop the model of a simpler modified-LRU replacement strategy first, and then embellish this model to obtain a model of pure LRU replacement. The modified-LRU strategy can produce better allocations than those produced by pure LRU for some reference strings.

The measure of optimality used here is the overall miss rate of a cache memory, and an optimal partition is a partition of cache memory among competing processes that achieves a minimum miss rate. Belady [2] introduced an algorithm that is optimal among demand-replacement algorithms. Among all possible ways to choose which cache line to replace on a miss, Belady's algorithm produces the lowest miss rate. Belady's algorithm does not indicate how to partition cache among competing processes, so it cannot be applied directly to the problem addressed in this paper.

For the first allocation problem, the paper also develops a model for the transient behavior of a cache as it moves from one allocation to another in response to a change in the characteristics of data and instruction processes. The differential equation obtained generally cannot be solved neatly in closed

Manuscript received March 29, 1989; revised January 12, 1990 and May 8, 1991.

The authors are with IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

IEEE Log Number 9200307.

form, but can be solved numerically. The model is validated by a trace-driven simulation and by a statistical simulation of the competing processes. Both simulations produce data that fit the mathematical characterization of dynamic behavior. Of related interest in the literature is the paper by Strecker [18], which describes a differential equation to model the dynamics of cache occupancy in the absence of competition.

As a potential means for reducing the allocation transient, we describe an algorithm that first appeared in Thiebaut, Stone, and Wolf [22] for computing the optimal cache allocation. The algorithm allocates lines of cache sequentially among N processes in a way that maintains the miss-rate derivatives as equal as possible, and terminates when the cache memory is fully allocated. We also show how this algorithm generalizes to multilevel cache memory systems.

The second allocation problem treated in this paper is the allocation of cache memory among processes in a multiprogrammed environment. This problem differs from the first because the first scheme deals with interlaced streams, whereas in the second problem, one address-reference stream has exclusive access to cache for a quantum of time, and then yields to a new address-reference stream that has exclusive access for another quantum of time. For the second problem, there tends to be a cache-reload transient each time a new process takes over the processor. The miss rate tends to be high during the early part of the transient, and then drops as the working set of the process becomes resident in cache. A statistical model of the transient that gives an accurate measure of the number of lines reloaded appears in Thiebaut and Stone [21].

For large caches a better replacement policy than LRU replacement is to increase the cache allocation of a running process until the marginal improvement in miss rate multiplied by the time remaining in the quantum is less than some threshold. From this point until the end of the quantum, no additional cache memory should be allocated to the running process. The modified policy tends to retain in cache some items that belong to the next process to run on the machine. When cache is too small to be likely to retain pages of the next process to run, the modified policy is the same as LRU replacement. Thus, the modified policy only makes sense to use when caches are large enough to retain lines of a process in cache through periods when other processes have exclusive use of the processor.

We briefly note some related work. Specifically, Ghanem [7] has studied dynamic partitioning of main memory among competing programs, and his work is the precursor of this work. Replacement strategies for cache have been studied by many people, with notable work by Smith and Goodman [15] and by So and Rechtschaffen [16] among others. Kirk [11] has analyzed the partitioning of an instruction cache into a static partition and an LRU partition. Multilevel caches and the inclusion principle were studied by Baer and Wang [1]. Vernon, Jog, and Sohi [23] have studied performance of hierarchical caches, and proposed optimal multilevel topologies. Przybylski, Horowitz, and Hennessy [12] have also studied optimal multilevel cache hierarchies.

Section II poses the allocation problem for interlaced data and instruction streams, and shows that the miss-rate derivatives are equal when the allocation is optimal. Section III discusses the characteristics of LRU replacement, and shows that it does not converge to the optimal allocation. It also describes a modified-LRU replacement policy and compares its allocations to the LRU allocations. In Section IV, we derive the dynamics for the allocation of memory as it converges to its equilibrium allocation. The efficient algorithm for finding the optimum allocation of cache also appears in Section IV. The results of Sections II through IV rely on several assumptions that are validated in Section V by a trace-driven simulation based on actual data. Section VI treats the allocation of memory to processes in a multiprogrammed system. The generalization of the allocation algorithm to multilevel caches appears in Section VII. An example in Section VII shows that LRU replacement for multilevel cache can come very close to optimal. The last section poses several related research questions that remain open at this time.

II. Allocation of Cache Memory Between Data and Instruction Streams

The model of cache allocation in this section deals with interlaced instruction and data streams that exhibit different cache behaviors. For this idealized form of the model, we show the optimal allocation occurs at a point where the miss-rate derivatives of the competing processes are equal.

For practical reasons, cache implementations at the fastest level of a memory hierarchy do not use fully associative search when seeking a match or an item to replace. Instead they search a small set of items, and replace the least recently used item in the set searched if replacement is necessary. If the set has four or more lines, typical replacement algorithms are further simplified and they only approximate LRU replacement because the complexity of maintaining LRU information for four or more items becomes excessive. At slower levels of a memory hierarchy, such as cache memories associated with large disks, the caches tend to be searched in a fully associative manner. In such caches, true LRU replacement is used for most references, with exceptions made for sequentially accessed data and other reference patterns that are highly predictable.

The focus of this paper is miss rate as a function of cache allocation of individual competing processes. Central to this paper is the assumption that competing processes can be characterized as having a miss rate as a function of allocation size. For fully associative caches, the miss rate for a given reference stream as a function of allocation is indeed a one-parameter function and depends only on the number of lines allocated to a process, since the entire cache is searched for a match during a cache lookup. For set-associative caches, the miss rate depends not only on how many lines are allocated to a process, but where they are in the cache, since only a set of a few lines is actually searched during a lookup.

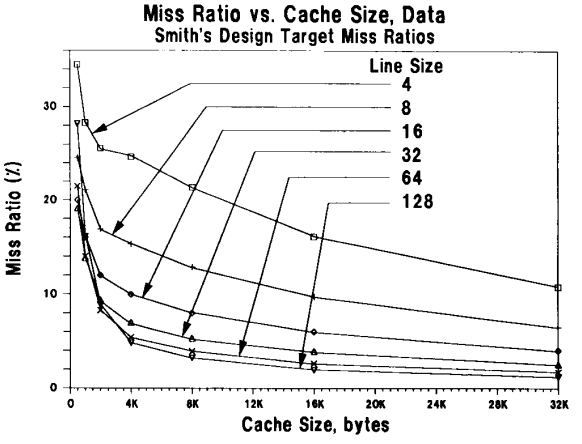
Because a model that accounts for the physical locations of lines allocated in cache is extremely complicated, this paper uses a simplified model of set-associative caches in which the miss rate is a one-parameter function, and that parameter is the number of lines allocated, regardless of the physical

distribution of lines per set. We ignore the effect of the actual distribution of cache lines on miss rate. Thus, set-associative caches are treated in the same way as fully associative caches, and the model is applicable to both types of caches. Section V validates this assumption by demonstrating the agreement between the model and a trace-driven cache simulation. Hence, we use the notation $M_A(x)$ to denote the miss rate of a process, process A, in a cache for which process A has a current allocation of x lines. The miss rate is also a function of the cache structure parameters, which include the number of sets, the set associativity, and the line size. For the purposes of this paper, we fix the structure of a cache, and vary the allocation of a process within that structure. Hence, we do not explicitly identify the cache-structure parameters on which $M_A(x)$ depends when we use this notation.

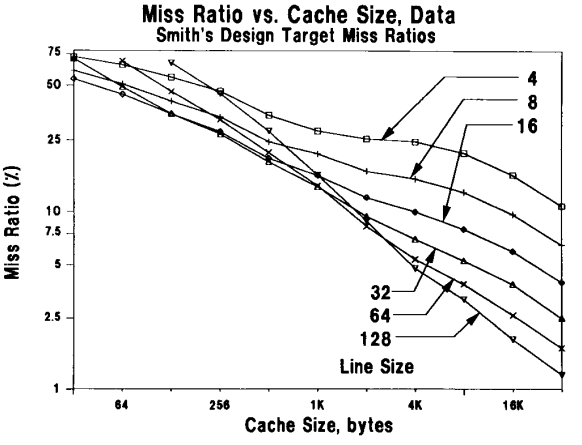
Now we examine the processes that generate the cache references. Assume that an address-reference stream is composed of two interlaced streams of addresses. One stream consists of instruction fetches, and the second stream consists of data fetches. The composite stream is an interleaving of the two streams so that its address references alternate between data and instructions. That is, the stream has the form I, D, I, D, ..., where I and D are instruction and data references, respectively. Each component stream has a known cache behavior given by a miss rate for that stream as a function of the cache memory allocated to the process. Let $M_I(x)$ be the miss rate for the I stream as a function of cache size x, and, similarly, let $M_D(x)$ be the miss rate for the data stream. We assume that both the instruction and data processes are stationary in time, so that the miss rates are not time varying functions.

Although this is a highly idealized model of the I and D processes, the results are not sensitive to the precise way in which the processes are interleaved, provided that the frequencies of the process accesses are equal and long strings of consecutive accesses of one type occur only rarely. The trace-driven validation has approximately equal frequencies of I and D references, but some strings of consecutive D references are hundreds of references long because of the execution of block-move instructions. We also assume that the miss-rate functions are convex functions of cache size. Later in the paper, we examine the case in which the frequencies of the two types of accesses are unequal.

