549

Instruction Cache Replacement Policies and Organizations

James E. Smith

James R. Goodman

Departments of
Electrical and Computer Engineering
and
Computer Sciences
University of Wisconsin-Madison
Madison, WI 53706

Abstract--Instruction cache replacement policies and organizations are analyzed both theoretically and experimentally. Theoretical analyses are based on a new model for cache references-the loop model. First the loop model is used to study replacement policies and cache organizations. It is concluded theoretically that random replacement is better than LRU and FIFO, and that under certain circumstances, a direct-mapped or set-associative cache may perform better than a full-associative cache organization. Experimental results using instruction trace data are then given and analyzed. The experimental results indicate that the loop model provides a good explanation for observed cache performance.

Index Terms--Cache memories, memory organization, replacement algorithms, direct-mapped, fully associative, set-associative, loop model.

1. Introduction

Systems with a cache memory specifically for instructions have been used in large scale scientific computers for some time [1,2], and are beginning to be used in other high performance systems [3-5]. If they continue to follow the evolutionary pattern of other performance enhancement techniques, instruction caches will be used in a wide variety of computer systems within a few years.

Instruction caches provide the same advantages as more general caches, i.e., reduced memory access time and reduced memory bandwidth requirements, and they provide the following additional advantages.

(1) When used in conjunction with a data cache, they increase the total cache bandwidth.

(2) An instruction cache designed for a modern architecture can be simpler than a data cache or a combined instruction/data cache because stores into cache locations can be disallowed (i.e. there can be no self-modifying code; an assumption consistent with modern programming principles and protection methods.)

(3) An instruction cache can be tailored to the specific referencing patterns found in fetching instruction streams. A separate data cache can also be tailored to data referencing patterns, resulting in better usage of both.

This paper is aimed at exploiting the last advantage given above. It studies cache organizations and replacement policies that are intended solely for instruction caches. We first propose a new model for instruction address patterns, the loop model. This model is the basis for a theoretical analysis of cache organizations and replacement policies. Then, we give results of an experimental study using instruction trace data. This experimental data is analyzed in light of the theoretical conclusions.

1.1. Definitions

We consider memory/instruction cache systems where memory is divided into contiguous *blocks* of some fixed size (typically a power of 2). An instruction cache holds a fixed number of blocks of instructions, and all instruction fetch references are checked to see if the requested instruction word is in the instruction cache. If so, there is a *hit* and the instruction word from the cache is immediately decoded and executed. If it is not present, there is a *miss;* the program block containing the requested instruction is brought in from main memory, and it replaces some block in the instruction cache according to a *replacement policy*. In this paper, we assume cache blocks are only replaced following a miss: there is no prefetching of program blocks as in [2]. Note that we call the physical locations that make up the cache the *cache blocks*, and all the blocks of a program, only some of which may be in the cache, are the *program blocks*.

There are three common ways to organize a cache. Each program block resident in a cache has a *tag* associated with it to help in determining if a referenced program block is in the cache. In a *fully associative cache*, any program block potentially can be found in any of the cache blocks. A memory address is divided into two fields, shown in Fig. 1a. The word field indicates the instruction word within a block, and the tag field is compared against the tags of all the blocks in the cache to determine if there is a hit.

In a *direct-mapped cache*, any program block can be placed in only one cache block. A memory address is divided into three fields (Fig. 1b). The word field indicates word within a block; the block field indicates the cache block that may contain the program block, and the tag field is used for comparison with the tag of the block in the cache to determine if there is a hit.

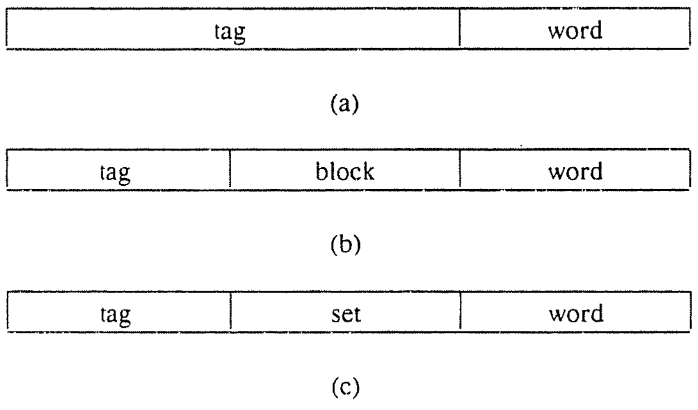

Fig. 1. Cache addresses for
a) a fully associative cache,
b) a direct-mapped cache,
c) a set-associative cache.

In a *set-associative cache*, cache blocks are divided into *sets* and a program block may be within any of the cache blocks in one set. The address is again divided into three fields. The word field is as before, the set field identifies the set of cache blocks that may contain a referenced program block, and the tag field is compared with the tags of all the blocks in the set to determine if there is a hit or a miss.

An fetches a basic unit of storage, typically a "word," into the instruction unit. It may include part or all of one or more instructions. For an instruction cache, the *hit ratio* is the number of instruction references that result in hits divided by the total instruction references. Hit ratio is a measure often used for cache performance. In this paper, we will be more interested in *block hit ratio*. To evaluate the block hit ratio, we consider only instruction references that are in a program block different from the block holding the immediately previous instruction reference. We call these *new block references*. The block hit ratio is then found by dividing the new block references that result in instruction cache hits by the total new block references.

We define block hit ratio because only a new block reference can result in a cache miss, and, more importantly, a block replacement. By considering only references that potentially can result in a replacement, the block hit ratio provides a better measure of the impact of replacement policy on cache performance. Conversely, the measure is intended to eliminate cache hits caused by instruction word "prefetching" that comes with using block sizes larger than the instruction word size. This prefetching effect occurs when there are consecutive instruction references to the same block. All hits after the first are independent of replacement policy, and are eliminated from our performance measure.

1.2. Instruction Reference Model

In most real programs a significant percentage of instructions are executed within loops [6], this leads us to study instruction caches with a *loop model*. In the loop model, an instruction reference stream consists of an unbounded sequence of new block references formed by periodically repeating the same finite sequence of new block references. Each member of the repeating sequence is assumed to differ from the others. The loop model is an approximation of real program behavior based on the observation that certain sequences of code are repeatedly executed; but there are many other more complex examples of repeated code sequences that can contain several branches.

We further divide the loop model into two sub-models: the *simple loop model* and the *complex loop model*. In the simple loop model, the reference stream contains a repeating sequence of new block references at consecutive program block addresses except for the jump from the last to the first block in the sequence; this behavior would be observed in a simple DO loop. In the complex loop model, the repeating sequence of new block references do not have to be at consecutive program block addresses.

Most real programs contain many different loops of different lengths. Consequently, the loop model does not allow one to predict accurately the hit ratio of a specific program; rather it provides a mechanism for explaining and predicting qualitative differences in hit ratios as cache dimensions and replacement policies are changed. In addition, there are many instances of program behavior that do not conform to the loop model, for example, multiple calls to the same procedure within a loop. Hence, the loop model can probably best be used as a guide to the system designer during the initial phases of the design process. Finally,

Following the survey of previous research given in the next subsection, sections 2, 3, and 4 consider the loop model for full-associative, direct-mapped, and set-associative caches, respectively. Section 5 gives experimental results and analysis.

