Genetic Algorithms and Cache Replacement Policy

Erik R. Altman

McGill University, Montreal

October 1991

A Thesis submitted to the Faculty of Graduate Studies and Resaearch in partial fulfillment of the requirements for the degree of Master of Engineering.

Acknowledgements

Thanks start with Jonathan C. Rand, who introduced me to genetic algorithms and kindly lent, then gave me Goldberg's book on the subject. Anne Brindle's unsolicited donation of her Thesis further sparked my interest in genetic algorithms.

Professor Guang R. Gao's inspired teaching has stimulated my interest in many areas of computer architecture, particularly caches. Professor Vinod K. Agarwal has offered numerous insightf ' suggestions along the way, and has been extremely patient in waiting for the final result. Professor P.C.P. Bhatt's review of an early, and none too polished, draft has improved this one considerably. As well, the many conversations with my fellow students helped crystalize several ideas for me.

This work also depended on the considerable computing facilities made available to me in the Computer Systems and Circuits Laboratory. Jacek Slaboszewicz insured that system downtime was minimized, and was always willing to increase my disk quota when needed.

Thanks also to Russell A. Olsen and Yoshiko Fueki, who provided me with a delicious dinner and merriment more times than I can count. Visits and conversation with my family were always fun and helped me keep things in perspective.

Finally and especially thanks to my wonderful wife, Sheila Sundaram, who never complained about my odd hours, who gave me encouragement when I needed it the most, who proofread this thesis on a moment's notice, who translated the abstract, and who helped me with the proofs in the Appendix.

Contents

List of Tables

List of Figures

Abstract

The most common and generally best performing replacement algorithm in modern caches is LRU. Despite LRU's superiority, it is still possible that other *feasible* and *implementable* replacement policies could yield still better performance [34] found that an optimal replacement policy (OPT) would often have a miss rate 70% that of LRU.

If better replacement policies exist, they may not be obvious One way to find better policies is to study a large number of address traces for common patterns. Such an undertaking involves such a large amount of data, that some automated method of generating and evaluating policies is required *Genetic Algorithms* provide such a method, and have been used successfully on a wide variety of tasks [21].

The best replacement policy found using this approach had a mean improvement in overall hit rate of 0.6% over LRU for the benchmarks used. This corresponds to 27% of the 2.2% mean difference between LRU and OPT. Performance of the best of these replacement policies was found to be generally superior to *shadow cache* [33], an enhanced replacement policy similar to some of those used here.

Résumé

Le plus performant et le plus utilisé des algorithmes de remplacement dans les systèmes de cache modernes est l'algorithme LRU. Malgré cette supériorité, il est possible d'implanter d'autres algorithmes de remplacement avec une meilleure performance. Dans [34], on a trouvé un algorithme optimal qui le plus souvent fait preuve d'un "miss rate" valant 70% de celui du LRU.

Même si des algorithmes plus performants existent, la question d'en trouver est autrement difficile. Une technique pour trouver un meilleur algorithme est d'étudier un grand nombre de traces d'adresses pour le même motif. Cette tâche exige le traitement d'une quantité énorme de données, et par conséquent n'est pas praticable sans un outil automatique pour engendrer et évaluer les différents algorithmes. La méthode des *Algorithmes Génétiques* fournit un tel outil; les chercheurs s'en sont servis avec succès pour une large gamme de tâches [21].

Le meilleur algorithme de remplacement trouvé dans cette thèse, en utilisant cette méthode, jouit d'une amélioration moyenne de 0,6% dans le "hit rate" global, relativement à celui de l'algorithme LRU, par rapport aux "benchmarks" utilisés. Cette amélioration correspond à 27% de la différence moyenne de 2,2% entre les algorithmes LRU et OPT.

La performance du meilleur des algorithmes de remplacement ici présentés s'est révélée généralement supérieure à celle du "shadow cache" [33], une règle de remplacement qui ressemble à certains des algorithmes dans cette thèse.

Chapter 1

Introduction

This introduction assumes a basic familiarity with caches and genetic algorithms. If this is not the case, Chapter 2 reviews fundamentals of cache operation, and Chapter 3 provides basic motivation and theory of genetic algorithms.

An important aspect in determining cache performance is the replacement policy used. In a typical system, main memory may be 10 times slower than cache [36] [41] [23]. In this case improving cache hit rate from 90% to 95% reduces the effective memory access time by more than 30%. Furthermore the gap between cache speed and main memory speed is growing [27].

The problem of increasing hit rate has been handled by:

1. Increasing the cache size
2. Increasing the line size
3. Maintaining separate instruction and data caches
4. Increasing the associativity
5. Prefetching
6. Software "hints" to the cache

7. Investigating alternative replacement policies

The order of this list gives some indication of the relative emphasis that has been placed on different solutions. However, the ranking is not rigid and some may prefer a slightly different order. Nevertheless, the most common replacement policies now in use, such as LRU and FIFO, have been employed at least since the 1960's [36].

There are several reasons for this. Both LRU and FIFO are relatively simple to implement and offer good performance, with LRU generally having the higher hit rate. Other replacement methods have been tried, for example Random, Least Frequently Used (LFU), and Partition LRU—a slightly simplified version of LRU [29] [36]. However LRU has generally been found to have the highest hit rate of all these replacement policies [36]. Upon reflection, the reader will likely find that few additional policies come to mind.

Different cache parameters are most often tested using software simulators. A typical way to do this is to collect "representative" benchmark programs and execute them on an architectural simulator. As part of the simulation, the stream of addresses generated by the benchmark are stored to a file. This *address trace* file is then used as input to the cache simulator.

Variations are possible, such as integrating the cache simulator with the architectural simulator, thereby eliminating the need for the intermediate address trace file. This is important, as address trace files are often many megabytes or even gigabytes. Unfortunately, address trace files are needed here for reasons given in Chapter 4.

One of the advantages of using trace driven cache simulation is that it is possible to determine the optimal performance achievable by any replacement algorithm [6]. This optimal replacement policy is sometimes called MIN, but here it will be referred to as OPT following the convention of [29]. It has been found that OPT often has a miss rate only 70% that of LRU [34]. Thus there is significant room for improvement in replacement policies.

However, if improved replacement policies exist, they may not be obvious. One way to find better policies is to study a large number of address traces for common patterns. Such an undertaking involves such a large amount of data, that some automated method of generating and evaluating policies is required. *Genetic Algorithms (GA's)* provide such a method, and have been used successfully on a wide variety of tasks [21].

Some examples of these include VLSI circuit layout [13], adaptive filter design [17], the travelling salesman problem [8] [22] [42], prisoner's dilemma [5], and job shop scheduling [12]. Other areas of application range from cellular biology to demographics.

In essence, genetic algorithms attempt to mimic evolution: different replacement policies compete, with the fittest surviving and evolving to even fitter policies. Different replacement policies are represented as different bit strings, in a manner analogous to DNA sequences.

As described here, genetic algorithms would be employed during the design of an architecture or possibly at compile time. They would *not* be used on the fly by the cache, as the time and space overhead is far too prohibitive.

The basic notion is as follows. Start with a population of random strings (replacement policies). For each string, run a cache simulation on a benchmark address trace. Record the hit rate or some other performance measure for the string. After all strings have been simulated, *reproduce* the better strings in a new generation of strings. Finally apply additional genetic operators such as *mutation* and *crossover* between strings to find new and better replacement policies.

The goal of this work is to use this genetic algorithm approach to find replacement policies which improve upon traditional policies, in particular, LRU.

Chapter 2

Cache Basics

CPU's have historically been able to operate at faster clock rate than main memory and this imbalance is likely to continue for the foreseeable future. [27]. One means of alleviating this bottleneck has been to employ fast cache memory between the CPU and main memory. However, in order to achieve this speed, the size of caches has been far smaller than that of main memory. (Because they are small, caches can employ more expensive, higher power memory chips than main memory. They can also be placed on the die of modern CPU's, thereby avoiding the speed penalty caused by the large capacitances on the chip's output pins.)