To illustrate the use of the model on representative data, we use the published data from Smith [14] as the source of a running example. Smith's data are design target miss ratios for caches of varying total size and line size, and the data used appear in Figs. 1 and 2. The plots are the miss-rate functions for I and D streams averaged over many different workloads and instruction repertoires. The line sizes in those figures are measured in bytes per line. The log/log plots do not show the convexity of the curves, but the corresponding linear/linear plots demonstrate that these functions are convex except for data caches with 4-byte line sizes in the region between 1K and 4K bytes. We do not claim that Smith's data represent any specific cache design and workload. Therefore, the running example in this section based on Smith's data does not validate the model. The validation appears later.



(a)



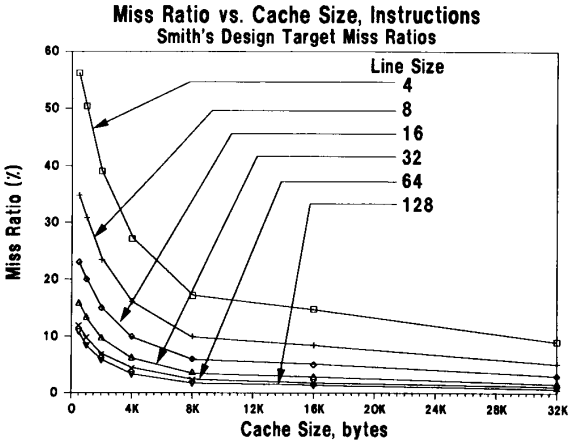
(b)

Fig. 1. Smith's design target miss ratios for data cache. (a) Lin/lin scaling. Line size in bytes. (b) Log/log scaling.

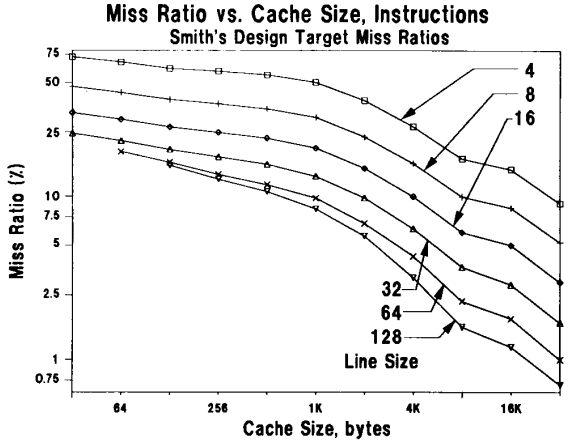
To determine the optimal fixed allocation of cache for the I and D streams, we find an expression for the misses in a period of time that has exactly T references, and find an allocation at which the derivative of the miss rate function goes to zero. Because we take derivatives, we assume that miss-rate functions $M_I(x)$ and $M_D(x)$ are continuous and differentiable, although, in reality, they are measurable only at discrete points on the x-axis. For this derivation we approximate the actual functions by continuous functions.

Assume that we must allocate C bytes of memory between D and I references so that the I stream uses x bytes of memory, and the D stream uses the remaining $C - x$ bytes of memory. Each cache partition is used exclusively by the process that owns it. What value of x achieves the overall minimum miss-rate?

The total number of misses in a time period with T references is the composite miss rate times the length of the period. Since we assume that I and D references occur with equal frequency in the interval T, the total number of misses



(a)



(b)

Fig. 2. Smith's design target miss ratios for instruction cache. (a) Lin/lin scaling. (b) Log/log scaling.

is given by

$$\text{Total misses} = (M_I(x) + M_D(C - x))T/2. \quad (1)$$

To minimize the overall miss rate, we minimize the total misses given in (1) by setting the derivative of the right-hand side of (1) to 0, which occurs at a value of x that satisfies

$$\frac{dM_I(x)}{dx} = -\frac{dM_D(C-x)}{dx} = \left.\frac{dM_D(y)}{dy}\right|_{y=C-x}. \quad (2)$$

By convexity of the miss-rate functions, such a point is indeed a minimum. If either of the miss-rate functions is strictly convex, the point is unique. If the D process occurs r times as frequently in the composite reference stream as the I process, then at the minimum miss-rate the derivative of M_D is weighted by a factor of r. To simplify the discussion, in the remainder of the paper we assume that the factor r is unity, but if not, it should appear as a multiplier of M_D or its derivative wherever they appear. The fact that the optimum allocation appears at a point where the miss-rate derivatives are equal was observed by Ghanem [7] in the context of page faults in a system in which multiple processes compete for main memory. The report by Thiebaut, Stone, and Wolf [22] derived (2) for I and D processes competing for cache memory.

The notion of "optimal" is with respect to the ensemble of possible address-reference streams as represented by the miss-rate functions. Among the possible address-reference streams described by the miss-rate functions are some streams whose optimal allocations can be different from the optimal allocation of the ensemble.

As an example of the application of this theory, we use the miss-rate functions shown in Figs. 1 and 2. Note that the curves in Figs. 1 and 2 have a strong linear structure when plotted on a log/log graph. When using linear regression to fit a straight line to the data points, the quality of the fit as indicated by the correlation coefficient ρ varies between 0.942 and 0.982 for the instruction caches and between 0.987 and 0.999 for data caches. These measures show that the straight line fit captures the majority of the behavior of the miss ratio with respect to cache size for Smith's data. The second derivative of the fitted function is positive, and the fitted function itself is a strictly convex function of the cache allocation x.

For the running example in this section, we use Smith's data for a 32 K-byte cache and a line size of 32 bytes The ρ measures for the curve fitted to line sizes of 32 bytes are 0.999 and 0.959, respectively, for data and instruction caches. A straight line fitted to the log/log data produces the following expressions for $M_I(x)$ and $M_D(x)$:

$$\log_{10} M_I(x) = 0.1177 - 0.11484 \log_2 (x), \quad (3)$$

$$\log_{10} M_D(x) = 0.5570 - 0.14223 \log_2 (x). \quad (4)$$

The logs of miss rates are base 10 and the logs of cache sizes are base 2 to simplify the interpretation of the display of the data. The precision required for the exponents in the model is greater than the precision of Smith's data because the curve fit is very sensitive to small variations of the exponent of the cache size. For the running example, we assume that this precision is available to us. Taking exponents in (3) and (4) produces

$$M_I(x) = 1.311x^{-0.11484 \log_2 10} = 1.311x^{-0.38151}, \quad (5)$$

$$M_D(x) = 3.606x^{-0.47249}. \quad (6)$$

Taking the derivatives of (5) and (6) yields

$$\frac{dM_I(x)}{dx} = 0.500x^{1.38151}, \quad (7)$$

$$\frac{dM_D(x)}{dx} = -1.704x^{-1.47249}. \quad (8)$$

The miss-rate derivatives in (7) and (8) are plotted in Fig. 3. Also, the argument of (8) in Fig. 3 is $C - x$ rather than x where C is 32K bytes. The crossing point of the curves is the point at which the miss-rate derivatives are equal. The optimal allocation occurs at the point where the I allocation $\hat{x}$ is approximately equal to 14432 (to the nearest multiple

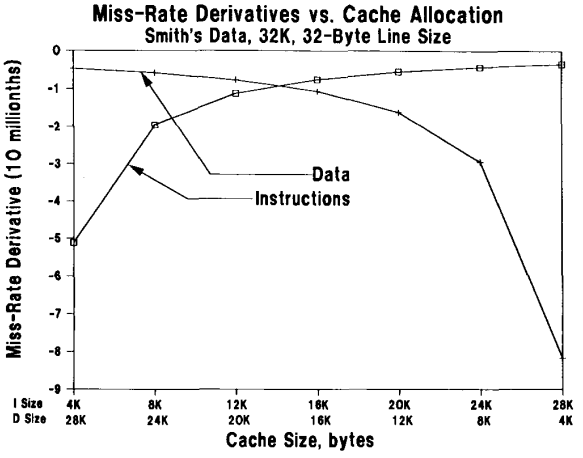


Fig. 3. Miss-rate derivatives as a function of cache allocation.

of 32). At this point the composite miss-rate is equal to the average of (5) and (6), which is

$$\begin{aligned}\text{Maximum Miss-Rate} &= (M_I(\hat{x}) + M_D(C - \hat{x}))/2 \\ &= (0.0340 + 0.0349)/2 \\ &\simeq 0.0344. \end{aligned} \tag{9}$$

This completes the discussion of the derivation of the optimal allocation of cache memory. In the next section we show that a conventional LRU-replacement policy has a most probable state that is not the optimal allocation of memory between the I and D references streams, but for the example data it produces very good allocations. It does extremely well for the trace-derived data in Section V.

III. The Most Probable State for LRU Replacement Policies

Consider a cache managed by a normal LRU replacement policy. What is the most probable allocation of cache memory between the I and D processes and what is the average allocation between these processes? Are these distinct? Is either allocation equal to the optimal allocation? The development of this section begins with a simplified version of LRU replacement that we call *modified-LRU replacement.* Subsequently, the model is embellished to approximate LRU replacement more closely. For the running example, the modified policy produces better allocations than LRU allocations in some cases, and poorer allocations in many more. In all cases among these examples, LRU allocations are very close to optimal, but are always suboptimal.