1.3. Previous Research

Most past research on cache memories has considered data caches or combined instruction/data caches. One exception is in [7] where it is assumed that there is a fixed amount of cache memory, and it can be partitioned into instruction and data portions. The authors of [7] conclude that such a scheme is not worthwhile. As we have pointed out, however, current high performance systems are being built with separate caches. This is probably because separate caches yield up to twice the cache bandwidth, a factor not considered in [7]. Also, hardware costs are now low enough to permit extra hardware for instruction caches (i.e., the amount of cache available for data is not necessarily reduced as an instruction cache is added). In a recent survey paper on caches [8], A. J. Smith studied separate instruction and data caches, but looked at a different set of problems than those discussed here. He considered ways data and instruction caches should be partitioned and whether duplicate entries should be allowed; all the instruction caches studied were set-associative with LRU replacement. Theoretical work has also been geared to data or combined caches. A good example is [9] where the independent reference model [10] is used.

Other related work is in the ATLAS system [11] where virtual memory page replacement algorithms attempted to recognize looping behavior. Since they were used at a higher level in the memory hierarchy, these methods could be implemented in software and could be more sophisticated than the hardware methods that must be used for cache replacement. In any case, the ATLAS method was later found to be of questionable value [12].

2. Fully Associative Cache

We first consider a fully associative cache because it is the easiest to analyze. Later sections consider direct-mapped and set-associative organizations. We assume the loop model, and assume as an initial condition that the cache is full, but not with instructions from the loop. This assumption is to model more closely a real program environment, where, when a loop is entered, some instructions preceding the loop are in the cache. We are also interested in steady state behavior. That is, after some start-up transient as the loop is begun, cache replacements typically fall into a steady state pattern from which a block hit ratio can be determined.

2.1. LRU and FIFO Replacement

We first consider First-In, First-Out (FIFO) and Least Recently Used (LRU) replacement policies, because they are the most commonly used strategies in data and combined caches. We assume that a complex loop contains N program blocks, and the instruction cache contains M cache blocks. If $N \leq M$, then the loop fits entirely in the associative cache, and, after an initial period as the cache is filled, the steady state block hit ratio is 1. Under these circumstances the replacement strategy is irrelevant since replacement never takes place.

For $N > M$, consider the following example. Let $M = 4$, and number the cache blocks 0,1,2,3. Let $N = 5$, and label the loop program blocks A, B, C, D, E. That is, the block reference pattern is:

$$ABCDEABCDEABCDE\ldots$$

After the first four references (A, B, C, D), both FIFO and LRU lead to the same assignment of program to cache blocks:

$$A \to 0; \quad B \to 1; \quad C \to 2; \quad D \to 3.$$

Now, when block E is referenced, there is a miss and a program block in the cache must be replaced. In this example, both FIFO and LRU would replace A and put E into block 0. Then the next instruction reference is to block A as the loop begins again, but A has just been removed. So, both FIFO and LRU replace block B (just before it is needed). This continues, and the block hit ratio in steady state becomes 0.

A situation similar to the above occurs whenever $N > M$, and this leads to the following theorem.

Theorem 1: Under the complex and simple loop models with a fully associative cache, if $N \leq M$ the steady state block hit ratio for both FIFO and LRU replacement is 1. If $N > M$, both FIFO and LRU lead to a steady state block hit ratio of 0.

For brevity, proofs to Theorems have been omitted; for the most part they are straightforward and follow from discussion appearing in the paper.

To summarize, when $N \leq M$, FIFO and LRU are no better than any other replacement strategy (replacement is not needed) and when $N > M$ FIFO and LRU give the worst possible block hit ratio.

2.2. Optimum Replacement

In [13], it is shown that a replacement strategy that maximizes hit ratio replaces the block that will be needed the farthest in the future. Of course, this cannot be practically implemented since it requires knowledge of the future, but it does provide an upper bound against which other replacement strategies can be measured. For the loop model and a fully associative cache one can derive a closed-form expression for the optimum steady state block hit ratio.

It is easiest to begin as the complex loop is entered. According to our initial condition assumption, the cache is full when the loop is begun, but all the M residual blocks are replaced by the first M blocks of the loop. Without loss of generality, assume that the cache blocks are filled in order $(0,1,\ldots M-1)$. Then, there are $N - M$ further misses as the first pass through the loop completes. All these missed blocks go into cache block $M - 1$ if the optimum algorithm is followed. Then, on pass 2 through the loop, the first $M - 1$ blocks are all hits. Then, there are again $N - M$ misses, all of which are placed in cache block $M - 2$. Reference N of pass 2 is still in cache block $M - 1$ and hits, as do the first $M - 2$ references of loop 3. This again gives $M - 1$ consecutive hits. This is followed by $N - M$ misses, all of which go into cache block $M - 3$, etc. By formalizing the above argument, it can be shown that the hit-miss pattern in the steady state is always $M - 1$ hits followed by $N - M$ misses. This leads to the following Theorem.

Theorem 2: Under the complex and simple loop models with a fully associative cache, if $N > M$ the optimum steady state block hit ratio is $(M - 1) / (N - 1)$.

2.3. Random Replacement

Assuming a finite M and N, random replacement intuitively leads to a steady-state hit ratio greater than 0, regardless of M and N. Hence, under the loop model, we expect random replacement to behave as well as LRU and FIFO when $N \leq M$ and to be better when $N > M$. In this section we provide a method for computing the steady state hit ratio with the complex loop model and random replacement. As before, we assume a fully associative cache. Results are given to show that random replacement can perform well, relative to the optimum performance given in the previous section.

To compute block hit ratio under random replacement, we need to develop a Markov model. Initially we consider a state to consist of: 1) the current mapping of program blocks to cache blocks and 2) the most recently used program block in the cache. For example, if $M = 4$ and $N = 5$ a typical state might be represented as an ordered 4-tuple with a distinguished element: $\textcircled{A}BED$. That is, A is in cache block 0, B is in cache block 1, E is in cache block 2, and D is in cache block 3. The circle distinguishes A as the most recently used program block. Then, the next state must be: $A\textcircled{B}ED$. The next block reference to C must then result in a miss, and each of the four following states have equal probability of being entered: $\textcircled{C}BED$, $A\textcircled{C}ED$, $AB\textcircled{C}D$, $ABE\textcircled{C}$. The portion of the Markov model just described is shown in Fig. 2. We define a *miss state* as one where the next block reference is a miss and a *hit state* as one where the next block reference is a hit. In the above example, $A\ B\ ED$ and $ABE\ C$ are miss states and the others are hit states.

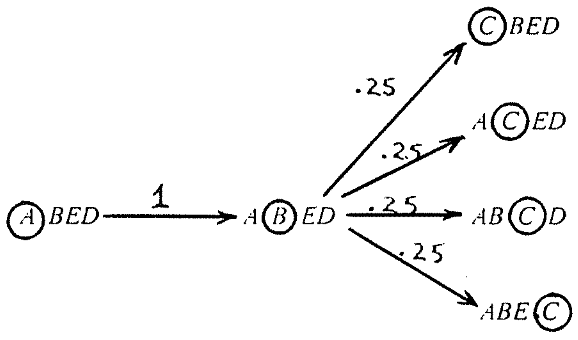
$$\begin{tikzcd}
 & & \textcircled{C}BED \\
 & & A\textcircled{C}ED \\
\textcircled{A}BED \arrow[r, "1"] & A\textcircled{B}ED \arrow[uur, ".25"] \arrow[ur, ".25"] \arrow[r, ".25"] \arrow[dr, ".25"] & AB\textcircled{C}D \\
 & & ABE\textcircled{C}
\end{tikzcd}$$

Fig. 2. A portion of the Markov model when $M = 4$, $N = 5$.