Since caches are smaller than main memory, some means is required to map main memory addresses to cache. There are a multitude of ways to do this, but the basics of almost all caches are the same. Caches function as an associative memory. As is illustrated in Figure 2.1, internally they are almost all divided into 3 parts which determine the overall size of the cache:

- N Sets
- K Lines per Set (K is the Associativity)
- L Bytes Per Line
- Cache Size = $N \times K \times L$

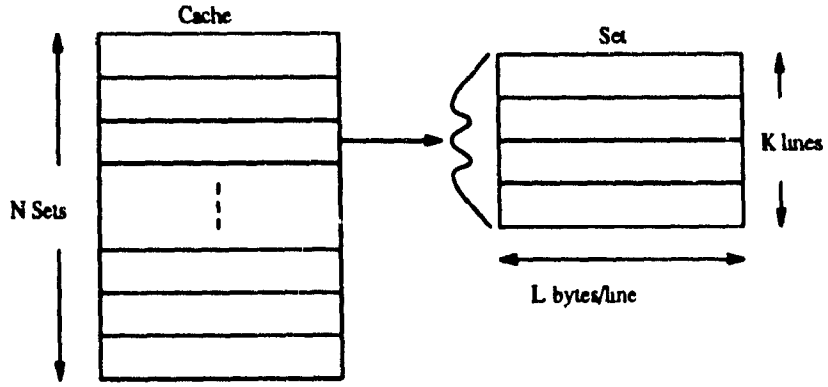

Figure 2.1: Typical Cache Layout.

In the literature, the synonym *block* is often used for *line* [36].

This structure facilitates quickly finding whether a particular main memory address is present in the cache. To see this, consider the steps a sample cache follows in finding whether an address is present:

1. 16-bit Address, *ABCDEFGHIJKLMNOP*
2. 1024 byte Cache
3. K = 4-way associativity
4. L = 8 bytes per line
5. $N = \frac{Cache\ Size}{K \times L} = \frac{1024\ bytes}{4-way \times 8\ bytes} = 32$ sets
6. Address Bits *NOP* specify which of the 8 bytes in the line
7. Address Bits *IJKLM* specify which of the 32 sets
8. Address Bits *ABCDEFGH* form a *tag*
9. Use $K = 4$ comparators to determine if the tag is present in the set

Note than every memory address maps to a specific cache set, and hence one of K lines. The cache must determine whether a value requested by the CPU is present and return it as quickly as possible. Hence the number of tag comparisons (K) performed to check if a value is present must be minimized. Even if all the comparisons are performed in parallel, there are problems: a large amount of space is required for the comparators and the larger space implies a reduced maximum clock frequency. Peak power consumption is increased. On the other hand, too small an associativity, K, can cause problems when many memory addresses to map to the same set. The result is *thrashing* while the different addresses compete for space in the set. Typically K is 2 or 4, but values from 1 to 16 have been used [36]. K is typically a power of two, but as can be seen in the example above, it need not be. [31] for example uses a $K = 3$-way associative cache. N and L must be powers of 2.

As can also be seen in the example, the lower order bits are used to map addresses to sets. As one might expect this acts as a quasi-random function to evenly distribute addresses among all sets. A random mapping generally minimizes the number of multiple addresses competing for space in the same set, i.e. it minimizes thrashing. However, since caches are smaller than main memory, conflicts sometimes occur. If each of the K lines in a set contains an address, then one of the K addresses must be replaced, when a new line also maps to that set.

There are several common algorithms which are used to decide which line to replace. The algorithms must be simple as usually they must operate in one or two CPU cycles.

- **LRU**, the *Least Recently Used* line is replaced.
- **Partition LRU**, a simplified approximation to LRU. Here the K lines in a set are partitioned into subsets. For example, an 8-way associative cache may have 4 subsets of 2 lines each. The order of use of each subset is maintained, as is the order *within* each subset. On a miss the least recently used line in the least recently used subset is replaced. *Partition LRU* is used mainly in "highly" associative ($K \geq 4$) caches.
- **LFU**, the *Least Frequently Used* line is replaced.
- **FIFO**, the *First In* line is the *First Out*.
- **Random**, a pseudo-randomly chosen line is replaced.

- **OPT**, the *optimal* strategy where the line which is used furthest in the future is replaced [6].

Since future information is unknown to the cache, OPT cannot be used in real caches. However software cache models are commonly used to test a specific cache configuration [36] [37] [1] [16]. These models take as input, a trace of memory addresses generated by a particular benchmark or benchmarks. (Methods of generating such a trace are described below.) A prepass can be done on this trace to determine how long until each line is referenced again. OPT can then use this information on a second pass to expel the line referenced furthest in the future. The performance achieved using OPT provides a good basis against which other strategies can be compared. It is actually possible to write a single pass OPT [7]. LRU is the most common and generally the best performing of the (implementable) algorithms above [36]

In order to determine which replacement and which cache parameters are best, some measure of cache performance is required. There are several:

- *Hit Rate*, the percentage of time that the location desired by the CPU is present in the cache. This metric is a good measure of overall cache performance.

- *Line Hit Rate* [39], similar to *hit rate*, but for a given set, the only accesses considered are those which are to a *different* line than the previous time the set was accessed. Any series of accesses *within* a single line produces hits no matter what replacement algorithm is used. As a consequence maximizing the *line hit rate* also maximizes the *overall hit rate*. However, the *line hit rate* more closely reflects the difference between two replacement policies.

  The behavior of *line hit rate* can be surprising. If the metric used is *overall hit rate* and one of the cache size parameters—line size (L), associativity (K), or number of sets (N)—is increased while the others are held constant, then the *overall hit rate* is increased (or possibly held constant). However if the metric used is *line hit rate*, this is not true.

  This is because the *line hit rate* considers only those references which access a different line than the previous time a given set was accessed. When the cache size is increased, the number of references in which a new line is referenced is generally decreased. The

line hit rate of this decreased number of new line references may be lower than for a smaller cache with more *new line* references.

- *OPT Match Rate*, the percentage of time that the line replaced is the same line that OPT would have replaced.
- *Average Access Time*, the average number of cycles the cache takes to return a value to the CPU. This is directly related to the *hit rate*.
- *Memory Traffic*, the number of transfers between cache and main memory. This is especially important for a multiprocessor system with global memory and where each processor has a local cache.
- *Utilization*, the number of times each location in the cache is accessed. This is a measure of the value of a cache compared to alternative uses of the chip or board space. If *utilization* is low, it may be a better use of space to have two simple processors with small caches, than one complex processor with a single larger cache.

Here we are most interested in the effectiveness of replacement algorithms. Hence *hit rate*, *line hit rate*, and *OPT match rate* are used. Trying to find a replacement policy which matches OPT is one way of having the cache recognize certain access patterns and respond to them, hopefully in an optimal way.

Although most caches are organized as has been described, a single CPU need not have only one *unified* or *combined* cache. The CPU may have multiple caches, each containing distinct information. Two types of divisions are most often suggested [36]:

- Instruction Cache and Data Cache
- Supervisor Cache and User Cache

Separate supervisor and user caches are designed to increase *hit rate* and *utilization* by keeping user code in cache during interrupts. These goals are not always accomplished. This is because two, half-size caches can be too small to accommodate the *working set* of references, whereas one larger cache would be sufficient. The larger cache also dynamically shifts what fraction of it is used for one purpose and what fraction is used for the other. For this reason, separate supervisor/user caches are rarely used [36].

Separate instruction and data caches have two other advantages. Bandwidth from cache to processor is increased—barring cache misses the processor can be continuously fed with instructions and data. An instruction cache can also be simpler (and hence larger and faster) since it need never be concerned with memory writes. (Most modern architectures assume that self-modifying code is not used.)

A CPU may also have multiple levels of cache. In a multi-level system, instead of placing a single cache in between the CPU and main memory, an additional, generally slower cache is placed between the first cache and main memory. This hierarchy may be extended arbitrarily deep, but there is generally not enough difference in speed between main memory and the CPU to warrant more than two levels of cache [41].

Finally as already noted, most cache study is done using software cache simulators and address traces. Some methods of trace collection are

- Special instrumentation and recording hardware for existing systems.
- Software architectural simulators.
- *ATUM* or Address Tracing Using Microcode [2]. Patches are made to microcode to record all address references made by the machine including operating system calls and interrupts.