To obtain the results of interest, consider the behavior of a program in execution over a long time. We assume that the allocation of memory between data and instructions varies randomly, and is controlled by the statistics of the miss rates $M_I(x)$ and $M_D(x)$, where x is the amount of memory allocated to the I and D processes, respectively. We define the state of the cache to be the number of bytes allocated to the I process. That is, a cache of size C is in state x if x bytes are allocated to instructions and $C - x$ bytes are allocated to data.

The state x of cache is a time varying random variable. To characterize the state of a cache we need to be able to quantify state probabilities, a most-probable allocation, and an average allocation. If the random process is stationary, then we assume that it has been running long enough for transients to have died out. If the random process is nonstationary but varying slowly, as is the most likely case for cache allocations, we assume that measurements of averages and probabilities have significance if they are done during periods when the processes are locally stationary. This permits us to define for each state x, the probability of being in state x, which is denoted as $S(x)$. Given that the state probabilities are well-defined, a cache is in statistical equilibrium during a period of time if at each state x the rate of entering state x from state $x + 1$ by decreasing the cache allocation is equal to the rate of leaving state x by increasing the allocation and causing the cache to enter state $x + 1$. Fig. 4 illustrates this model.

Consider a replacement policy in which an instruction miss increases the number of instructions in the cache unless there are no data lines to replace, and similarly a data miss increases the number of data lines in the cache unless there are no instruction lines to replace. The item replaced is the least recently used item from among the items eligible for replacement. We call this policy the *modified-LRU* policy. An LRU-replacement policy does not distinguish between data and instructions. For fully associative caches, modified-LRU and LRU replacement choose to replace the same item if the globally least recently used item happens to belong to the other process. For set-associative caches, they also choose to replace the same item when all of the items of a set belong to one process. For this reason, the two policies are less distinguishable as associativity diminishes from full associativity to direct mapped. The two policies are identical for direct-mapped caches.

For the modified-LRU policy, the rate at which a cache of size C in state x increases the allocation of of instructions is $S(x)M_I(x)$, and the rate at which state $x + 1$ decreases its allocation of instructions is $S(x + 1)M_D(C - (x + 1))$. Because the rates are in balance at equilibrium, we have for each x in the interval $0 \leq x < C$,

$$S(x)M_I(x) = S(x+1)M_D(C - (x+1)). \tag{10}$$

The boundary conditions are $S(x) = 0$ for $x < 0$ and for $x > C$. The probability ratio of $S(x+1)/S(x)$ at equilibrium is given by

$$\frac{S(x+1)}{S(x)} = \frac{M_I(x)}{M(C - (x+1))} \tag{11}$$

for $0 \leq x < C$. Equations (10) and (11) are quite accurate for fully associative caches, and become less accurate as associativity diminishes. For direct mapped caches, these equations should not be used, because the modified-LRU policy becomes identical to LRU as described later in this section.

Since both $M_I(x)$ and $M_D(x)$ are nonincreasing functions of x, $M_D(C - x)$ is a nondecreasing function of x. Therefore (11) is a nonincreasing function of x. This and the fact that $S(x)$ is greater than unity at $x = 0$ together imply that $S(x)$ is

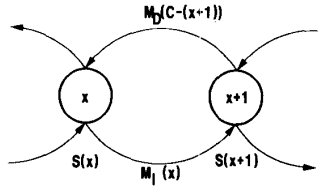


Fig. 4. State diagram and state transitions for cache allocations.

a unimodal function. That is, $S(x)$ grows monotonically with x to some peak value, and then diminishes monotonically with x for larger x. The peak occurs where the probability ratio in (11) falls below unity. The probability ratio might actually attain the value of unity at some state x, and it might be equal to unity over a contiguous interval of states. In this case, the peak value of $S(x)$ is attained at several contiguous values of x. Let $\hat{x}$ denote the largest value of x for which the following inequality holds:

$$M_I(x) \geq M_D(C-(x+1)). \tag{12}$$

Then $S(\hat{x})$ is the maximum state probability over all x.

For the example of the prior section, Fig. 5 plots the probability of state x as a function of x for the two different replacement policies. The curve that describes the state probabilities for (11) is labeled "modified-LRU replacement." The calculation that produced the curves in Fig. 5 is a standard calculation in which the probability of each state is expressed as a multiple of $S(0)$ and the value of $S(0)$ is set to a value that forces the sum of the probability coefficients over all states to be equal to unity.

Although the modified-LRU probability function in Fig. 5 shows a distinct peak, in the region of the peak the probability function is almost constant from state to state. Thus, at the peak, the following approximation holds because $S(\hat{x}) \approx S(\hat{x}+1)$:

$$M_I(\hat{x}) \simeq M_D(C-(\hat{x}+1)). \tag{13}$$

Equation (13) indicates that $M_I(x) \approx M_D(C-(x+1))$ at the most probable state, which is a statement regarding the equality of the miss-rate functions rather than a statement regarding the equality of the derivatives of the miss-rate functions. We can conclude that a replacement policy that produces an allocation that satisfies (13) is not optimal, in general, because this is not the equation satisfied by an optimal policy. Since an optimal allocation occurs when miss-rate derivatives are equal, a policy that produces an optimal allocation must somehow perform replacement as a function of observations of miss-rate derivatives. How this can be implemented efficiently is a subject of a separate paper by Stone, Thiebaut, and Wolf [17].

For the example problem, using the assumption that cache allocations are multiples of 32 bytes, we find that the most probable allocation state for modified-LRU replacement is cache state 13916 whereas the optimal allocation state is cache state 14432. The miss rate for state 13916 is 0.03443. The miss rate for the optimal state, 14432, is 0.03442. The probability of being in state x changes from 0.01632 to 0.01635 to 0.01637 to 0.01635 as x changes from 13856 to

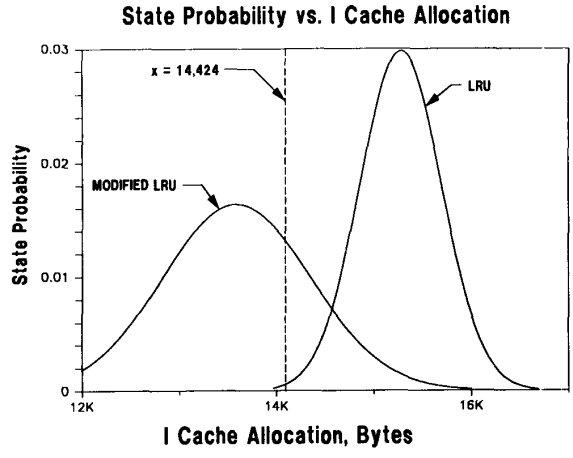


Fig. 5. State probabilities for two replacement policies.

13952 in steps of 32. Consequently, the probability is very nearly flat at this point, and the approximation in (13) is valid for the example. From the probability of being in state x as expressed in Fig. 5, we find the average allocation for that probability density to be 13921. Since allocations have to be multiples of 32 bytes when the line size is 32 bytes, the most probable state is the state closest to the average allocation for this particular example. If the probability density is narrowly concentrated as is the case in Fig. 5, the average allocation tends to be the same as the most probable allocation.

Because we are interested in LRU replacement, we embellish the model slightly to reflect the behavior of LRU replacement. For LRU replacement, the probability that an instruction replaces a datum by an instruction is equal to the probability that an instruction reference is a miss times the probability that the LRU item is a datum. Regardless of whether a cache is fully associative, set associative, or direct mapped, we assume that the probability that the item to be replaced is a datum is equal to the proportion of data lines in the cache. This same assumption was made by Strecker [18] for characterizing the cache-fill rate. (The assumption is validated in Section V.) Thus, the probability that x increases to $x+1$ is given by

$$\text{Prob}[x \text{ increases to } x+1] = (1-\frac{x}{C})S(x)M_I(x), \tag{14}$$

and, conversely, in state $x+1$, the probability that the allocation decreases to x is

$$\text{Prob}[x+1 \text{ decreases to } x] = \left(\frac{x+1}{C}\right)S(x+1)M_D(C-(x+1)). \tag{15}$$

When we equate these probabilities, which is the equilibrium condition, we find

$$\frac{S(x+1)}{S(x)} = \frac{(1-\frac{x}{C})M_I(x)}{(\frac{x+1}{C})M_D(c-(x+1))}. \tag{16}$$

The monotonicity discussion that follows (11) holds as well for (16). Thus the ratio in (16) is a nonincreasing function of