In the model just described, there are $\frac{M \cdot N!}{(N - M)!}$ states. If $M = 4$, $N = 5$, there are 480 states, and as M and N grow, the number of states grows rapidly. To make the problem more tractable, we use state collapsing, i.e., combine equivalent states in the Markov model. First, since the cache is fully associative and replacement is random, the actual assignment of program to cache blocks is unimportant; it is only necessary to know which program blocks are in the cache. For example, $\textcircled{A}BCD$, $\textcircled{A}CBD$, $\textcircled{A}DCB$, $DCB\textcircled{A}$, etc., are all equivalent states. A second, and less obvious, collapsing of states can be achieved by considering states to consist of present program pages relative to the one most recently referenced. Hence, we collapse states by using as the representative of an equivalence class the state where A is the most recently referenced. For example $\textcircled{A}ECB$ is equivalent to $\textcircled{C}BED$; the blocks E, C, B are in the same loop positions relative to A as the blocks B, E, D are to C. The entire model for $N = 5$, $M = 4$ is shown in Fig. 3.

$$\begin{tikzcd}
ABCD \arrow[r, "1."] & ABCE \arrow[r, "1."] & ABDE \arrow[r, "1."] & ACDE \arrow[lll, bend left, ".25"] \arrow[ll, bend left, ".25"] \arrow[l, bend left, ".25"] \arrow[loop right, ".25"]
\end{tikzcd}$$

Fig. 3. The Markov model for $M = 4$, $N = 5$ after state collapsing.

The number of states in a model with collapsed states becomes

$$\frac{(N-1)!}{(N-M)!\,(M-1)!}.$$

This does give a tractable model, at least for most values of N and M that would be of practical interest. For example if $M = 4$ and $N = 5$, the transition matrix is shown in Fig. 4. To relate Figs. 3 and 4, ABCD is state 0, ABCE is state 1, etc. The only miss state is State 3, and the transition matrix can be solved to show that in the steady state, the probability of being in State 3 is .4; hence, the block hit ratio is .6.

		State			
		0	1	2	3
	0	0	1	0	0
	1	0	0	1	0
State	2	0	0	0	1
	3	.25	.25	.25	.25

Fig. 4. State transition matrix for $M = 4$, $N = 5$.

We consider $N = M + 1$ to be an important case because it models the smallest loop that does not fit entirely in the cache. A cache would ordinarily be designed so that most loops fit, and because larger loops probably would occur with decreasing likelihood, the $N = M + 1$ case would be the most likely of those that do not fit. For $N = M + 1$, the transition matrix can be generalized and solved to yield the following Theorem.

Theorem 3: Under the complex and simple loop models with a fully associative cache, if $N = M + 1$, a random replacement policy gives a steady state hit ratio of $\frac{M-1}{M+1}$.

Using Theorem 2 the optimum block hit ratio when $N = M + 1$ is

$$\frac{M-1}{N-1} = \frac{M-1}{M}.$$

We see that for the case $N = M + 1$ the random replacement policy is reasonably close to optimum. When $N \leq M$, it can be shown that the steady state hit ratio with random replacement becomes 1 with probability 1.

For values of M and N where $N > M + 1$ we have not found a general closed form solution. We did find block hit ratios for several specific cases, however, and these are shown graphically in Fig. 5. Fig. 5a assumes a cache with 4 blocks and shows block hit ratios for loops that vary in size from 3 blocks to 12 blocks. Loops of size less than 3 have block hit ratios of 1. Fig. 5b assumes a cache with 16 blocks and shows block hit ratios for loops that vary in size from 15 blocks to 24 blocks. Loops of size less than 15 have block hit ratios of 1. Along with the block hit ratios for random replacement, the block hit ratios for optimum and LRU/FIFO replacement are also shown in Fig. 5. In addition, results for a direct-mapped cache, to be discussed in the next section, are given. The superiority of random replacement over LRU/FIFO is clearly shown. In addition, random replacement is shown to be reasonably close to optimum for smaller loops, but as loop sizes become larger, hit ratios for random replacement tend to fall off faster.

3. Direct-Mapped Cache

In a direct-mapped cache, a program block can only be placed in one cache block; hence, there is only one possible replacement policy. Nevertheless, it is interesting to see how well a direct-mapped cache performs under the loop model. The discussion in this section assumes that addresses are broken up into fields as in Fig. 1b.

We begin by considering simple loops. A simple loop fills consecutive direct-mapped cache blocks with the bottom of the cache wrapping around to the top. For example, if $N = 6$ and $M = 4$ then program blocks might fill in as:

program blocks(s)		cache block
A, E	$\rightarrow$	0
B, F	$\rightarrow$	1
C	$\rightarrow$	2
D	$\rightarrow$	3

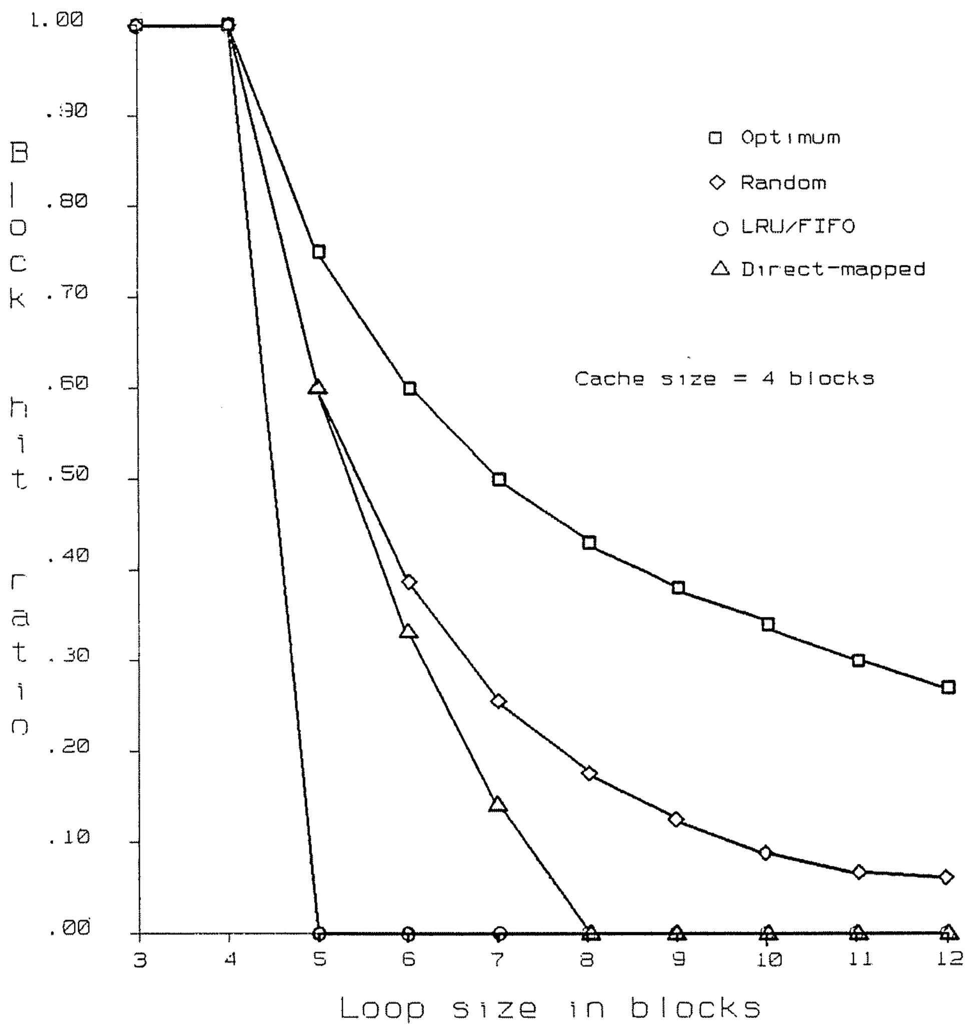

Fig. 5a. Theoretical block hit ratios. Optimum, random, and LRU/FIFO replacement are for full-associative caches; for the direct-mapped cache, simple loops are assumed; four-byte blocks.