Architectural simulators are simplest, but have the drawback that they usually generate traces from only a single (user) program. Obtaining and properly mixing addresses from system code and interrupts is virtually impossible. Microcode patches become less viable as RISC (no microcode) and VLSI (microcode not patchable) grow ever stronger. Instrumentation hardware is too expensive and too complicated to be of general use. It is especially difficult to synchronize trace collection with the execution of specific programs.

Trace simulation allows complete flexibility as to cache parameters, and even allows the OPT replacement policy to be used. However, trace simulation is also orders of magnitude slower than an actual cache and deals with traces that almost always represent very little computing time. 20 million addresses might represent only one second of computing in a modern RISC processor, but take up 80 megabytes of disk space.

This large disk space requirement has spurred some researchers to integrate their cache simulator more fully with an architectural simulator [26]. Desired benchmarks are executed on the simulator and as address references are generated, they are immediately passed to the cache simulator, which maintains the desired statistics. Unfortunately this approach is not practical here. As will be seen, the same trace must be reused hundreds and thousands of times. The additional overhead of simulating an entire architecture running the desired benchmark is prohibitive.

An alternative to trace simulation or architecture/cache simulation is to have the cache maintain CPU addressable counters for the number of hits and the number of misses [4]. The CPU could clear these counters and later record the number of hits and misses at any desired point. Incrementing of the counters can be done in parallel with other cache activity and hence need not slow it down. This method could provide the hit rate for the system including all interrupts and system code. With a small additional bit of hardware, the *line hit rate* could also be maintained in this manner. Memory traffic could also be accurately measured via onboard counters.

Clearly, this counter scheme could not be used directly in developing new architectures. However if it were generally implemented, cache performance statistics would be available for a much wider variety of machines and workloads than is presently the case.

Chapter 3

Genetic Algorithms

The goal of genetic algorithms is to mimic evolution to find optimal or near optimal solutions for a given problem[1]. The problem must be well defined, and a function must exist to evaluate proposed strategies. A number of approaches for genetic algorithms have been proposed. Almost all modern approaches use a population of binary strings (or strategies) [21], all of the same length, l-bits Each string represents a possible solution.

Any moderately complex problem has more solutions than can reasonably be enumerated and searched for an optimum. If $l = 32$, exhaustive search requires that $2^{32} \approx 4$ billion strategies be evaluated to find the optimum of the function Genetic algorithms attempt to reduce the search space to a manageable size at the risk of finding only an approximately optimal solution.

A simple genetic algorithm, typical of many modern approaches, is outlined below:

1. Generate a random population of l-bit binary *strings*.
2. Evaluate fitness of all strings.
3. Use fitness to determine which strings to reproduce in the next generation.

[1]The exposition in this Chapter borrows heavily from descriptions given in Goldberg's comprehensive 1989 book [21]

4. Original generation "dies".
5. Pair off strings in new generation.
6. For each pair, randomly choose a bit position, p in the string.
7. Swap the bits to the right of p between the two strings in the pair. *(Crossover)*
8. Randomly mutate (with small probability) bits in resulting population.
9. Goto Step 2.

This algorithm uses three operations to systematically seek improvements in the population, or in other words to search for an optimum solution.

- *Reproduction* rewards the fittest strategies.
- *Crossover* generates new strategies and rewards good substrings.
- *Mutation* adds diversity and insures that all strings may be generated at all times.

To make matters more concrete, consider the following example which goes through one iteration of the basic algorithm above.

1. Initial Population: $A = 0111000$, $B = 0101010$, $C = 1001001$, and $D = 1010110$
2. Assume A, C, and D are best by some criteria and two A's, one C, and one D are reproduced in the next generation.
3. Assume the pairing is A/C and A/D.
4. Randomly choose 3 as the crossover position for A/C, and 5 as the crossover position for A/D.
5. This yields: $A' = 0111001$, $B' = 1001000$, $C' = 0111010$, and $D' = 1010100$.
6. Randomly choose the 2nd bit of C' and the 7th bit of D' to mutate.
7. The new generation is: $A' = 0111001$, $B' = 1001000$, $C' = 0011010$, and $D' = 1010101$.

A simple and innovative use of genetic algorithms was proposed by Axelrod and Forest [5] [18] for the game, Prisoner's Dilemma. In this game, two accomplices are held in separate rooms by police as suspects in some crime. In truth both prisoners are guilty However, the suspects will receive longer or shorter sentences depending on whether both, neither or one of them confess. To make the game more interesting, this sequence is repeated for a large number (150) of moves, thereby allowing each of the two prisoner's to learn about the other's behavior.

At a 1985 computer tournament, a tit-for-tat strategy was victorious over many more complicated strategies. (In tit-for-tat, one does what the opponent did in the previous move.) Using a genetic algorithm approach, Axelrod and Forest found a strategy that consistently beat tit-for-tat as well as the other strategies in the tournament Their approach can be summarized (and slightly simplified) as follows:

1. At each move each prisoner can confess or not. Hence each move can be recorded with 2 bits.
2. Record the 6-bit history of past 3 moves.
3. Choose the next move based on this 6-bit history.
4. 6 bits $\Rightarrow 2^6 = 64$ possible histories.
5. For each of the 64 histories, the strategy must indicate whether to confess or not.
6. *Goal*: Find the optimal 64-bit strategy.
7. *Method*: Generate a random population of 64-bit strings and use genetic algorithm as described above. At each generation, play each string against every other. A string's fitness is a (nonlinear) function of the total amount of prison time incurred by that string.

Although genetic algorithms have some appeal merely by analogy to nature, it can be shown in a more mathematically rigorous manner, that the operations of reproduction, crossover, and mutation will tend to produce successively more optimal populations [24]. To begin a pair of definitions are needed.

- A *schema* is a binary string, but not all bit positions need be defined. *Don't Cares*, *, are allowed. For example, $H = 011*1**$. (The plural of *schema* is *schemata*).
- The *defining length*, $\delta(H)$ of a schema, H, is the number of bits between the first and last defined bit position. For example, $\delta(H = 011*1**) = 4$.

These definitions can now be used to show that over time, *reproduction* creates exponential growth in the number of above average schemata in the population.

- Let there be a population, P containing n strings of length l at time t.
- Let the fitness of string, j be f_j.
- Let the average fitness of strings representing schema, H in this population be $f(H)$.
- Let $m(H,t)$ be the number of strings containing schema, H at time t.
- Then the expected number of strings containing H after reproduction at time $t+1$ is proportional to $\frac{f(H)}{\sum_1^n f_j}$:

$$\begin{aligned} m(H,t+1) &= m(H,t)\frac{nf(H)}{\sum_1^n f_j} \\ &= m(H,t)\frac{f(H)}{\bar{f}} \end{aligned}$$

- If strings containing H are a factor c better than average at each generation, then

$$\begin{aligned} m(H,t+1) &= m(H,t)\frac{(\bar{f}+c\bar{f})}{\bar{f}} \\ &= (1+c)m(H,t) \end{aligned}$$

- This produces exponential growth in number of above average schemata and exponential decay for below average schemata.

$$m(H,t) = (1+c)^t m(H,0)$$

Reproduction never produces new strings. Alone it can never improve on the performance of the original population. *Crossover* does introduce new strings, and does so in such a way as to reward highly fit, short schemata. In other words good, small building blocks are favored. To see this, consider an example.

- Let $H_1 = *1****0$ and $H_2 = ***10**$. Then $\delta(H_1) = 5$ and $\delta(H_2) = 1$.
- There are 6 possible crossover locations for both H_1 and H_2. H_1 is destroyed by $\delta(H_1) = 5$ of the 6 choices. H_2 is destroyed by only $\delta(H_2) = 1$ of the 6 crossover choices.

More generally, this idea can be summarized as follows.

- If crossover occurs with probability, p_c, the probability, p_s, of schema, H surviving is

$$p_s(H) = 1 - p_c \frac{\delta(H)}{l-1}$$

- In pure reproduction, $p_c = 0$ and $p_s = 1$. Allowing crossover changes the number of occurrences of schema, H expected in the next generation:

$$\begin{aligned} m(H, t+1) &= m(H,t)\frac{f(H)}{\bar{f}} p_s(H) \\ &= m(H,t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{l-1}\right] \end{aligned}$$

- Now the number of schemata, with short defining lengths, $\delta(H)$ grows exponentially: the system rewards good, small building blocks.