TABLE I
CACHE ALLOCATION DATA: COMPARISON OF ANALYTICAL AND SIMULATION MODELS

r Value	Optimal Policy		Model or Simulation	Modified-LRU Policy			LRU Policy		
	I-Cache Size	Miss Rate		*I*-Cache Size	Miss Rate	Standard Deviation	*I*-Cache Size	Miss Rate	Standard Deviation
0.2500	22,080	0.0321	Model	30,815	0.0404	350.3	23,127	0.0322	388.0
			Simulation	30,820	0.0403	305.8	23,045	0.0324	368.0
0.333	20,608	0.0328	Model	29,355	0.0387	456.5	21,719	0.0329	403.1
			Simulation	29,293	0.0384	324.0	21,529	0.0331	351.1
0.500	18,368	0.0337	Model	25,613	0.0363	626.9	19,571	0.0337	419.0
			Simulation	25,942	0.0364	494.1	19,379	0.0340	260.0
0.750	16,064	0.0342	Model	19,358	0.0347	761.8	17,294	0.0343	427.5
			Simulation	19,704	0.0344	321.7	17,169	0.0346	264.8
1.000	14,432	0.0344	Model	13,919	0.0344	779.6	15,645	0.0345	428.4
			Simulation	14,619	0.0344	373.0	15,530	0.0347	358.6
2.000	10,656	0.0340	Model	3,884	0.0374	525.0	11,773	0.0340	413.1
			Simulation	4,009	0.0372	320.2	11,679	0.0343	376.0
3.000	8,704	0.0333	Model	1,515	0.0404	340.0	9,704	0.0334	393.9
			Simulation	1,575	0.0402	319.0	9,487	0.0335	378.6
4.000	7,456	0.0327	Model	764	0.0423	240.1	8,368	0.0327	376.7
			Simulation	769	0.0425	238.7	8,186	0.0329	364.9

x, and the most probable allocation for the LRU policy occurs at the value $\tilde{x}$, which is the greatest value of x for which the following inequality holds:

$$\left(1 - \frac{x}{C}\right) M_I(x) > \left(\frac{x+1}{C}\right) M_D(C - (x+1)). \qquad (17)$$

Because (17) is different from (12), the most probable allocation for LRU replacement is not the same as the most probable allocation for the modified-LRU replacement. We have no basis to judge which replacement strategies produce better allocations in general, but for the running example, the modified-LRU replacement policy produces a slightly better allocation.

For the running example, the most probable state for LRU replacement is 15 648, and the average allocation is 15 645. The miss rate for the most probable allocation is 0.03448 as compared to 0.03442 for the optimal allocation. These results together with simulation results are summarized in Table I. The table shows, respectively, the optimal allocation, the average modified-LRU allocation, and the average LRU allocation for various values of the ratio r. The most probable allocations are very nearly equal to the average allocations in all cases, and have been omitted from the table. With each allocation is the miss ratio at that allocation. The average miss ratios are very nearly equal to the tabulated miss ratios in every case, and have also been omitted from the table. The standard deviation shown in the table gives some idea of the width of the peak of the probability density function for the allocation.

The simulation data in Table I are drawn from simulations of the Markov process model of the cache allocation. The simulations are based on the statistics collected for 125 000 time steps after the system reached the expected steady-state allocation. They controlled the miss-rate function as a function of allocation, and they also rigidly alternated between I and D processes. Thus the simulations confirm the analytical model, but do not necessarily model actual cache behavior.

The simulation results show what happens when two processes compete for cache and those processes have miss rates expressed by the functions $M_I(x)$ and $M_D(x)$. The LRU simulation used the weights x/C and $(1 - x/C)$ to control the probability of increasing or decreasing the cache allocation. The modified-LRU simulation forced the instruction allocation to increase on an instruction miss and forced the data allocation to increase on a data miss, and thus models the behavior of a fully associative cache, and approximates the behavior of set-associative caches. The observed most probable and average allocations for both LRU and modified-LRU simulations are within a standard deviation of the corresponding model allocations for the two policies obtained from the derivations above.

For the running example, because the optimal allocation occurs very close to the point of equal allocation of storage between D and I processes, the weights x/C and $(1 - x/C)$ are very nearly equal, and thus the weights do not shift the position of $\tilde{x}$ far from the position of $\hat{x}$. Fig. 6 gives a plot of relative miss rate as a function of allocation for the running example, and we see that miss rate is not very sensitive to the exact allocation. The instruction allocation can be anywhere between 8K and 20K, and still produce a miss rate that is not greater than 10% above the minimum. Since the absolute miss-rate is less than 5% in this range, a difference of less than 10% in miss-rate contributes to a performance difference no more than 0.5% across this range of allocations.

Recall that r is the ratio of data references to instruction references. The running example assumes $r = 1$. LRU produces better allocations for most values of r in Table I with the exception being $r = 1$.

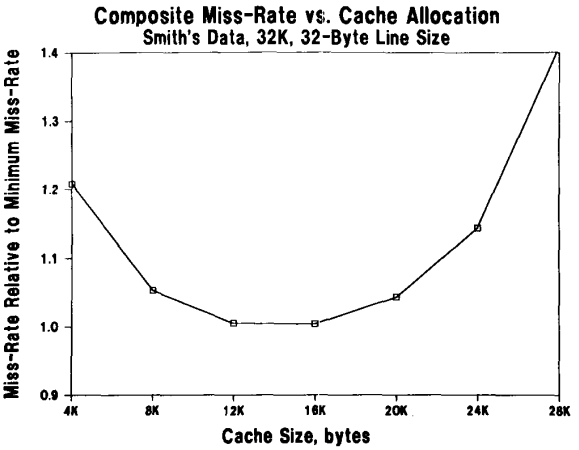


Fig. 6. Relative miss-rate versus cache allocation.

The LRU policy introduces a weighting factor that tends to force allocations to be more nearly equal than does the modified-LRU policy. In our example, for $r = 1$ the correction factor moves the LRU allocation too close to an equal allocation, and leaves the modified LRU allocation closer to the optimal allocation. For the other values of r, the allocations produced by the modified-LRU policy tend to be fairly extreme, and the LRU policy introduces a correction factor that produces a more nearly balanced allocation that happens to be closer to the optimal allocation.

In general, we find that LRU replacement produces near-optimal allocations provided that:

1) $\tilde{x}$ is close to $C/2$ to reduce the effects of weights x/C and $(1 - x/C)$ in (17), and
2) the miss rate is not very sensitive to the cache allocation in the vicinity of the optimum allocation.

Both of these characteristics hold for the running example, and it is quite possible that they hold for most workloads. However, the near-optimality of the LRU allocation has not been thoroughly investigated for real workloads and real cache designs, so the question is still a matter of future research.

It is rather curious that a small change in the LRU replacement policy can produce a cache allocation that is markedly different than that produced by LRU replacement. Although we have shown that LRU replacement is not optimal, the point of this study is to determine why LRU replacement appears to be so close to optimal.

Occasionally, system designers consider an alternative of allocating a fixed amount of cache to instructions and a fixed amount to data, instead of letting data and instructions compete for cache. If the sole criterion for the allocation is to minimize the long-term miss rate, then the results above indicate that a fixed partition is probably not a good idea because, in practice, cache memory automatically becomes partitioned in a near-optimal way. However, other considerations may force cache to be partitioned, such as to support simultaneous access to data and instructions. The fact that an optimum allocation of memory occurs when the weighted miss-rate derivatives are equal should be helpful to designers of such caches.

IV. Cache Allocation Dynamics

Thus far, we have shown that that LRU replacement can produce near-optimal allocations. This suggests that the LRU policy is a good one to use in general for managing a cache. But LRU is not optimal, and there may some advantages in using a non-LRU policy. What are those advantages?

One possible reason for managing cache with a policy other than LRU replacement is to avoid the transient in cache allocation when the components of a composite address-reference stream change their characteristics. For example, the data process may change from low locality to high locality. As locality characteristics change, the optimum partition of cache between data and instructions changes, and the allocation of an LRU-managed cache moves to a new, hopefully, near-optimum allocation. If by some external means the cache-management algorithm is given the new steady-state allocation, it may be able to impose that allocation on cache immediately, and avoid the transient that a cache would normally experience.

This section shows that during the initial phase of a transient, cache allocation changes rapidly if the new allocation is quite different from the present allocation. The rate of change slows considerably as the current allocation comes close to the steady-state allocation, and in this region it may be possible to improve performance by using another strategy to hasten convergence. We outline a combinatorial approach for finding the steady-state allocation directly, which, if possible to implement in practice, can eliminate the transient.

The question at hand, then, is what is the dynamic behavior of memory allocation of an LRU-managed cache? We start by modeling the dynamics of modified-LRU replacement because it is somewhat simpler, and then embellish the model to deal with LRU replacement.

If we treat x, the amount of memory allocated to the I process, as a state variable of the cache then the rate at which the state of the cache changes is given by

$$\frac{dx}{dt} = \text{Rate of increasing } x - \text{ Rate of decreasing } x. \quad (18)$$

For a cache managed by the modified-LRU replacement policy, we have the following differential equation that describes the dynamics of the cache allocation:

$$\frac{dx}{dt} = M_I(x) - M_D(C - x). \quad (19)$$

Strecker's model of cache dynamics captures the fill rate of a process in the absence of competition. Thus, his differential equation uses only the first term of (19), and his characterization of a cache transient is very different from the dynamics of cache allocation in the presence of competition.

For the running example, we substitute (5) and (6) into (19) and we find

$$\frac{dx}{dt} = 1.311x^{-0.38151} - 3.606(C - x)^{-0.47249}. \quad (20)$$

There is no neat closed-form solution for this differential equation. But it can be solved numerically, and its solution is

plotted in Fig. 7. To find the dynamics for LRU replacement, we have to include the weights x/C and $1 - x/C$. Equation (19) becomes

$$\frac{dx}{dt} = \left(1 - \frac{x}{C}\right)M_I(x) - \left(\frac{x}{C}\right)M_D(C - x). \quad (21)$$

After substituting (5) and (6) in (21), we obtain

$$\frac{dx}{dt} = 1.311\left(1 - \frac{x}{C}\right)x^{-0.38151} - 3.606\left(\frac{x}{C}\right)(C - x)^{-0.47249}. \quad (22)$$

The solution to this equation is also plotted in Fig. 7. For comparison purposes, the dynamics produced from the simulations are also plotted, and they track the numerical predictions reasonably well. The simulations, of course, are subject to statistical variations that are not captured by the dynamic model. For this simulation, as time increases beyond the right edge of the graph, the allocation varies over a range centered on the long-term asymptote.