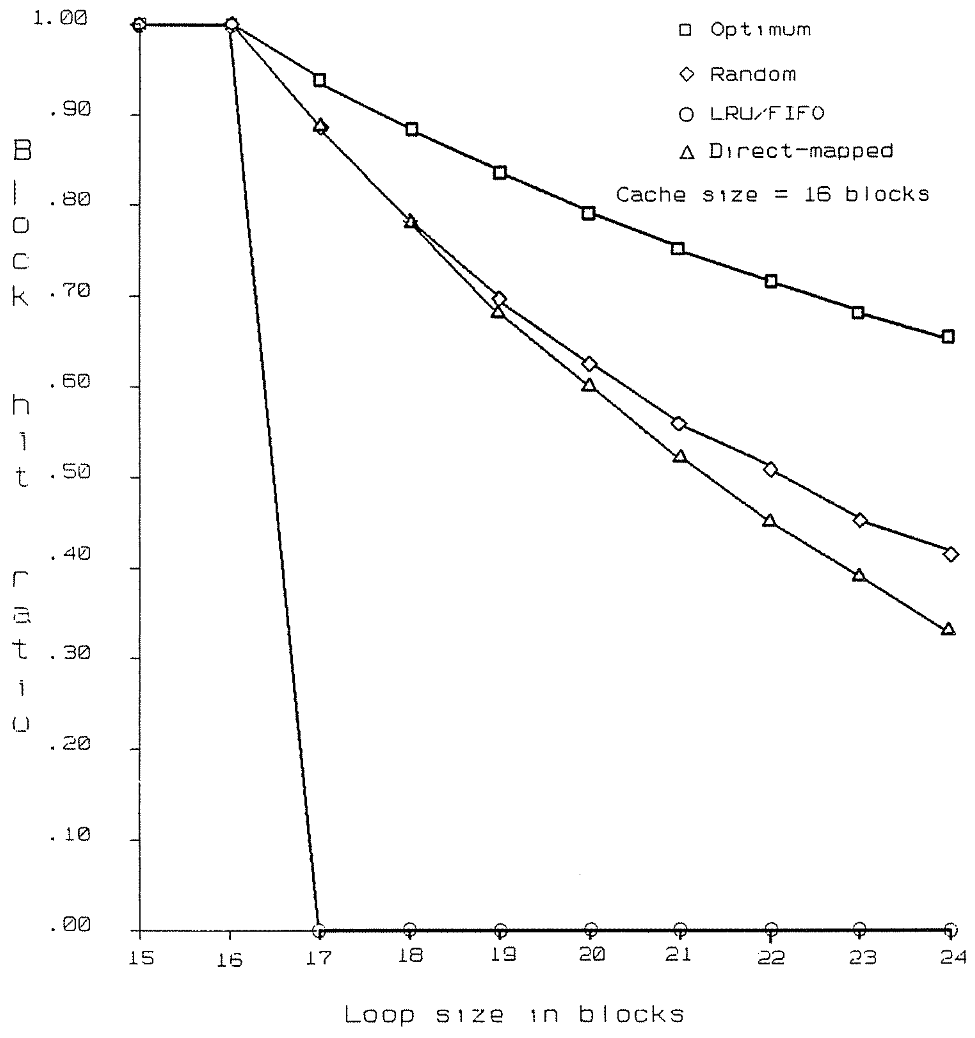

Fig. 5b. Theoretical block hit ratios. Optimum, random, and LRU/FIFO replacement are for full-associative caches; for the direct-mapped cache, simple loops are assumed; sixteen-byte blocks.

That is, program blocks *A* and *E* contend for cache block 0, B and *F* contend for block 1, and *C* and *D* each have their own cache block. When more than one program block contend for a cache block, then the block hit ratio for each of them is 0. On the other hand, if a program block has a cache block to itself, the block hit ratio for that particular block is 1. For the above example, this leads to an aggregate block hit ratio of .33.

The above observations lead to the following Theorem.

Theorem 4: Under the simple loop model with a direct-mapped cache,

a) if $N \le M$ the block hit ratio is 1

b) if $M < N < 2M$ the block hit ratio is $\frac{2M - N}{N}$

c) if $2M \le N$ the block hit ratio is 0.

Fig. 5 contains block hit ratios for a direct-mapped cache with simple loops. These graphs, derived from Theorems 1 and 4, lead us to conclude that for the simple loop model, a direct-mapped cache gives better performance than a fully associative cache of the same size using FIFO or LRU replacement.

For the special case $N = M + 1$, Theorem 4 gives a hit ratio of $\frac{M - 1}{M + 1}$, the same as for random replacement in a fully associative cache. For $2M \le N$ it is always worse than random, since random is nonzero for any finite M and N. This, as well as the data in Fig. 5, leads us to conclude that for the simple loop model, using a direct-mapped cache gives no better performance than a fully associative cache using random replacement. This conclusion is somewhat weaker than the others because it is not proved for all combinations of M and N.

The complex loop model is more difficult to analyze because program blocks do not necessarily map into the cache blocks in any regular way. In the worst case, all N blocks map into the same cache block, and the block hit ratio is 0, even if $1 \le N \le M$. In the best case, $M - 1$ of the program blocks map into $M - 1$ different cache blocks, and the remaining program blocks all map into the single remaining cache block. The block hit ratio is $\frac{M - 1}{N}$, and is close to optimum. Hence, for the complex model, we can only observe that for the complex loop model, the particular assignment of program to cache blocks determines whether a direct-mapped cache or an associative cache works better.

4. Set-associative Cache

We now show ways some of the previous results can be extended to set-associative caches. When analyzing a set-associative cache, we come up against the same problem we did with a direct-mapped cache: for complex loops, the mapping of program blocks into sets must be known. Once the mapping is known, all the blocks mapping to the same set behave as if they are in a fully associative cache the size of one set. Then by collecting together the block hit ratios for each of the sets, the aggregate block hit ratio can be derived by forming a weighted average.

Let $bhr(P, B, S, R)$ be the block hit ratio for a fully associative cache with S blocks of size B, using replacement policy R, when P program blocks all map into the cache and follow the complex loop model.

Theorem 5: Under the complex loop model with a set-associative cache of set size S, block size B, T sets, with P_i program blocks mapping into set i, and replacement policy R, the block hit ratio is

$$\frac{1}{N} \sum_{i=1}^{T} P_i \cdot bhr(P_i, B, S, R)$$

To take one example, consider a set-associative cache with 4 sets of 2 blocks each that uses random replacement. Let a complex loop have 5 blocks where the mapping into sets is as follows:

$A \to Set\ 0;\quad none \to Set\ 1;\quad B, D, E \to Set\ 2;\quad C \to Set\ 3.$

Then, using Theorems 3 and 5, the block hit ratio is $\frac{1}{5} \cdot (1. + 0 + (3 \cdot 0.5) + 1.) = 0.7$. With the above theorem, many of the earlier results can be easily extended to a set-associative cache for the simple loop model.