Note that in this derivation, we ignore the small chance that crossover of two schemata not containing H will create an instance of H in the new generation. Including this effect does not change the main result [21].

Here *mutation* is assumed to occur as inversion of single bits. Multiple bits in a string may mutate, but each occurrence is assumed to be independent. Mutation has two roles. It can add diversity to a population, as well as guarantee that it is always possible to generate

any string. Always being able to generate any string is necessary to avoid getting stuck at a suboptimal solution when all strings in the population are identical at certain bit positions. Mutation also has a danger. If the mutation rate is too high, then the progress made through reproduction and crossover can be destroyed. Formally, the effect of mutation is as follows.

- Let p_m be the probability of a bit mutating.
- Define the *order* of a schema, $o(H)$ to be the number of fixed positions in H. For example, if $H = 011*1**$, then $o(H) = 4$.
- The probability of H surviving mutation is

$$p_{msur} = (1 - p_m)^{o(H)} \approx 1 - o(H)p_m \text{ for } p_m \ll 1$$

- The probability of H not surviving mutation is

$$p_{mdie} \approx 1 - [1 - o(H)p_m] = o(H)p_m$$

Subtracting the destruction rate of schemata due to mutation from the previous recurrence for number of schemata in a generation yields *The Fundamental Theorem of Genetic Algorithms:*

$$m(H, t+1) = m(H, t)\frac{f(H)}{\bar{f}}\left[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right]$$

Chapter 4

Genetic Algorithms Applied to Cache Replacement

4.1 Basics

As noted in Chapter 2, LRU is currently the most common replacement strategy used in caches. To improve upon it, either more or different information must be kept about the lines residing in cache. One way of using *more* information is to use both LRU and FIFO information in deciding which line to replace. Keeping a *history* of cache accesses is a way of using *different* information. A *history* could be kept and used with a genetic algorithm as follows:

- For the previous m accesses to a cache set, record whether the access is a hit or miss.
- If the access is a hit, record to which line it is a hit.
- If the access is a miss, record which line was replaced.
- If K is the cache associativity, then a genetic algorithm would require a population of strategy strings, each of B bits, where B is

$$B = log_2(K) \times 2^{m[1+log_2(K)]}$$

The derivation of this is quite simple. For each of the previous m accesses $1 + log_2(K)$ bits are used to record what happened on that access. Thus there are $2^{m[1+log_2(K)]}$ possibilities for the previous m accesses. For each of these possibilities, the line to be replaced must be specified. This takes $log_2(K)$ bits, which yields the formula for B given above.

- For K = 4-way associativity and a *history* of the last 4 accesses, each string is 8192 bits! For comparison [10] used some of the longest strings in genetic algorithm applications. They were the equivalent of less than 4000 bits. There are ways to reduce the complexity here as is discussed briefly below in Section 4.4 and more fully in Section 5.4.1.

Regardless of what approach is used—a combination of LRU and FIFO, *history*, or something else—some means is needed to find good solutions in the myriad of possible solutions. As noted, if even a relatively short history is kept, a brute force method would entail a search of 2^{8192} possibilities. Genetic algorithms offer an attractive alternative to such a method.

Note that only one approach—an *LRU-FIFO* combination, *history*, etc—is tried at a time. A genetic algorithm tries to find a near optimal strategy within that one approach. However, for each approach, the same basic method is used to find a good replacement strategy:

1. Generate a random population of strategies (strings) for *the* approach (*LRU-FIFO*, *history*, etc).
2. For each string in the population, perform a cache trace simulation using a particular benchmark or set of benchmarks.
3. During each simulation, use the string to determine which line to replace when a cache miss occurs.
4. After each simulation, record the *line hit rate* or some other performance measure for the string.

5. Calculate the fitness of the string as a nonlinear function of the performance measure.
6. After a simulation has been performed for each string, use reproduction, crossover, and mutation to generate a new generation of strategies based on these fitnesses.
7. Goto step 2.

4.2 Example

To make this more concrete, consider an example using the *history* method. Let the cache have $K = 4$-way associativity and keep a history of $m = 1$ previous accesses. This requires

$$B = 2 \times 2^{1 \times [1 + log_2(4)]} = 16 \; bits$$

for each strategy string. More intuitively

- The history of the previous access is recorded in 3 bits, $Hist = B_2 B_1 B_0$. The 3 bits refer to the line referenced (2 bits) and whether the access was a hit or a miss (1 bit).
- The strategy must indicate which line to replace on a miss for each of the $2^3 = 8$ possible histories.
- 4-way associativity $\Rightarrow$ 2 bits to specify the line to be replaced.
- Since 8 lines must be specified, $8 \times 2 = 16$ bits are required for each strategy string.

Several additional assumptions are required:

- Let $B_2 = 1$ for a hit, 0 for a miss.
- Let $B_1 B_0$ specify the line to be replaced.
- Let the Strategy string S under consideration be

$$S = \overset{7}{01}\ \overset{6}{00}\ \overset{5}{10}\ \overset{4}{00}\ \overset{3}{11}\ \overset{2}{01}\ \overset{1}{11}\ \overset{0}{10}$$

  S can be viewed as an array of 8, 2-bit elements, one for each possible history. Each element specifies which line to replace for a given history.

- Assume $Hist = 101$ and there is a miss.

The history method then works as follows

- The line replaced is that specified by the $B_2B_1B_0 = 101_2 = 5th$ pair in the strategy, $10_2 = 2$.
- Thus the new $Hist = 010$: a miss occurred and line 2 was replaced.
- Next assume there is a hit to line 0 of the set.
- Therefore the new $Hist = 100$.

The string S is used in this manner for the entire address trace, i.e. based on the past history of accesses to a set, S indicates which line should be replaced when a miss to that set occurs. When the simulation finishes with the address trace, the performance of S is noted. Then the same procedure is repeated for the rest of the strings in the population. Finally after the performance of all strings has been measured, the reproduction, crossover, and mutation operators are applied in an attempt to produce a new and hopefully improved generation. This process continues, generation after generation until each generation performs approximately as well as its predecessor. (There is no hard and fast rule for determining when this convergence has occurred [21].)

Clearly the choice of address trace is critical to this process. (The address trace corresponds to some benchmark program or programs.) Some strings perform well with some programs, others with different programs. One goal is find a single string which performs well, although perhaps never optimally for a wide variety of traces. *Well* might be defined as better than LRU.

Alternatively the replacement string could be dynamically specified to the cache. This would allow different applications to use different replacement policies. Such an approach raises the question: how does the application know what is a good replacement policy? One way is to generate an address trace from the application and use the complete genctic algorithm procedure described above. Another possibility is for the compiler to use heuristics to specify the strategy. For example a large number of deeply nested loops might suggest

one replacement strategy, and recursion another. Using this approach it might even be possible for each subroutine or procedure to specify its own replacement strategy.

A third alternative would be for the cache itself to recognize certain access patterns. Upon recognizing a pattern, the cache would use a corresponding replacement strategy. Patterns would probably have to be relatively simple in order that the cache could recognize them in real time. (For example, the full blown genetic algorithm approach described above is clearly too complex for a cache to perform in real time.)

4.3 Combination Approaches

In addition to *LRU-FIFO*, several "combination" approaches are possible. Those combination approaches implemented here are listed below along with number of bits they require in each strategy string. The computation of the number of bits is similar to that described above for *history*. Note that in all cases the number of bits B is a multiple of $log_2(K)$, since $log_2(K)$ bits are needed to specify which line is to be replaced.

1. *Combination of LRU and FIFO information* (**LRU-FIFO**). In this case the replacement algorithm must know the FIFO rank of each LRU line. (In this document the term, "LRU line" is sometimes used to mean a line's rank from most recently used to least recently used. In such cases it does not mean literally the *least recently used* line. Context should make clear whether this or the literal meaning is intended.)

   However most LRU to FIFO mappings are impossible. LRU lines 1 and 2 cannot both correspond to FIFO line 4. For example the following mapping is legal:

   ```
   1 -> 4
   2 -> 2
   3 -> 3
   4 -> 1
   ```

   While this is not:

```
1 -> 4
2 -> 4
3 -> 3
4 -> 1
```

Formally the LRU to FIFO mapping must be 1:1, i.e. each LRU line must correspond to a single FIFO line and no other LRU line may correspond to that same FIFO line.