Note that both replacement policies produce rapid convergence when the current value of x is far from steady state but the convergence becomes much slower as x nears the steady-state region. The maximum convergence rate is $rM_D(x)/(1+r)$ and $M_I(x)/(1+r)$ when D and I, respectively, are the minority and majority cache owners, and D references occur r times as frequently as I references. This follows because the minority owner increases cache occupancy with every miss and the majority owner does not alter occupancy on its misses.

If the current steady-state allocation is not optimal, can a cache manager move quickly to an optimal allocation to avoid the transient in allocation? We assume that a cache manager is given the parameters of the competing processes, and the goal is to find the optimal allocation without solving (16) numerically. The following discussion presents an efficient solution to this problem. It treats the more general situation in which there are N processes competing for cache, not just two processes. It relies on the fact that at an optimal allocation, all of the miss-rate derivatives are equal. This is a generalization of the result given in (2), and it is proved in Thiebaut, Stone, and Wolf [22], but it follows directly from the arguments used to derive (2) for two competing processes.

To introduce the allocation problem for N competing processes, suppose they are to use a cache of size C. Let M_i denote the miss rate for process i as a function of allocated cache. Suppose that each process i generates P_i references, giving a total of

$$T = \sum_{i=1}^{N} P_i \quad (24)$$

references in all. If each process i is allocated a portion of the cache of size C_i, then the overall miss rate is then given by

$$\sum_{i=1}^{N} P_i M_i(C_i)/T. \quad (25)$$

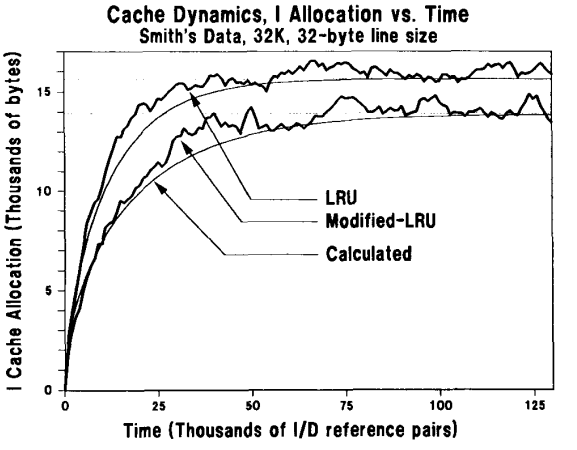


Fig. 7. Dynamic behavior of computed and simulated cache allocations.

We wish to minimize this function subject to the constraint that

$$\sum_{i=1}^{N} C_i = C \quad (26)$$

by giving or taking cache away from a competing process. The least possible overall miss rate occurs when miss-rate derivatives with respect to cache size are equal.

We assume that the cache miss-rate for each process is a (decreasing) strictly convex function of cache size. Thus the functions $F_i(C) = P_i M_i(C)/T$, the fraction of the total miss rate attributed to process i, are also (decreasing) strictly convex functions of cache size C allocated to a process. We wish to minimize the sum of N such functions. The values of the functions $F_i(x)$ are computed for discrete integer values of x in the range $0 \leq x \leq C$.

This problem fits into the category of separable convex resource allocation problems. A greedy algorithm due to Fox [4] solves such problems in time $O(N + C \log N)$. While we base our discussion on this work, we note that the subsequent results due to Galil and Megiddo [1979] and Frederickson and Johnson [6] are more efficient, the latter having complexity $O(\max(N, N \log(C/N)))$. This bound was shown by Frederickson and Johnson to be optimal to within a constant multiplicative factor. We would therefore recommend that an implementation of this approach actually be based on the Frederickson and Johnson algorithm.

Ibaraki and Katoh [10] treat resource allocation problems of this type in depth, and provide excellent background for the optimization problem treated here. Also note that generalizations of the optimization algorithm described above have been solved and employed by Tantawi, Towsley, and Wolf [19], by Wolf, Dias, and Yu [24], and by Wolf, Iyer, Pattipati, and Turek [25] to solve other optimization problems in computer science.

Define for each $1 \leq i \leq N$ and each $1 \leq j \leq C$ the "marginal return" $g_{i,j} = F_i(j-1) - F_i(j)$. This function is essentially the first-difference discrete analog of the negative of the miss-rate derivative for process i with cache allocation j. Note that for each i, the values $g_{i,1}, \cdots, g_{i,C}$ are positive

and nonincreasing because the miss rate of the process is a decreasing convex function of allocated cache. The trick is to allocate cache, chunk by chunk, to the collection of processes to maintain the miss-rate derivatives as equal as possible as cache is added. Fox's algorithm can be defined inductively as follows:

1. *Initialization*: Set $C_1 = \cdots = C_N = 0$.
2. *Induction Step*: Given the kth assignment of values to the C_i, and noting that it satisfies

$$\sum_{i=1}^{N} C_i = k, \tag{27}$$

the induction step produces a new assignment satisfying

$$\sum_{i=1}^{N} C_i = k + 1 \tag{28}$$

by increasing C_i to $C_i + 1$ for some index i. The step selects the first value of i such that

   a. $C_i < C$, and
   b. g_{i,C_i+1} is maximum.

3. That is, the step finds the value of the marginal return function g_{i,C_i} as each C_i is incremented in turn, and then retains the value of C_i for which the marginal return is greatest. All other C_i values remain unchanged.

After C steps through the algorithm, the values $C_1, \cdots, C_N$ form an optimal assignment of cache to the competing processes such that process i receives C_i units of cache, and the total amount of cache allocated is C units.

The algorithm amounts to computing a Cth smallest number in an $N \times C$ matrix whose rows are nondecreasing. This is a special case of the so-called "selection problem," and is solved, as noted above, with considerably greater speed using the algorithm proposed by Frederickson and Johnson [5]. Details may be found there or in Ibaraki and Katoh [10].

This completes the discussion of allocation strategies for interlaced processes. The next section presents the validation of the several assumptions in the model as obtained from actual trace data.

V. Experimental Validation

The model presented in Sections II through IV relies on a number of assumptions that need to be validated. This section shows that a trace-driven simulation behaves almost precisely as predicted by the model. While Smith's data are useful in examples, the published data were created by taking a composite of many workloads and cache structures. The trace-driven validation in this section demonstrates that the predictions of the model hold true for the cache behavior for a specific workload and cache structure.

The trace used comes from a single processor in a two-processor IBM System/370 architecture complex executing a multiprogrammed workload. The workload is commercial, and contains a mixture of user code and operating-systems code. The trace contains over 18 000 000 references in total. It was created with the intention of being representative of workloads encountered in typical environments and has been used internally in IBM as a representative of such workloads.

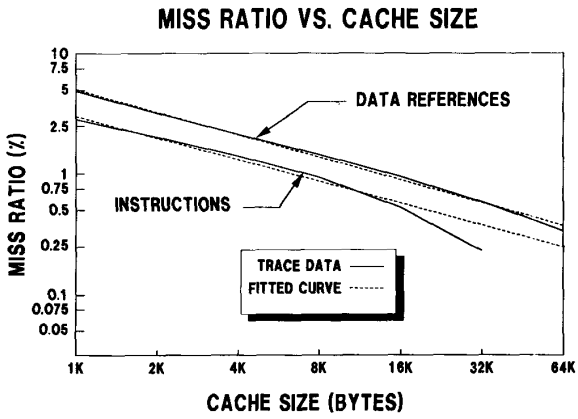


Fig. 8. Miss-ratios as a function of cache size for trace data.

The model requires functions $M_I(x)$ and $M_D(x)$. To make the validation as similar as possible to the running example, the cache structure was chosen to be 32 K-bytes, 4-way set-associative, with a line size of 32 bytes. The variation of miss rate with cache size was obtained by varying the number of sets while fixing the line size to 32 bytes and the associativity to 4-way.

The data that characterizes these functions appears in Fig. 8. The analysis of miss rates was performed by simulating the I or the D process in isolation. During the period when the cache was not fully initialized, the trace simulation filled the cache without recording miss-rate statistics. The simulation was continued without recording statistics until the reference count reached the next multiple of 100 000, and at that point the recording of miss-rate statistics began. This eliminated the bias caused from initialization misses in the simulation. Because the I process did not touch all of the lines in a cache of size 32 K-bytes, this cache was declared to be filled when all but 10 lines (out of 1024) were initialized. In this case, 6.4 million references were traced after the initialization point, and in all other cases the number of references traced after initialization exceeded 8 million.

Note that the miss-rate curve for data misses is well approximated by a straight line on a log/log scale. The miss-rate curve for instruction misses is straight for most of its length, but the data point for 32K lies below the straight-line trend. To fit the curves to the data shown, we used all of the points for data caches and dropped the point for the instruction cache of size 32K. The instruction occupancy does not reach 32K in normal operation, and, hence, the region of interest for instruction occupancy is the region of the curve through which the straight-line trend is drawn. The equations for the fitted curves in Fig. 8 are $M_I(x) = 1.996x^{-0.60147}$ and $M_D(x) = 4.072x^{-0.63165}$. The respective ρ's are 0.994 and 0.984, which indicate that the fits are excellent.