It was shown earlier that both a direct-mapped cache and a random cache behave better than a fully associative cache with FIFO or LRU replacement for simple loops. A direct-mapped cache works well because it guarantees some blocks will never miss, provided $N \leq 2M$. Random replacement works well because even if multiple program blocks must fit into fewer cache blocks, a program block has a chance of surviving and producing a hit.

Another observation is that simple loops are probably smaller than complex loops. Hence, when a cache is very small (as might be the case if an instruction cache is part of a microprocessor chip) then some simple loops may not fit, and policies that work well for simple loops may be particularly advantageous. For larger caches, complex loop behavior probably becomes dominant. These observations lead us to conjecture that a set-associative cache with random replacement combines the advantages of a direct-mapped cache and a fully associative cache with random replacement for the simple loop model. Hence, this configuration may give the best performance for programs that are dominated by simple loops or where the cache is small compared with most loops.

5. Experimental Results and Conclusions

We ran several experiments with instruction trace data to test the above theoretically-based conclusions. We intentionally avoided code that is heavily simple-loop-dominated (e.g., scientific code). In particular, we looked at VAX-11 programs written in C and running under UNIX*. The specific routines chosen were:

NROFF	The program *nroff* interpreting the Berkeley macro package *-me*.
CACHE	The program used to determine hit ratios and other relevant statistics for these traces.
COMPACT	A program using an on-line algorithm which compresses files using an adaptive Huffman code.

A real program typically consists of a complex combination of loops, procedure calls, and returns. In a real system cache, the organization, dimensions, and replacement policy are fixed while the various program loops differ in size: some may fit entirely in the cache and some may not. Consequently, we cannot generate experimental results that can be quantitatively compared with theoretical projections. Rather, we can compare experimental results with each other and explain the observed behavior via the theoretical conclusions.

Each of the three programs was simulated for a total of 100,000 new block references, using a variety of cache configurations and replacement policies. Instruction references from the cache are aligned longwords (4 bytes) and include the entire instruction stream (i.e. opcodes, operand specifiers, immediate data, etc.) For each configuration and replacement policy the resulting hit ratios for

*UNIX and NROFF are trademarks of Bell Laboratories.

Table 1: Block Hit Ratio, 16-Byte Blocks

Cache Size	Fully Associative				4-way Set Associative				2-way Set Associative				Direct
(bytes)	OPT	RAN	LRU	FIFO	OPT	RAN	LRU	FIFO	OPT	RAN	LRU	FIFO	
64	.313	.185	.183	.188	.313	.185	.183	.188	.277	.187	.178	.176	.182
128	.439	.274	.265	.263	.420	.276	.261	.261	.380	.279	.274	.273	.260
256	.605	.411	.372	.379	.570	.412	.377	.382	521	.415	.403	.402	.407
512	.774	.599	.511	.505	.735	.600	.567	.552	.680	.588	.566	.551	.541
1024	.945	.866	.777	.748	.907	.842	.792	.779	.848	.796	.774	.763	.749
2048	.993	.985	.989	.985	.987	.977	.971	.968	.965	.952	.948	.945	.889
4096	.995	.994	.995	.995	.995	.993	.994	.994	.992	.990	.990	.989	.975
8192	.995	.995	.995	.995	.995	.995	.995	.995	.995	.995	.995	.995	.994

Table 2: Total Hit Ratio, 16-Byte Blocks

Cache Size	Fully Associative				4-way Set Associative				2-way Set Associative				Direct
(bytes)	OPT	RAN	LRU	FIFO	OPT	RAN	LRU	FIFO	OPT	RAN	LRU	FIFO	
64	.748	.702	.701	.703	.748	.701	.701	.703	.734	.701	.699	.698	.700
128	.793	.732	.728	.728	.786	.733	.727	.727	.771	.734	.733	.732	.727
256	.853	.782	.767	.769	.840	.782	.769	.770	.822	.783	.778	.777	.780
512	.916	.851	.818	.816	.901	.851	.839	.834	.880	.846	.837	.832	.829
1024	.979	.950	.916	.907	.965	.940	.920	.916	.943	.923	.915	.910	.905
2048	.997	.994	.996	.994	.995	.991	.989	.987	.987	.982	.980	.979	.958
4096	.998	.998	.998	.998	.998	.997	.998	.998	.997	.996	.996	.996	.990
8192	.998	.998	.998	.998	.998	.998	.998	.998	.998	.998	.998	.998	.998

Table 3: Block and Total Hit Ratios, 4-Byte Blocks

Cache Size	Fully Associative				4-way Set Associative				2-way Set Associative				Direct
(bytes)	OPT	RAN	LRU	FIFO	OPT	RAN	LRU	FIFO	OPT	RAN	LRU	FIFO	
64	.371	.215	.176	.176	.347	.214	.185	.186	.303	.209	.189	.191	.198
128	.539	.339	.236	.234	.503	.342	.291	.291	.458	.345	.318	.319	.333
256	.726	.551	.522	.503	.691	.553	.513	.504	.631	.533	.508	.505	.504
512	.858	.730	.712	.687	.820	.717	.693	.678	.776	.704	.688	.680	.674
1024	.956	.893	.857	.833	.933	.885	.868	.856	.896	.859	.848	.841	.828
2048	.984	.975	.976	.973	.981	.973	.976	.974	.965	.955	.955	.954	.907
4096	.985	.983	.985	.984	.984	.982	.983	.983	.981	.978	.978	.978	.964
8192	.985	.985	.985	.985	.985	.984	.984	.984	.984	.983	.984	.984	.983

the three programs were averaged. For 16-byte cache blocks, results appear in Tables 1 and 2. Some of the same information appears in graph form in Figs. 6-11. Table 1 contains block hit ratios and Table 2 contains hit ratios for all instruction references. We used fully associative, direct-mapped, and two varieties of set-associative organizations. Table 3 gives similar results for caches with 4-byte blocks; because each instruction reference is for 4 bytes, the block hit ratio and the total hit ratio are the same. Figs. 6-11 plot block hit ratios for caches with 16-byte blocks and 4-byte blocks. For each block size, there is a separate graph for full-associative, 4-way set-associative, and 2-way set-associative cache organizations. On each graph, results for optimum, LRU, and random replacement are shown. Random replacement was determined via an improved version of the UNIX system function *rand()*, a pseudo-random number generator. Results for direct-mapped caches are shown on the full-associative graphs.

First, from the tables it is clear that there is little difference in the hit ratios for FIFO and LRU replacement. Consequently, only the LRU replacement policy is plotted in Figs. 6-11. Also, the data show that random replacement is better than LRU and FIFO. This confirms the loop model analysis.

If simple loops are typically smaller than complex loops as we have conjectured, then a direct-mapped cache can outperform a fully associative cache when the cache size is small and/or there is a significant number of simple loops. The data indicate that there are indeed instances when a direct-mapped cache performs better, and these occur for small cache sizes. We also observe that as cache sizes become larger and most loops fit entirely in the cache, the large complex loops can sometimes lead to worse performance with a direct-mapped cache.