There are $K!$ ways the LRU ordering can map to the FIFO order. Hence $B = K!\ log_2(K)$. This produces relatively short strings. For example, if $K = 4$, then $B = 48$ bits.

A more brute force method could be used in which an explicit LRU to FIFO table is maintained. This table would have K entries each of $log_2 K$ bits or $K log_2 K$ bits total. In this method $B = 2^{K log_2 K} log_2(K) = K^K log_2(K)$. Using the same example with $K = 4$ yields a far larger $B = 512$ bits.

2. *Combination of LRU and a c-bit count of hits to each line* (**LRU-Count**). Note that the count must stop incrementing when it reaches $2^c - 1$. Here cK bits are required to record the number of hits for all the lines, and hence there are 2^{cK} possible combinations of hits and lines. Thus $B = 2^{cK} log_2(K)$. This produces strings of intermediate length. For example, if $K = 4$ and $c = 2$, then $B = 512$ bits.

3. *Combination of LRU and an h-bit history (hit/miss) of accesses to each set* (**LRU-History**). With an h-bit history, there are 2^h possible histories, so $B = 2^h log_2(K)$. This produces short strings. If $K = 4$ and $h = 4$, then $B = 32$ bits.

4. *Combination of FIFO and a c-bit count of hits to each line* (**FIFO-Count**). The analysis is the same as with LRU and a c-bit count, hence $B = 2^{cK} log_2(K)$. If $K = 4$ and $c = 2$, then $B = 512$ bits.

5. *Combination of FIFO and an h-bit history (hit/miss) of accesses to each set* (**FIFO-History**). The analysis is the same as with LRU and an h-bit history, hence $B = 2^h log_2(K)$. If $K = 4$ and $h = 4$, then $B = 32$ bits.

4.4 Hardware Implementation

The amount of space and time required for the approaches described is generally small. Each method requires combinational logic, PROM, or RAM to encode the string corresponding to the replacement policy. When a cache miss occurs, the function must produce the $log_2 K$ bit value of the line to replace, where as usual K is the cache associativity, and is typically less than or equal to 4 [36]. Different replacement functions require different input: *history* requires a record of hits, misses, and lines accessed while *LRU-FIFO* requires the mapping from LRU to FIFO. Let the replacement strategy be a function of B_I bits. Note B_I is different than B, the number of bits in a strategy string. However, the two are related:

$$B_I = \left\lceil log_2 \left(\frac{B}{log_2(K)} \right) \right\rceil$$

A block diagram of the cache replacement hardware is in Figure 4.1.

A more detailed hardware description for each method is given below. Note that standard cache hardware is not covered, instead only what is unusual or extra for these methods is discussed.

1. **LRU-FIFO.**

   $B_I = \lceil log_2 K! \rceil$. If $K = 4$, then $B_I = 5$ bits. Note that the brute force approach would require $B_I = K log_2 K$ bits.

   To implement this, the LRU to FIFO mapping must be updated whenever a set is referenced. A block diagram of this is shown in Figure 4.2. The mapping must be updated differently depending on whether the reference results in a hit or a miss. This *hit* logic must also know which line is the new MRU. For the *miss* logic, the new MRU is just the replaced line. The replacement line specified on a miss is not a physical line in the set, but instead the LRU ranking of the line to be replaced—for example, replace the 2nd least recently used line. Hence a second translation would be necessary to obtain the physical cache line.

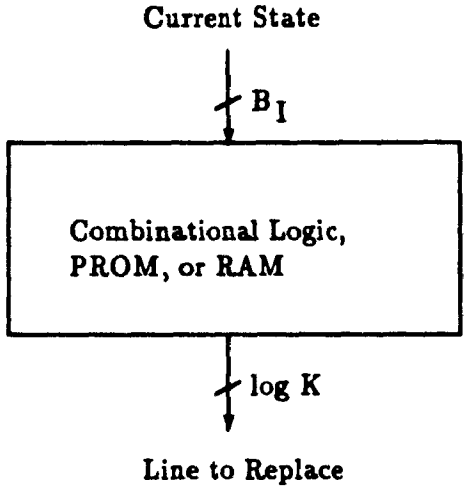

Figure 4.1: Block Diagram of Cache Miss Hardware.

The combinational logic in Figure 4.2 could equally well be PROM or RAM as previously noted. All sets could share a single copy of the combinational logic portion of the mapping hardware. Each set must however, maintain the register which maps LRU lines to FIFO lines.

2. **LRU-Count** or **FIFO-Count.**

   $B_I = cK$. If $K = 4$ and $c = 2$, $B_I = 8$ bits. The c-bit counter must be updated whenever there is a hit to the line. The counter must also stop when the count reaches its maximum and be resettable to 0 when the line corresponding to it, is replaced. Unlike LRU-FIFO, the actual physical line to be replaced is output from the combinational logic. This is because the combinational logic requires the LRU to physical line mapping in order to determine which LRU line to associate with each c-bit counter. Since the logic already has the physical mapping it makes use of it by providing the physical line as output. A block diagram of the circuitry required on a miss is shown in Figure 4.3. On a hit, all that is required is a demultiplexer to map the line with the hit to an increment signal for that line's counter.

3. **LRU-History** or **FIFO-History.**

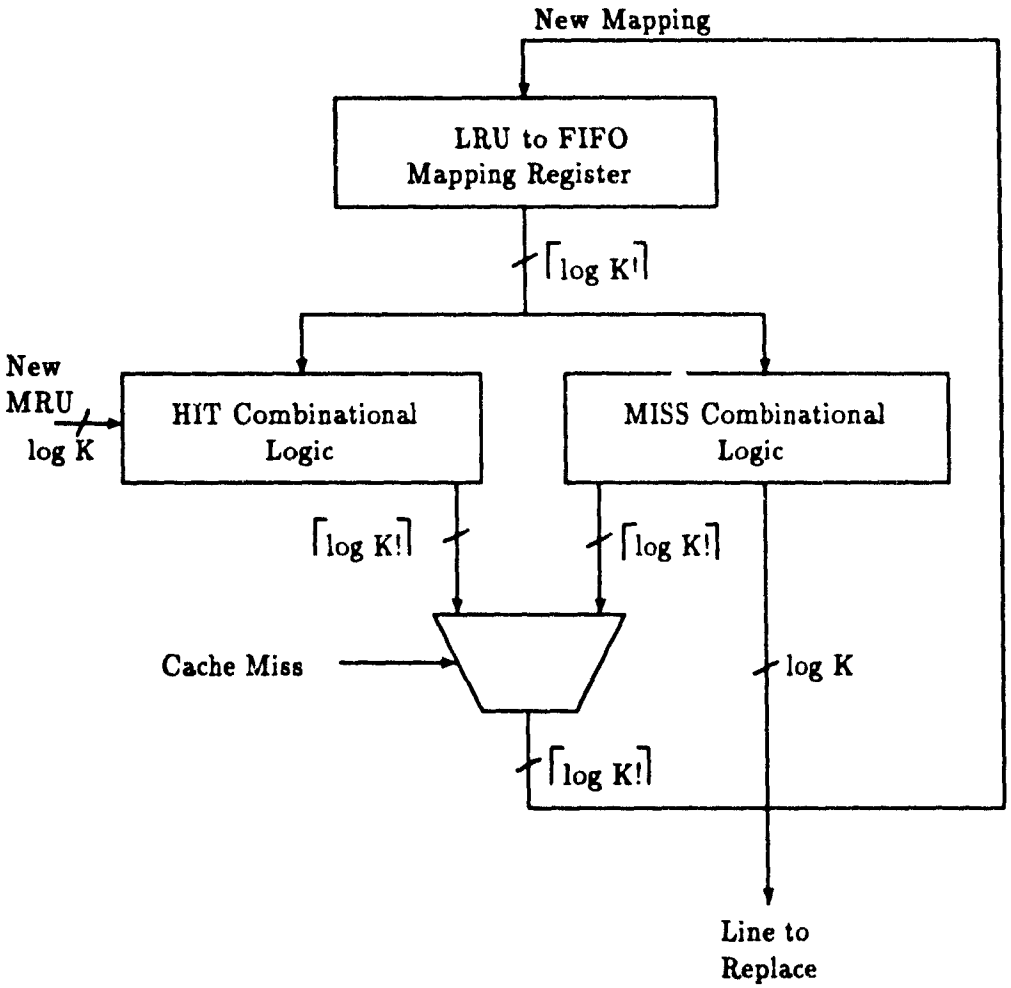

Figure 4.2: Hardware for *LRU-FIFO* Method.