The total number of references observed was approximately 16 million, and the ratio r was 0.9606. The model for LRU replacement embodied in (16) predicts an average allocation

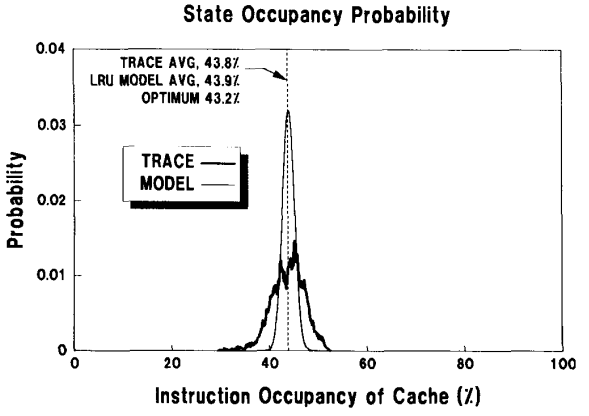


Fig. 9. Cache occupancy probability for trace and model data.

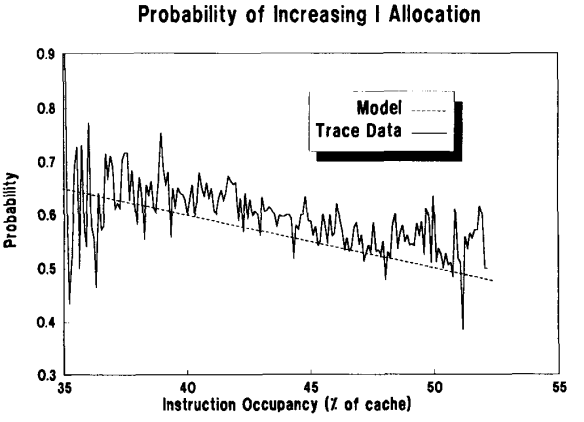


Fig. 10. The probability of increasing instruction occupancy as a function of I cache allocation.

of 14 392 bytes and standard deviation of 400 bytes as plotted in Fig. 9. This compares with an observed average allocation of 14 365 bytes and a standard deviation of 1100 bytes for the trace data. The optimum allocation for these parameters is 14 150 bytes. The model for modified-LRU replacement produces an average allocation of 11 364 with a standard deviation of 564 bytes.

Note that the model for LRU replacement and the observed behavior of the cache give excellent agreement in their average values. The model variance is somewhat less than the observed variance of cache occupancy. Also note that LRU replacement produces an allocation that is essentially optimal.

The wider variance of the trace-driven process is attributed in part to long sequences of references of one type, particularly to long sequences of data references. Nevertheless, the predicted average allocation for LRU replacement lies almost directly where the actual average allocation appears, because the trace-driven allocations lie in a narrow range, even though that range is somewhat larger than the allocation range of the model.

The respective miss rates for the data depicted in Fig. 9 are 0.0074 observed for the trace, 0.0072 for the LRU model, 0.0072 for the optimal allocation, and 0.0073 for the modified LRU-replacement policy.

Equation (16) reflects the assumption that the probability of increasing an allocation is proportional to the fraction of cache currently occupied by the competing process. The trace-driven experiment gathered data that confirmed this hypothesis as well. This is shown in Fig. 10. The data plotted shows the actual probability of increasing allocation as a function of current allocation, and the straight line shows the model. For occupancies less than 36%, there were fewer than 10 misses observed in each state, and thus the data are very noisy and not statistically significant. Above 36%, the data plotted in the graph follow the model closely except that they fall slightly above the model. However, the error in modeling is similar for data and instructions, and these errors tend to cancel in (16) because one error is a factor in the numerator and the other is a factor in the denominator. Hence, the model is an accurate predictor of cache allocation for LRU replacement in spite of the small inaccuracy. At this writing, it is not known what phenomenon causes the discrepancy between the model predictions and the observations shown in Fig. 10.

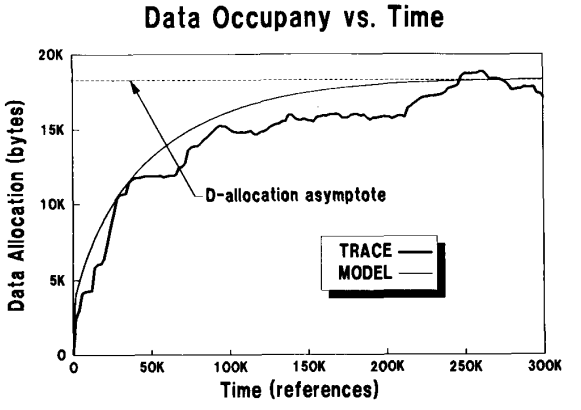


Fig. 11. The dynamics of data occupancy for the trace experiment.

Fig. 11 shows the dynamics of data cache occupancy starting with a minimum allocation of data (10 lines). The dotted curve is the numerical solution of the dynamic model, and the horizontal line is the asymptote, which is the average allocation for LRU replacement. Note the good agreement between the mathematical model and actual trace behavior. The data were gathered by simulating a cache on a trace, using just the instruction references from the trace and ignoring the data references until 1014 out of 1024 lines were in the cache. Then the simulation treated both instruction and data references, and recorded the data occupancy in cache as a function of time measured in references.

The validation in this section demonstrates that cache occupancy of instructions and data can be predicted accurately from parameters for the individual processes, and that for this particular experiment, LRU replacement produced an allocation that was essentially optimal. The next section treats the case in which references by competing processes are not tightly interlaced.

VI. Multiprogramming Replacement Strategies

In this section we revisit the allocation problem, but this time consider what happens when processes compete sequentially for cache instead of being tightly intertwined. An example of the problem occurs in multiprogrammed computers in which each process obtains exclusive access of the processor for a time quantum of length Q. For our purposes, we assume that there are two processes with miss rates $M_1(x)$ and $M_2(x)$, respectively. Each process makes precisely Q memory accesses in its time quantum.

How good is an LRU replacement policy in this situation? Because one process runs after the other under an LRU replacement policy, while process 2 is running each cache miss discards items older than any items touched by process 2 in its current quantum. This would usually be items that belong to process 1. Conversely, each miss during the execution of process 1 discards an item belonging to process 2 whenever this is possible.

While this strategy is basically good, let us examine the quality of the decision to replace an item belonging to process 1 when process 2 suffers a cache miss. The miss rate of process 2 improves by an amount $dM_2(x)$, but, if we assume that process 1 will reload the item displaced with probability p, then for each item belonging to process 1 that is discarded, a fraction p of a miss will be recorded when process 1 resumes control.

If a cache is small compared to the working set of process 2, the gain in adding a new item belonging to process 2 outweighs the cost of the miss that will be incurred when process 1 regains control of the machine. Since LRU replacement continually adds to the allocation for process 2, for small caches LRU replacement is the algorithm of choice.

But suppose that the cache is very large, and is large enough to hold the full working set of process 2 as well as lines belonging to process 1. The marginal reduction in misses obtained by increasing the cache allocation of process 2 is the product of the incremental change in miss rate due to the increased allocation times the number of references remaining in the time quantum. That is, if q references remain in the time quantum, then we expect to save $qdM_2(x)/dx$ misses by increasing the allocation of memory to process 2. But we must also account for the p additional misses when changing back to process 1, because process 1 will reload the cache line displaced by increasing the allocation of process 2 with probability p. Thus the net reduction in misses is equal to $qdM_2(x)/dx - p$. The threshold $q(x)$ is defined to be the remaining time at which the marginal gain is 0, and this is given by

$$q(x) = \frac{p}{dM_2(x)/dx}. \tag{23}$$

If the remaining time in a quantum is less than $q(x)$ for a cache allocation of x, then the replacement policy should not increase the cache allocation of process 2. The replacement policy should replace the least recently used eligible line of process 2, and retain the lines of process 1 currently in cache. If the remaining time exceeds $q(x)$, then the replacement policy should replace the least recently used lines of process 1, and thereby increase the allocation of process 2.

Clearly, if the marginal gain is very small or the time quantum is nearly over, then it is better to leave items of process 1 in cache because they will be likely to be referenced again in the near future. Small processor caches, typical of those in computers available through the end of the 1980's, rarely hold much more than the working set of a process. In such caches very little of the last run process is left in cache as the currently running process nears the end of its quantum. A non-LRU policy cannot easily benefit by retaining items belonging to process 1 as the time quantum for process 2 runs out. This situation changes when caches become much larger than the working sets of typical processes. With caches in the early 1990's approaching 1 megabyte in size, the possibility of improved performance makes the non-LRU policy an attractive candidate for closer study.