Our conclusion that a fully associative cache with random replacement is superior to a direct-mapped cache is also supported by the data. In no case was a direct-mapped cache worse than a fully associative cache, and typically was much superior.

Finally, there are some cache sizes where a set-associative cache with random replacement gives the best block hit ratio of all the organizations and replacement policies. This is a common occurrence for small caches and also tends to support our observations concerning simple loop behavior.

6. Acknowledgement

This material is based upon work supported by the National Science Foundation under Grants ECS-8207277 and MCS-8202952.

We would like to thank T.-H. Yang for developing the program for generating VAX trace data. We also thank M.C. Chiang for program development work on the trace generator and cache simulator programs. A.L. Fisher offered helpful suggestions based on a preliminary version of this paper. Finally, we thank the referees for their comments and suggestions.

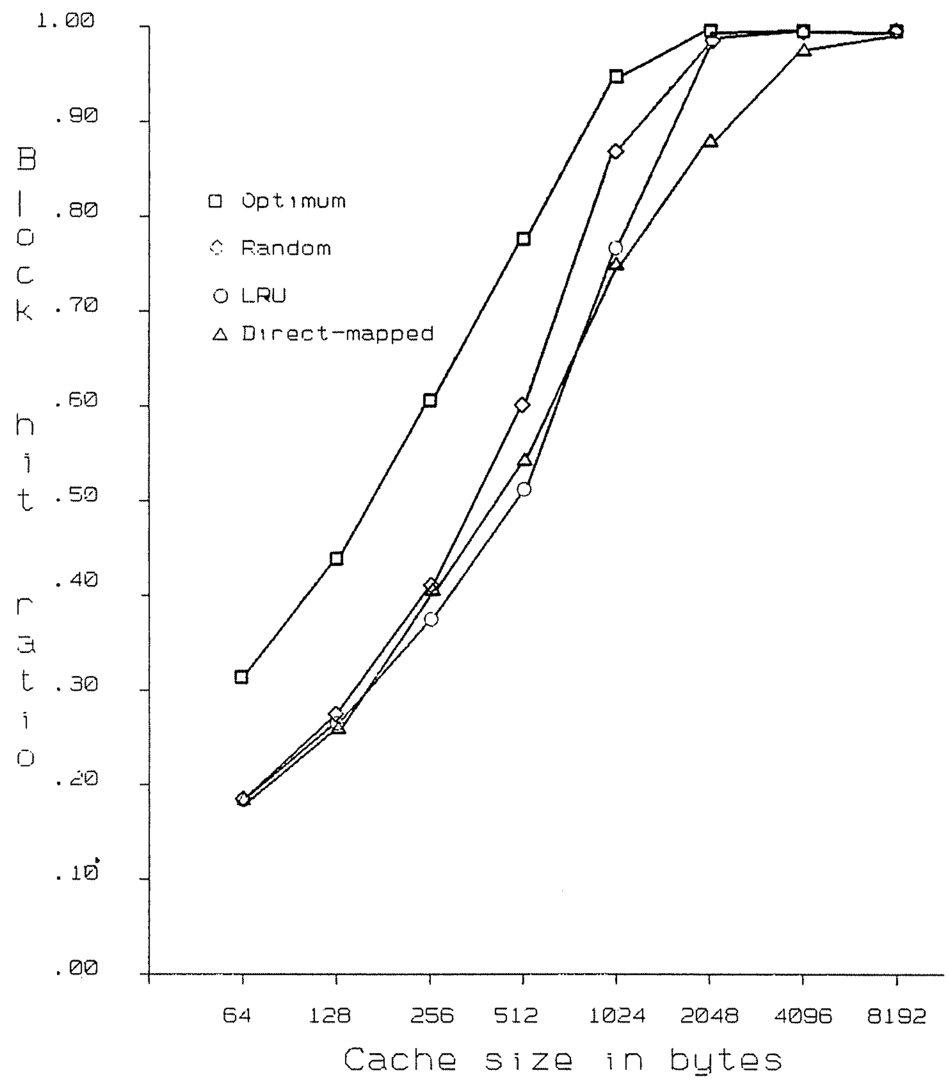

Fig. 6. Block hit ratios for a fully associative cache and a direct-mapped cache with 4-byte blocks.

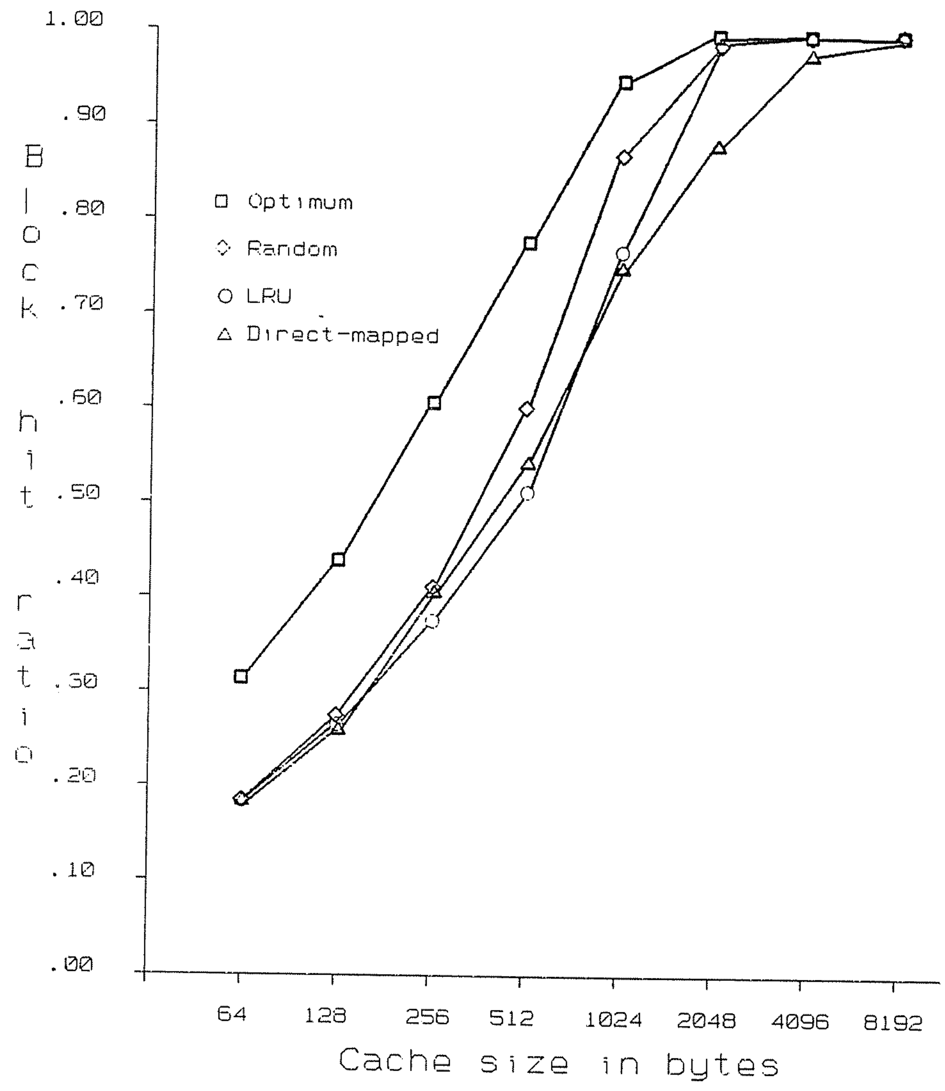

Fig. 7. Block hit ratios for a fully associative cache and a direct-mapped cache with 16-byte blocks.