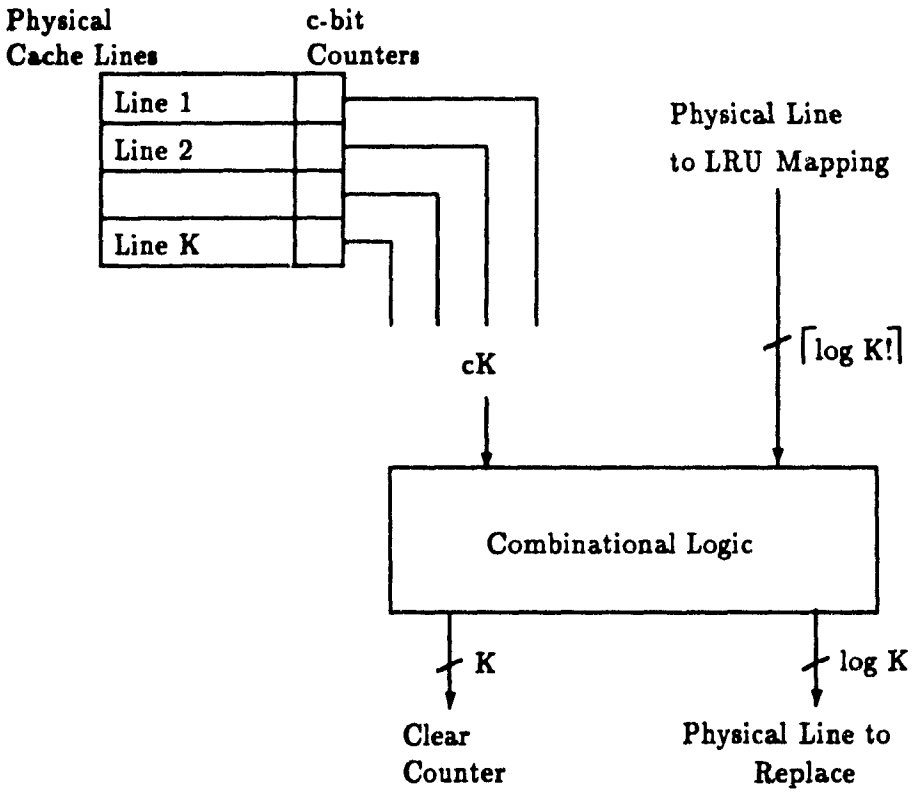

Figure 4.3: Hardware for *LRU-Count* Method on Misses.

$B_I = h$. If $h = 4$, then $B_I = 4$ bits. This method is particularly simple on both hits and misses: on a miss, a 0 is shifted into an h-bit shift register, while on a hit a 1 is shifted in. A simple function of h bits specifies which line to replace on a miss. In this case the LRU ranking of the line is specified and must be mapped to the physical line.

4. **History.**

   $B_I = m(1+log_2K)$. If $K = 4$ and $m = 4$, $B_I = 12$ bits. Recall that at the start of this chapter, it was stated that the complexity of the history method can be significantly reduced. Unfortunately the simplification applies only to genetic algorithms searching for a good replacement strategy, not to the hardware implementation.

   The simplification uses the fact that many histories are actually equivalent. To make use of that here would require a three step process· 1) Reduce to the equivalent form, 2) Determine line to replace, and 3) Tranlate the equivalent replacement line to the actual replacement line. This is almost certainly more complicated than implementing the function directly. For more details see Section 5.4.1.

   The implementation is simple and similar to *LRU-History*. Each time a hit occurs to a line, this information is shifted into a $1+log_2K$ bit wide shift register—the extra "1" is for the hit/miss information. Likewise when a line is replaced on a miss, the line number and miss bit are shifted into the shift register. This is depicted in Figure 4.4. Since this method manipulates lines directly without the need of correlating LRU or FIFO information, the line specified for replacement is the physical line.

As can be seen the time required for these methods is also generally small. Actually there are two separate times, the time for a cache hit and the time for a cache miss. The cache hit time is most important because most accesses are hits. Most cache studies find hit rates well over 80% and often greater than 90% [36] [1] [3] [11]. As was seen the methods proposed here require little time on a hit: generally a counter must be incremented or a shift register updated. *LRU-FIFO* is the most complicated. Miss time is generally small too, consisting of the combinational logic or PROM delay to find which line should be replaced plus time to clear a counter or update a shift register.

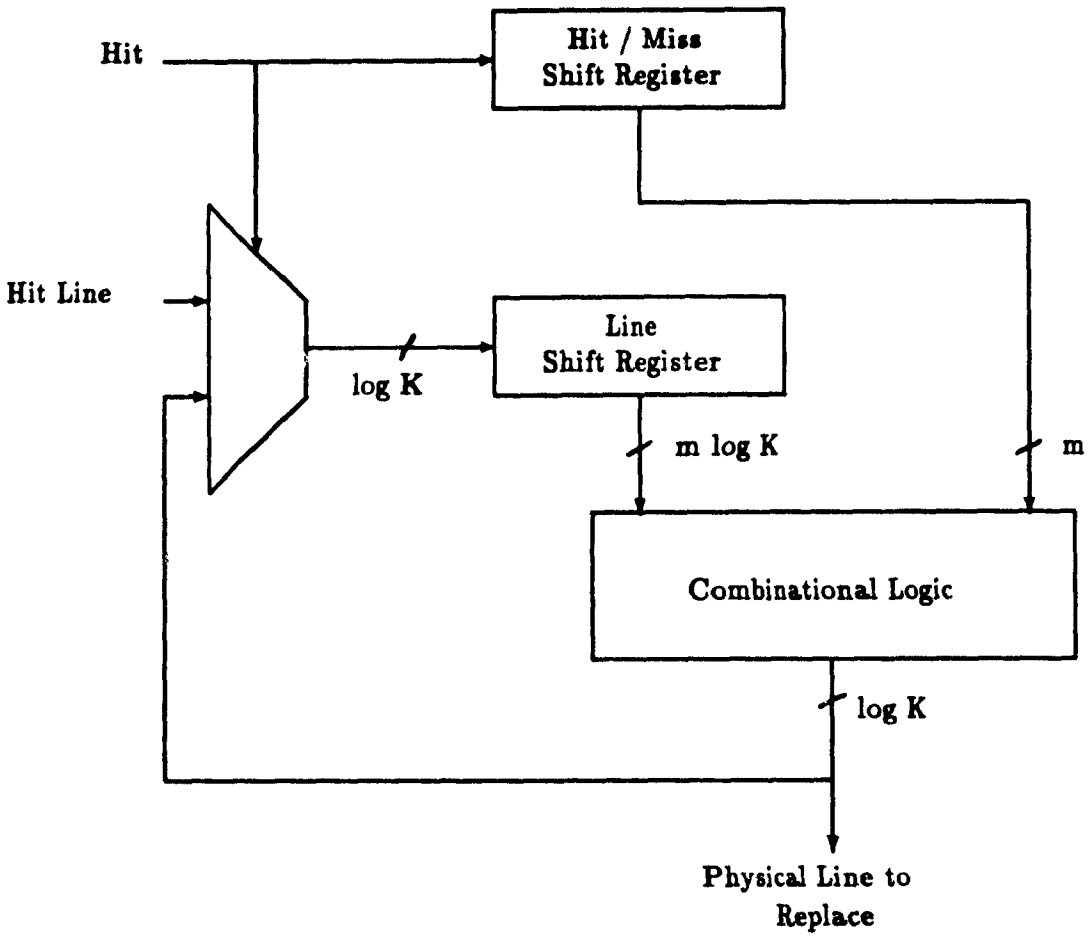

Figure 4.4: Hardware for *History* Method.