As an example of the use of $q(x)$ in a replacement policy, consider Fig. 10, which plots $q(x)$, the inverse of the miss-rate derivative, for the I-cache miss-rate derivatives plotted in Fig. 3. Fig. 12 shows what the threshold would be for a hypothetical workload and cache structure whose miss rates fit Smith's data for a line size of 32 and for $p = 1$. This is the threshold below which additional cache should not be allocated to the running process. In the middle range of the allocation, the remaining references in the time quantum should be on the order of 1 million for a process to continue to receive additional cache memory. Assume that a process governed by the miss-rate function of Fig. 2 initiates execution and quickly builds a cache allocation of 4K bytes. If the remaining quantum has 100 000 references or more, the allocation should increase, and in fact, under an LRU replacement policy it increases quickly. At some point depending on the time remaining, the miss-rate derivative is sufficiently small that no additional cache should be allocated to the running process. In Fig. 12, the additional allocation should stop when the current allocation is roughly half the cache if less than 1 million references remain. When the allocation is roughly 28K, additional cache allocation should cease if less than 2 million references remain in the quantum. If the initial time quantum is 1 million references, a non-LRU replacement policy will limit the cache allocation to roughly half of cache.

If an interrupt process takes over the cache for a brief period, a non-LRU replacement policy will grant the process only a small initial allocation, and sharply curtail its acquisition of cache beyond this amount because its quantum is so short. This tends to give the interrupt process a small region of cache to hold its working set, and to retain the working set of the interrupted process in anticipation of the future use of that portion of cache memory.

VII. Multilevel Caches

Next, let us consider a common multilevel cache design, in which caches are arranged in hierarchical fashion. The fastest and smallest cache is at level one, the next fastest and smallest is at the level two, and so on. An example of a study of this type of hierarchy appears in Baer and Wang

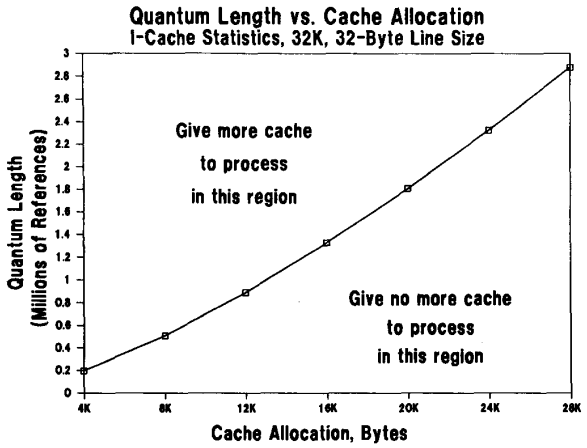


Fig. 12. The quantum-length threshold as a function of allocation.

[1]. In this section we shall extend our solution of the optimal cache-allocation problem to handle competing processes in multilevel hierarchies. An early formulation of a different cache-allocation problem for a cache-memory hierarchy is due to Chow [3].

For simplicity of exposition, we shall again assume that we are given the address streams of two competing processes, one corresponding to instructions and one to data, and that these occur with equal frequency. Furthermore, we shall attempt to allocate memory optimally for the two processes in each level of a two-level cache hierarchy. The extension to arbitrary numbers of competing processes, arbitrary relative frequencies, and arbitrary numbers of cache levels is straightforward. We shall also assume that the line sizes of the caches at each level are equal.

An illustration of a two-level cache hierarchy is given in Fig. 13. The first-level cache sees the full address stream produced by the processor, and passes only its misses to the second-level cache. The second-level cache responds to the residual stream, and sends to main memory only the references of its input stream that produce misses in this cache. Assume that the first-level cache has size c and the second-level cache has size C. In each level of the cache, we would like to use the results of the previous sections to allocate memory optimally. However, in order to deal with the second-level cache, we need to understand how the size of the first-level cache affects the miss rate in the second-level cache. Fortunately, this is quite easy, and we proceed as follows.

Suppose that a fraction $\alpha = \alpha(c, C)$ of the lines appearing in the first-level cache also appear in the second-level cache. Then the total number of distinct lines in the cache is given by $C + (1 - \alpha)c$. The global miss-rate of the combined cache on the full input stream is thus $M(C + (1 - \alpha)c)$, while the local miss-rate of the first-level cache alone is $M(c)$. (The terms *local* and *global* miss rates are due to Hennessy and Patterson [8].) A miss in the combined cache occurs when the reference misses in both levels. In other words, if $M(c, C)$ denotes the miss rate of the second-level cache with respect to the input stream that it receives, then $M(c)M(c, C) = M(C+(1-\alpha)c)$,

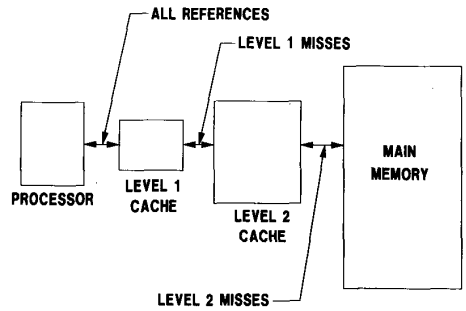


Fig. 13. A memory system with two levels of caches.

or, equivalently,

$$M(c, C) = M(C + (1 - \alpha)c)/M(c). \tag{29}$$

In general, the lines of the first-level cache are nearly completely contained in the much larger second-level cache, so that $\alpha \simeq 1$, or $1 - \alpha \simeq 0$. (In fact, we could force α to be 1 by imposing a cache inclusion condition.) Thus, the approximation

$$M(c, C) \simeq M(C)/M(c) \tag{30}$$

is very good.

Now let t denote the access time for a hit in the first-level cache of size c, T denote the total access time for a hit in the second-level cache of size C, and τ denote the total access time for main memory. By definition we have $t < T < \tau$, and in practice we can assume that $t \ll T \ll \tau$. We can also assume that $c \ll C$. For any partition $c = c_1 + c_2$ of the first-level cache, we can compute local miss rates $M_1(c_1)$ and $M_2(c_2)$. Given this first-level partition and any partition $C = C_1 + C_2$ of the second-level cache, we can compute global miss rates $M_1(c_1, C_1)$ and $M_2(c_2, C_2)$. Thus, the overall access time is given by

$$\begin{aligned} \boldsymbol{T}(c_1, c_2, C_1, C_2) &= \frac{1}{2}\left[\sum_{i=1}^{2}(1 - M_i(c_i))t + \sum_{i=1}^{2} M_i(c_i)(1 - M_i(c_i, C_i))T \right. \\ &\quad \left. + \sum_{=i1}^{2} M_i(c_i)M_i(c_i, C_i)\tau\right]. \end{aligned} \tag{31}$$

We want to minimize this function subject to the constraints $c_1 + c_2 = c$ and $C_1 + C_2 = C$. This can be done by repeated application of the algorithm given in Section IV, but there is a simplification that greatly limits the search effort. The simplification relies on the presence of good lower and upper bounds.

To derive a tight upper bound, note that in the region where $\alpha_i(c_i, C_i) \simeq 1$, (26) reduces to

$$\boldsymbol{T}_1 = t + \frac{1}{2}(T - t)\sum_{i=1}^{2} M_i(c_i) + \frac{1}{2}(\tau - T)\sum_{i=1}^{2} M_i(C_i). \tag{32}$$

In fact, $\boldsymbol{T}_1$ provides an upper bound for $\boldsymbol{T}$ over the entire feasible region, since $\tau > T$. Minimizing (27) is easy, since

the first summand is a constant, and $t < T < \tau$ implies that the second and third summands are positive. The problem simply decouples into two disjoint separable convex resource-allocation problems, one for the second summand and one for the third. Both problems are solved by the technique given in Section IV. Since, in practice, α will almost surely be close to 1 in this region, the optimal solution for (27) derived in this manner will be an excellent approximation to the optimal solution for (26).

The lower bound is obtained by investigating the behavior of (26) under the assumption $\alpha = 0$. Equation (26) reduces to

$$\boldsymbol{T}_2 = t + \frac{1}{2}(T-t)\sum_{i=1}^{2} M_i(c_i) + \frac{1}{2}(\tau - T)\sum_{i=1}^{2} M_i(C_i + c_i). \tag{33}$$

$\boldsymbol{T}_2$ differs from $\boldsymbol{T}$ only in its third summand. We claim that $\boldsymbol{T}_2$ provides a lower bound for $\boldsymbol{T}$ over the entire feasible region, since $\tau > T$. (This optimization problem does not quite decouple, but given any solution to the problem associated with the second summand, optimal or not, one can solve the problem with respect to the third summand, which is again a separable convex resource-allocation problem.) The corresponding exact reduction associated with $\boldsymbol{T}$ in (26), namely,

$$\boldsymbol{T} = t + \frac{1}{2}(T-t)\sum_{i=1}^{2} M_i(c_i) + \frac{1}{2}(\tau - T)\sum_{i=1}^{2} M_i(C_i + (1-\alpha_i(c_i, C_i)), c_i), \tag{34}$$

differs from $\boldsymbol{T}_1$ and from $\boldsymbol{T}_2$ only in its third summand. This summand does not give rise to a convex optimization problem, but is tightly bound by two problems that are. By utilizing these bounds one can restrict the search for the overall optimal solution to (26) to a neighborhood very close to the optimal solution of (27).