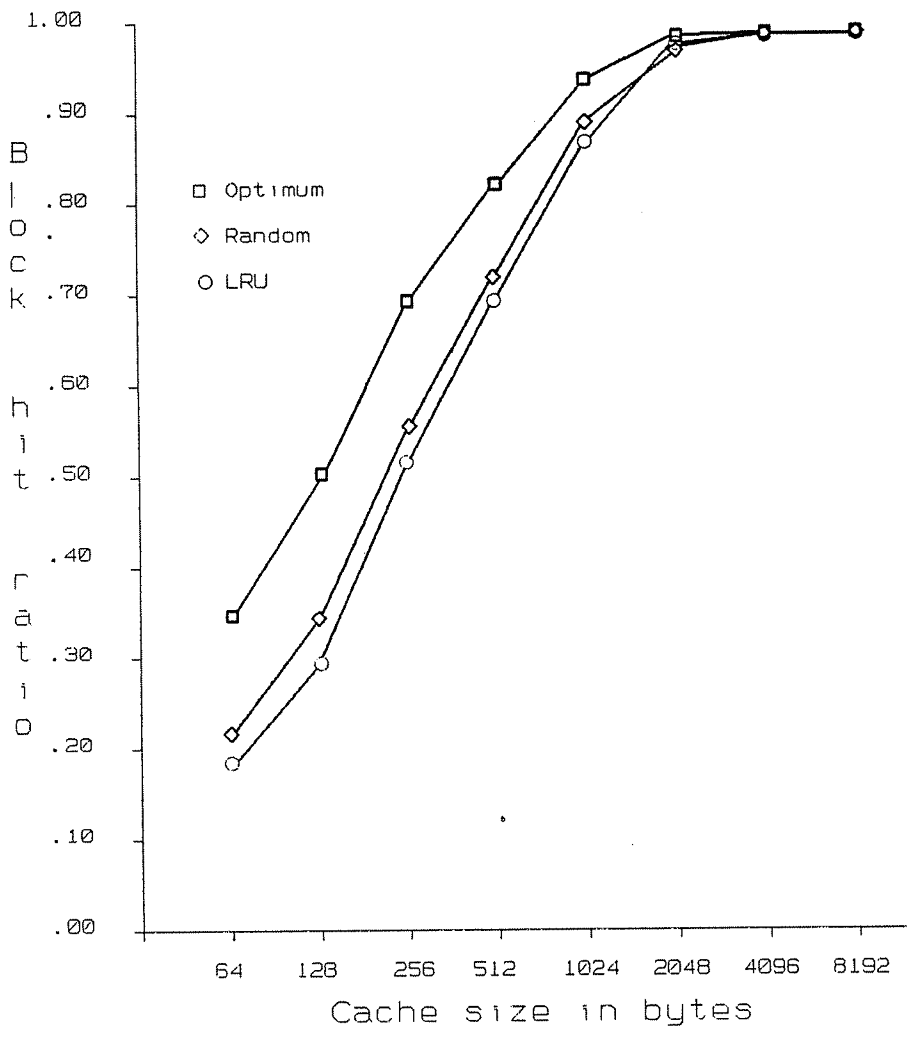

Fig. 8. Block hit ratios for a 4-way set-associative cache with 4-byte blocks.

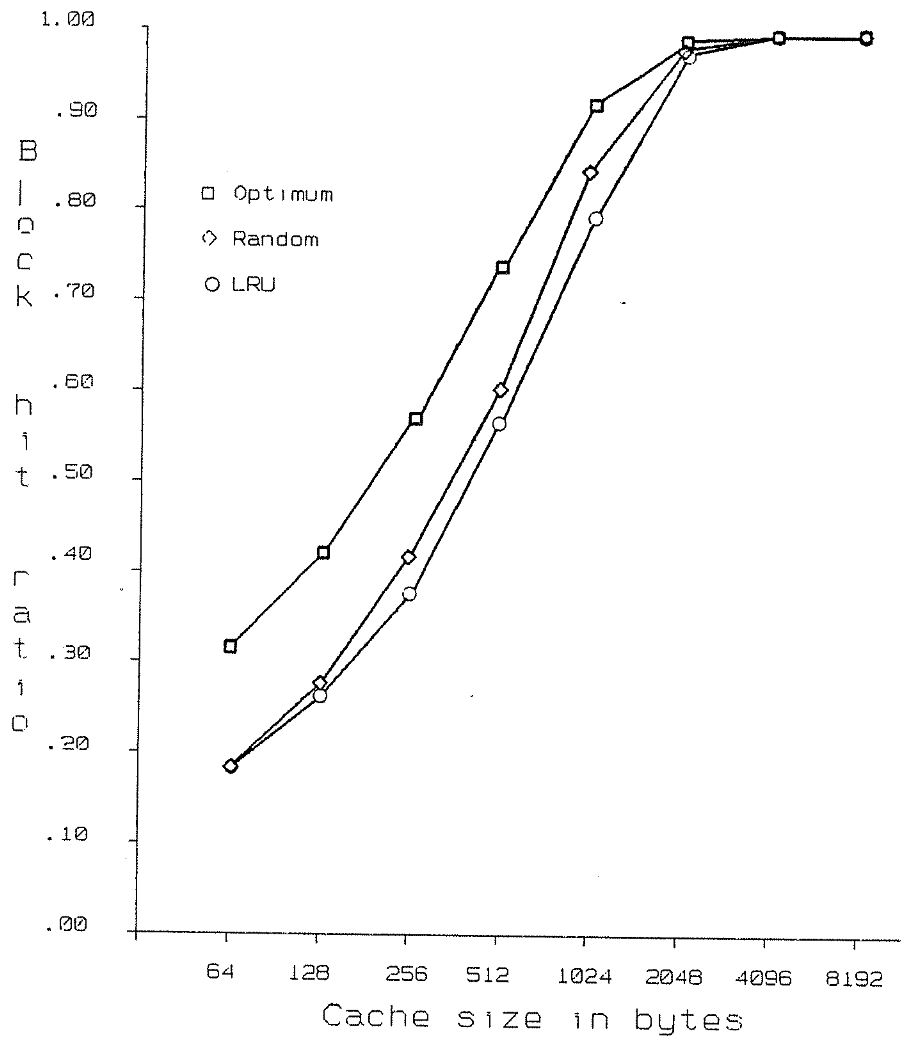

Fig. 9. Block hit ratios for a 4-way set-associative cache with 16-byte blocks.