Chapter 5

Simulations

5.1 Methodology and Details

The cache simulator used in this work was developed by R.A. Olsen, a student in the McGill Advanced Computer Architecture and Program Structures (ACAPS) Group [32]. The simulator has been extensively modified and supplemented by the author. The genetic algorithm portion of the simulator follows the outline presented in [21].

All code was written in the *C* language. The source contains over 9000 lines of code, while the optimized executable SPARC version has a *text* size of 81920 bytes.

Traces from six benchmark programs have been used throughout [1]. The benchmarks are drawn from a variety of applications, some numerically intensive, others not. The six benchmarks are:

1. ccal† (PASCAL): Simulates a simple pocket calculator.

[1] All simulations were run on Sun-4 SPARC (tm) architectures. The time to complete an individual simulation (one benchmark and one genetic algorithm approach) was typically 30 minutes to 6 hours depending on the parameters used.

2. **emacs** (): The common text editor program.

3. **kalman†** (FORTRAN): A kalman filter routine.

4. **mcsh** (): The "Mouse" shell, an enhanced emacs-like UNIX c-shell.

5. **poly†** (FORTRAN): Symbolic polynomial manipulation.

6. **whetstone†** (FORTRAN): The standard floating point benchmark program.

Items with a † are benchmarks distributed by Stanford University with their *Architect's Workbench* program.

The traces correspond to execution of the benchmarks on a SPARC architecture. Benchmarks were first compiled using standard UNIX *C*, FORTRAN, and *PASCAL* compilers. Traces were then collected by simulating program execution on a SPARC architectural simulator developed by Sun Microsystems.

The *ccal*, *kalman*, *poly*, and *whetstone* traces correspond to complete program execution. The other two, *emacs* and *mcsh* correspond to start of program execution. All traces are relatively small by modern standards, ranging up to a few hundred thousand addresses at most. However, the number and duration of the simulations required that the length of traces be kept relatively low. The exact number of addresses and number of unique addresses are given in Table 5.1. Note that in the Table the sum of instruction addresses and data addresses is slightly less than the number of combined addresses. This is because the traces also contain a small number of system traps which are not considered to be in either category.

All results given here refer to separate instruction and data caches of 512 byte caches each, and both with 4-way associativity. The use of separate instruction and data caches is in keeping with most modern architectures, for example the 68040 [15], the R6000 [28], the NS32532 [30], the Clipper [25], and the i860 [23].

The choice of 4-way associativity reflects the maximum value used in most modern caches [36]. Use of an associativity less than 4 minimizes the effect of the replacement policy. With an associativity of 1 (direct-mapped), there is no choice for the replacement policy, and for an associativity of 2 or 3, the choice of lines to replace is limited.

	INSTRUCTION		DATA		COMBINED	
	Unique	Total	Unique	Total	Unique	Total
CCAL	4721	53570	880	15128	5602	68803
EMACS	8525	205936	7195	50982	15721	256973
KALMAN	6258	73990	2085	17076	8344	91261
MCSH	2050	309929	2618	78024	4669	388012
POLY	4913	42679	2266	11975	7180	54835
WHETSTONE	2154	14046	1530	3811	3685	17914

Table 5.1: Number of Total Addresses and Number of Unique Addresses in Each Benchmark.

A 512 byte cache is quite small for modern processors, even for an onchip cache. This small size was chosen in order to emphasize the effect of the replacement policy. Using a larger (but still relatively small) 4K cache, results in little difference in hit rates between different replacement policies. Using a 512 byte cache results in overall hit rates of between 85% and 95% for most benchmarks and replacement policies. Table 5.2 gives *overall hit rates* for caches with 32 byte lines. Table 5.3 gives the corresponding *line hit rates*.

The results from two types of approaches are presented. *First* are results where a genetic algorithm approach was used to find a good strategy for each benchmark individually. This is useful for a cache with a small RAM containing the replacement strategy. The operating system could load this RAM with the appropriate strategy whenever the program is run. This approach is also useful in providing a rough upper bound on how well the second approach can do.

The *second* approach provides results from using a genetic algorithm to find a good strategy for a group of benchmarks. If these benchmarks represent a "typical" workload, then the best strategy can be hardcoded into combinational logic or a PROM in the cache.

The replacement policies simulated, *LRU-FIFO*, *LRU-Count*, *LRU-History*, *FIFO-Count*, *FIFO-History*, and *History* are as described in Chapter 4, with the exception of *History*. As described in Chapter 4, the *History* method maintained for each set a record of lines

	Instruction			Data		
	LRU	FIFO	OPT	LRU	FIFO	OPT
CCAL	88.8	88.8	91.0	89.6	88.4	92.1
EMACS	95.0	95.0	95.9	94.3	93.7	95.4
KALMAN	87.4	87.3	90.3	85.7	84.4	88.9
MCSH	92.3	92.1	95.5	94.1	92.7	95.6
POLY	90.7	90.6	92.7	85.5	84.3	88.1
WHETSTONE	91.5	91.2	93.8	82.6	82.3	85.0
Mean	91.0	90.8	93.2	88.6	87.6	90.9
SUITE	92.2	92.1	94.5	92.1	91.0	93.8

Table 5.2: Overall Hit Rates for 512 byte Instruction and Data caches with 4-way associativity, 32 byte lines.

	Instruction			Data		
	LRU	FIFO	OPT	LRU	FIFO	OPT
CCAL	29.4	29.2	43.2	67.7	64.0	75.6
EMACS	22.6	22.5	36.5	69.3	66.0	75.2
KALMAN	30.8	30.2	46.7	61.3	57.6	69.9
MCSH	56.6	55.7	74.8	83.3	79.4	87.5
POLY	29.6	28.5	44.5	56.4	52.8	64.1
WHETSTONE	30.1	27.5	49.0	36.3	35.0	45.2
Mean	33.2	32.3	49.1	62.4	59.1	69.6
SUITE	43.9	43.2	60.5	73.7	70.1	79.5

Table 5.3: Line Hit Rates for 512 byte Instruction and Data caches with 4-way associativity, 32 byte lines.

accessed *and* hits/misses. Except in Section 5.4.3, the *History* simulations in this Chapter maintain no hit/miss information.

There are several reasons for this. Common replacement methods such as LRU and FIFO maintain information about lines, i.e. recency of line use and order of line entry into the set respectively. No information is maintained about hits/misses. Thus keeping no hit/miss information makes *History* in some sense more comparable to LRU and FIFO. Excluding hit/miss information also makes the length of *History* strings 30 bits instead of 480 bits. 30 bits is more comparable to the other methods which generally have string lengths from 32 to 48 bits.

Additional justification for excluding hit/miss information is given in Sections 5.4.2 and 5.4.3. Section 5.4.3 also investigates the effect of excluding the hit/miss information.

The genetic algorithm techniques employed here are quite simple, employing only the basic techniques outlined in Chapter 3. Other techniques are widely used. Population size can be allowed to vary between generations, overlap between generations can be allowed, crossover can be done between deterministically chosen pairs instead of randomly chosen pairs—the best performing pairs would generally be crossed in this case. Techniques can also be used to promote diversity, particularly early in the simulation when the danger of finding a poor local optima is highest. Simulation of dominant and recessive genes can also be used. More sophisticated forms of crossover such as PMX (Partially Matched Crossover) could also be employed. These additional techniques were not employed primarily for four reasons:

1. The duration of the simulations is already quite lengthy and the memory requirements quite large. This additional overhead would make the simulations prohibitively long and large.

2. The time required to write software and simulate all of these additional techniques was judged to be excessive for the scope of this work.

3. The fundamental operators provide a sufficient basis on which to test the concept of applying genetic algorithms to cache replacement policy.

4. Many of the enhancements seem unlikely to produce significantly better results. For example, consider PMX. In all discussion thus far, it has been assumed that the

position of a group of bits in a string determines the function of those bits. However, this need not be the case, and is not in nature. If a gene is moved to a different location on a chromosome it will continue to perform the same function. PMX and related techniques allow this capability to be added to genetic algorithms.

Bit location is important. Bits which are highly correlated in function may be widely separated in the string. In order to obtain the full benefit of crossover and its effect of finding good, small schemata, the string must use a representation in which related bits are grouped closely together.