As an example of the application of this theory, we again use the miss-rate functions shown in Figs. 1 and 2. Specifically, we use an instruction stream whose miss rate is described by (5) and a data stream whose miss rate is described by (6). Assume a line size of 32 bytes, with $t = 1, T = 10$, and $\tau = 20$. Assume that the first-level cache is of size 8K bytes, so that $c = 256$ lines, and the second-level cache is of size 64K bytes, so that $C = 2048$ lines. The optimal partitioning occurs at $c_1 = 104$ lines, $c_2 = 152$ lines, $C_1 = 924$ lines, and $C_2 = 1124$ lines, and yields $\boldsymbol{T} = 1.8171$. Fig. 14 shows a contour plot of the various partitioning choices. By contrast, a cache simulation shows that LRU yields a time of $\boldsymbol{T} = 1.8189$, so that, once again, LRU is proven to be quite robust. The contour line for the LRU average access-time is shown as the innermost contour line in Fig. 14.

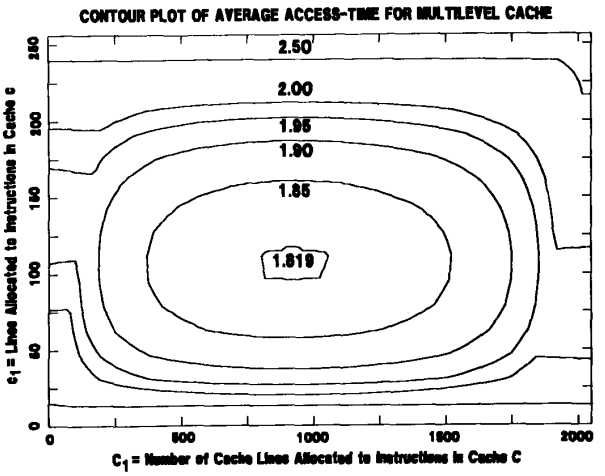


Fig. 14. A contour plot of average access-time as a function of cache allocations in a multilevel cache.

VIII. Summary and Conclusions

The major finding of this study is that the LRU replacement policy is good, but not optimal. We have shown that a modified-LRU replacement produces different allocations of data and instructions, and in some cases the modified policy produces marginally better performance than the LRU policy. For tightly interlaced address streams generated by two competing processes, the examples in the paper indicate that the miss rates produced by LRU replacement are only negligibly different from the miss rates produced by an optimal cache partition. Because all of the evidence for the quality of LRU is based on examples rather than on firm bounds, we cannot be sure that LRU replacement is as near-optimal in practice as it has turned out to be for the examples used in the paper. Until bounds on the near-optimality of LRU replacement are available, it is an open question whether there is a realizable alternate policy that produces better allocations. While we cannot justify the implementation of non-LRU replacement policies for the examples given in the paper, it may be necessary to partition cache for other reasons. If so, the techniques of the paper are useful in determining the relative sizes of the partitions by showing how to achieve the least miss rate for partitions within a fixed chip area, a fixed board area, or for a fixed total cost.

Research questions that remain open include the following:

1) Is there a simple bound on the near optimality of LRU replacement with regard to its ability to allocate cache between data and instruction processes?
2) Is there a simple replacement policy that actually achieves the optimum allocation between data and instructions?
3) How can the transient associated with cache allocation be reduced without using detailed knowledge about the miss-rate functions?
4) For multiprogrammed systems, what is a practical means for implementing a limit on cache allocation? Does such a scheme produce sufficient benefit to justify its implementation?
5) For cache in a multiprogramming environment, it is clear that there is no justification for using a strategy other than LRU when there is little possibility for retaining lines belonging to other processes in the cache. As caches become large, this is no longer true. But how

large do caches have to be for non-LRU replacement to be worthwhile?

Although LRU replacement is the cache-management algorithm of choice today, the paper has pointed out potential opportunities for improving performance by implementing a non-LRU strategy. On the other hand, the paper has shown that an LRU strategy is robust, near-optimal, and difficult to outperform in practice. Before adopting a non-LRU strategy, it is essential to explore the questions above to determine what strategies are most useful and what their benefits might be.

ACKNOWLEDGMENT

The authors are indebted to the referees for their perceptive comments and suggestions which led to material improvements in this paper. The authors would also like to thank Dr. T. Puzak of the IBM T. J. Watson Research Laboratory for access to the traces used in the simulation.

REFERENCES

[1] J. L. Baer and W.-H. Wang, "On the inclusion properties of multilevel-cache hierarchies," in *Proc. 15th Annu. Int. Symp. Comput. Architecture*, IEEE Cat. 88CH2545-2, June 1988, pp. 73–80.

[2] L. Belady, "A study of replacement algorithms for a virtual-store computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[3] C. K. Chow, "On optimization of storage hierarchy," *IBM J. Res. Develop.*, vol. 18, pp. 194–203, May 1974.

[4] B. Fox, "Discrete optimization via marginal analysis," *Management Sci.*, vol. 13, pp. 909–918, 1966.

[5] G. N. Frederickson and D. B. Johnson, "The complexity of selection and ranking in $X + Y$ and matrices with sorted columns," *J. Comput. Syst. Sci.*, vol. 24, pp. 197–208, 1982.

[6] Z. Galil and N. Megiddo, "A fast selection algorithm and the problem of optimum distribution of effort," *J. ACM*, vol. 26, pp. 58–64, 1979.

[7] M. Z. Ghanem, "Dynamic partitioning of the main memory using the working set concept," *IBM J. Res. Develop.*, pp. 445–450, Sept. 1975.

[8] J. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufman, 1990.

[9] M. D. Hill, "A case for direct-mapped caches," *IEEE Comput. Mag.*, vol. 21, no. 12, pp. 25–40, Dec. 1988.

[10] T. Ibaraki and N. Katoh, *Resource Allocation Problems: Algorithmic Approaches*. Cambridge MA: M.I.T. Press, 1988.

[11] D. B. Kirk, "Process dependent static cache partitioning for real time systems," in *Proc. Real Time Syst. Symp.*, 1988, pp. 181–190.

[12] S. Przybylski, M. Horowitz, and J. Hennessy, "Characteristics of performance optimal multilevel cache hierarchies," in *Proc. 16th Annu. Int. Symp. Comput. Architecture*, 1989, pp. 114–121.

[13] A. Smith, "Cache memories," *ACM Comput. Surveys*, vol. 14, no. 3, pp. 473–530, Sept. 1982.

[14] A. J. Smith, "Line (block) size choice for CPU cache memories," *IEEE Trans. Comput.*, vol. C-36, no. 9, pp. 1063–1075, Sept. 1987.

[15] J. E. Smith and J. R. Goodman, "Instruction cache replacement policies and organizations," *IEEE Trans. Comput.*, vol. C-34, no. 3, pp. 234–241, Mar. 1985.

[16] K. So and R. N. Rechtschaffen, "Cache operations by MRU change," *IEEE Trans. Comput.*, vol. 37, no. 6, pp. 700–709, June 1988.

[17] H. S. Stone, D. F. Thiebaut, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Trans. Comput.*, vol. 41, no. 6, pp. 665–676, 1992. The paper originally appeared as IBM Research Report RC 15072, Oct. 26, 1989.

[18] W. Strecker, "Transient behavior of cache memories," *ACM Trans. Comput. Syst.*, vol. 1, no. 4, pp. 281–293, Nov. 1983.

[19] A. N. Tantawi, D. Towsley, and J. L. Wolf, "An algorithm for a class-constrained resource allocation problem," in *Proc. 1988 ACM Sigmetrics Conf.*, May 1988, pp. 253–260.

[20] D. F. Thiebaut, "On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio," *IEEE Trans. Comput.*, vol. 38, no. 7, pp. 1012–1026, July 1989.

[21] D. F. Thiebaut and H. S. Stone, "Footprints in the cache," *ACM Trans. Comput.*, vol. 5, no. 4, pp. 305–329, Nov. 1987.

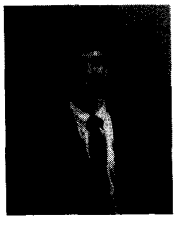
[22] D. F. Thiebaut, H. S. Stone, and J. L. Wolf, "A theory of cache behavior," IBM Res. Rep. RC 13309, Nov. 10, 1987.

[23] M. K. Vernon, R. Jog, and G. S. Sohi, "Performance analysis of hierarchical cache consistent multiprocessors," in *Perform. Distributed and Parallel Systems*, Proc. IFIP TC 7/WG 7.3, 1989, pp. 111–126.

[24] J. L. Wolf, D. M. Dias, and P. S. Yu, "An effective algorithm for parallelizing sort merge joins in the presence of data skew," in *Proc. 2nd Int. Symp. Databases in Parallel Distributed Syst.*, pp. 103–115, 1990.

[25] J. L. Wolf, B. R. Iyer, K. R. Pattipati, and J. Turek, "Optimal buffer partitioning for the nested block join algorithm," in *Proc. 7th Int. Data Eng. Conf.*, 1991, pp. 510–519.

Harold S. Stone (S'61–M'63–SM'81–F'87), for a photograph and biography, see the April 1992 issue of this TRANSACTIONS, p. 410.



John Turek (S'88–M'91) received B.Sc. degree from the Massachusetts Institute of Technology in 1984 and the Ph.D. and M.S. degrees from the Courant Institute of Mathematical Sciences of New York University in 1991 and 1990, respectively.

He is a Research Staff Member at IBM T. J. Watson Laboratories. His current interests include database systems, distributed computing, and the study of optimization problems in computer science.

Joel L. Wolf, for a photograph and biography, see the April 1992 issue of this TRANSACTIONS, p. 410.