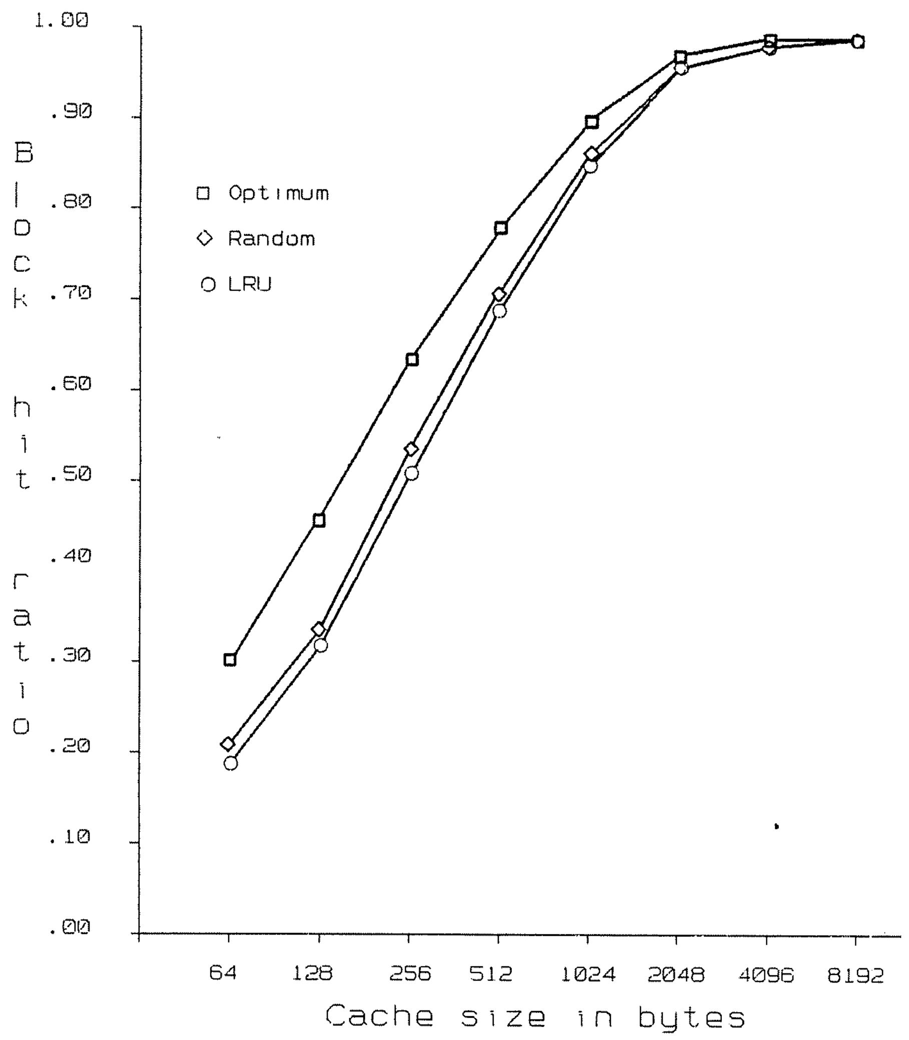

Fig. 10. Block hit ratios for a 2-way set-associative cache with 4-byte blocks.

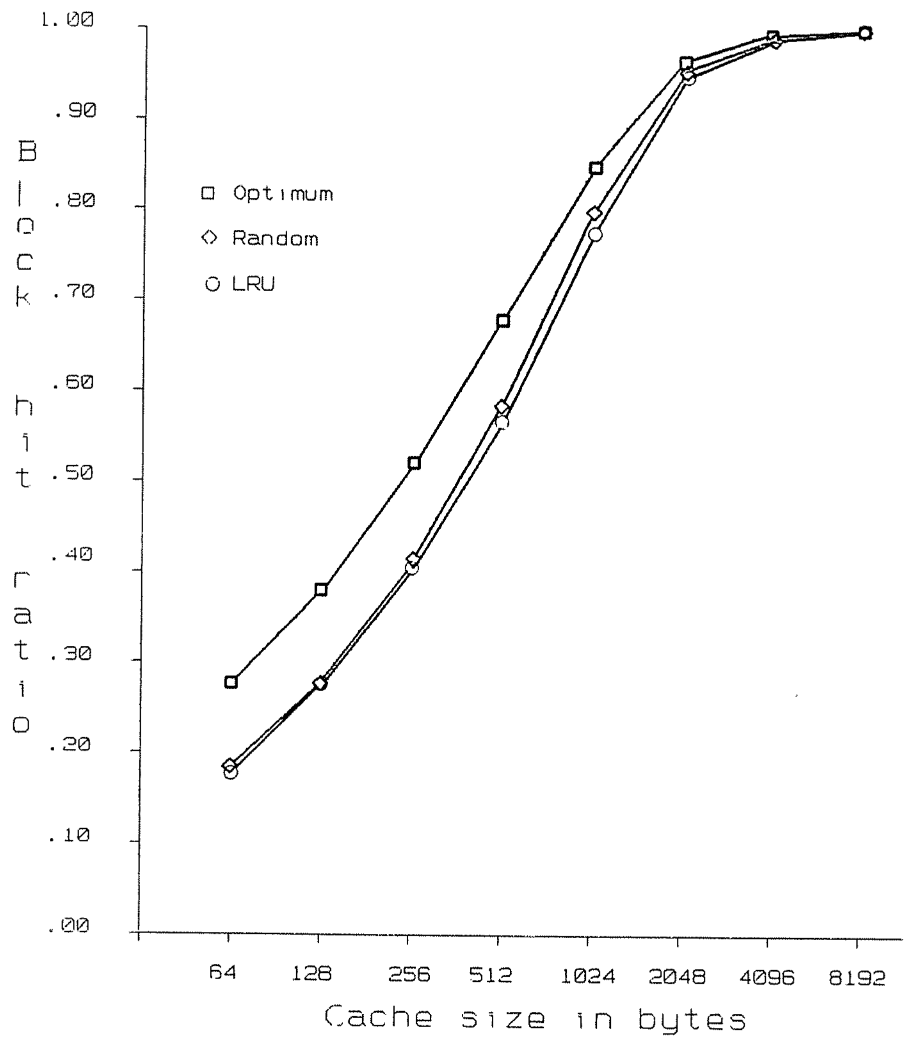

Fig. 11. Block hit ratios for a 2-way set-associative cache with 16-byte blocks.

7. References

[1] *CRAY-1 Computer Systems, Hardware Reference Manual*, Cray Research, Inc., Chippewa Falls, WI, 1979.

[2] *CDC CYBER 200 Model 205 Computer System Hardware Reference Manual*, Control Data Corp., Arden Hills, MN, 1981.

[3] *Data General ECLIPSE MV/8000 Principles of Operation*, Data General Corp., Westboro, MA, April 1980.

[4] *Amdahl 580 Technical Introduction*, Amdahl Corp., Sunnyvale, CA, 1982.

[5] S1 Project Staff, *Advanced Digital Computing Technology Base Development for Navy Applications*, Lawrence Livermore Laboratories, Tech. Report UCID 18038, 1978.

[6] M. Kobayashi, "Dynamic Profile of Instruction Sequences for the IBM System/370," *IEEE Trans. Comp.*, C-32, pp. 859-861, September 1983.

[7] J. Bell, D. Cassasent, and C. G. Bell, "An Investigation of Alternative Cache Organizations," *IEEE Trans. Comp.*, C-23, pp. 346-351, April 1974.

[8] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, pp. 473-530, September 1982.

[9] G. S. Rao, "Performance Analysis of Cache Memories," *Journal of the ACM*, Vol. 25, pp. 378-395, July 1978.

[10] W. F. King, "Analysis of Paging Algorithms," *Proc. IFIP Congress*, pp.485-490, August 1971.

[11] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer, "One-level Storage System," *IRE Trans. on Electronic Computers*, Vol. EC-11, pp. 223-235, April 1962.

[12] M. H. J. Baylis, D. G. Fletcher, and D. J. Howarth, "Paging Studies Made on the I.C.T. Atlas Computer," *Proc. of IFIP Congress*, North-Holland, Amsterdam, pp. 340-346, 1968.

[13] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Systems Journal*, Vol. 5, No. 2, pp. 78-101, 1966.