PMX is essentially a more complicated form of crossover [12], that makes a bit's function independent of its position. It attempts to find natural groupings of bits in addition to finding good values for the bits. In this work, some effort was made to manually find representations in which correlated bits are grouped together. In most cases, natural representations appear to group bits reasonably well. For example, in the *LRU-Count* method, bits representing what to do when the least recently used line has been accessed 1 or 2 times are grouped adjacently. Given this natural grouping (and the added time and space requirements for adding position independence), it was decided not to implement position independence. However this might be an interesting area for further study.

Nonetheless some of the advanced techniques described above could prove useful in extending this work. For example, dominant and recessive genes are useful when the environment changes over time. Some criteria may be important at some time, then cease to be so, and later become important again. Recessive genes provide a natural way of storing useful information, even in generations where it is not needed [21].

For cache replacement algorithms, recession and dominance could be useful in the following procedure.

1. Simulate the first benchmark program for a "few" generations.
2. Take the population of strategies from the last generation of the previous benchmark and use them as the initial population in simulating the next benchmark for a "few" generations.
3. Repeat Step 2. until the last benchmark is reached, then go to Step 1.

Although this procedure is reasonable without employing recession and dominance, using them could improve performance. This is because certain criteria might be very important in some benchmarks, but not important in others. These criteria would have a good chance of being preserved in recessive genes, but might be lost otherwise. This might make an interesting area for further study.

5.2 Line Hit Rate

Much of the discussion here also applies to Section 5.3, *Opt Match Rate*. Here the goal is to maximize the *line hit rate*. (Recall from Chapter 2 that maximizing the *line hit rate* also maximizes the *overall hit rate*.) In Section 5.3, the goal is to maximize the fraction of misses in which the replacement algorithm replaces the same line OPT would have. In particular, much of the *Parameters* Section below applies equally to the Section on *OPT Match Rate*.

5.2.1 Parameters

As discussed in Chapter 3, there are four major genetic algorithm parameters in addition to the many cache parameters. To try all possible reasonable combinations of these would result in simulations taking billions of years, even for these six small benchmarks. Hence only a few variations are presented here, along with discussion on the effect of altering the values selected for the parameters.

The four basic genetic algorithm parameters are

- The probability of crossover
- The probability of mutation (or mutation rate)
- The population size
- The number of generations

In addition to these four parameters, there is an additional important "parameter" which is hard-coded into the simulator. This is the *objective function.* The objective function takes the metric used to measure the performance of a string, such as the *line hit rate*, and returns some (usually nonlinear) function of it. The value of the objective function probabilistically determines the relative number of the string that will be *reproduced* in the next generation.

Here the objective function used was $2^{\textit{line hit rate}}$. Hence if string, A has a 1% better *line hit rate* than string B, then approximately twice as many of A will usually be produced in the next generation as are produced of B. Since the simulation uses a fixed finite population, strings with low *line hit rates* will likely not be reproduced at all in the next generation.

Less steep objective functions, such as $\textit{line hit rate}^{10}$, were tried, but reproduced too many poor strings, resulting in slow improvement from generation to generation. Steeper objective functions have the opposite problem and tend to lose needed population diversity. The result is that they often converge at relatively poor local optima.

Returning to the four "major" parameters, the *probability of crossover* was always 0.6 in these simulations. This value has been found to be suitable for a wide range of applications by other researchers [14].

A very important parameter in obtaining good results in this study is the *mutation rate.* Recall from Chapter 3 that the mutation rate is the probability that a single bit will change its value. Two values of mutation rate were used in this study, 1% and 10%. Figures 5.1 and 5.2 compare the performance of *LRU-FIFO* using a 1% mutation rate and a 10% mutation rate. As can be seen, a mutation rate of 10% generally provides superior results to a mutation rate of 1%.

With a 1% mutation rate, the simulations tend to get stuck at relatively poor local optima. The increased mutation rate allows the simulation to more readily jump out of these local optima when they occur. (This is somewhat similar to raising the temperature in simulated annealing.) Because of its universally superior performance, the 10% mutation rate is used in all the findings reported below.

The *population size* was problematic. A population size of 100 was chosen for all simulations. This number is a compromise between the conflicting goals of a diverse population and the need to minimize simulation time. It is also comparable to what other researchers

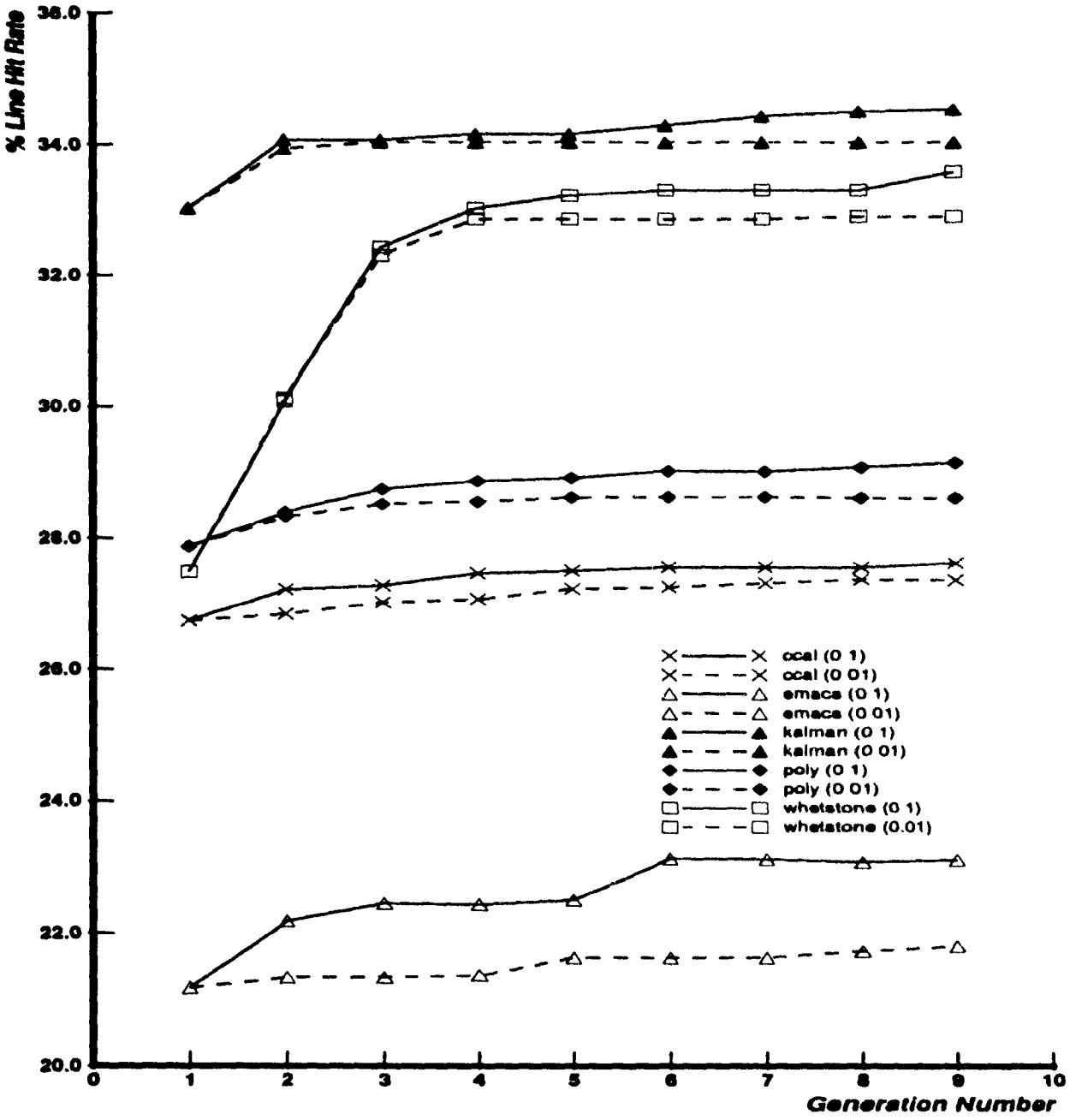

Figure 5.1: Comparison of Performance of Best String by Generation for a Mutation Rate of 0.1 (solid lines) and 0.01 (dashed lines) for a 512 byte I-cache, 4-way associativity, 16 byte lines.

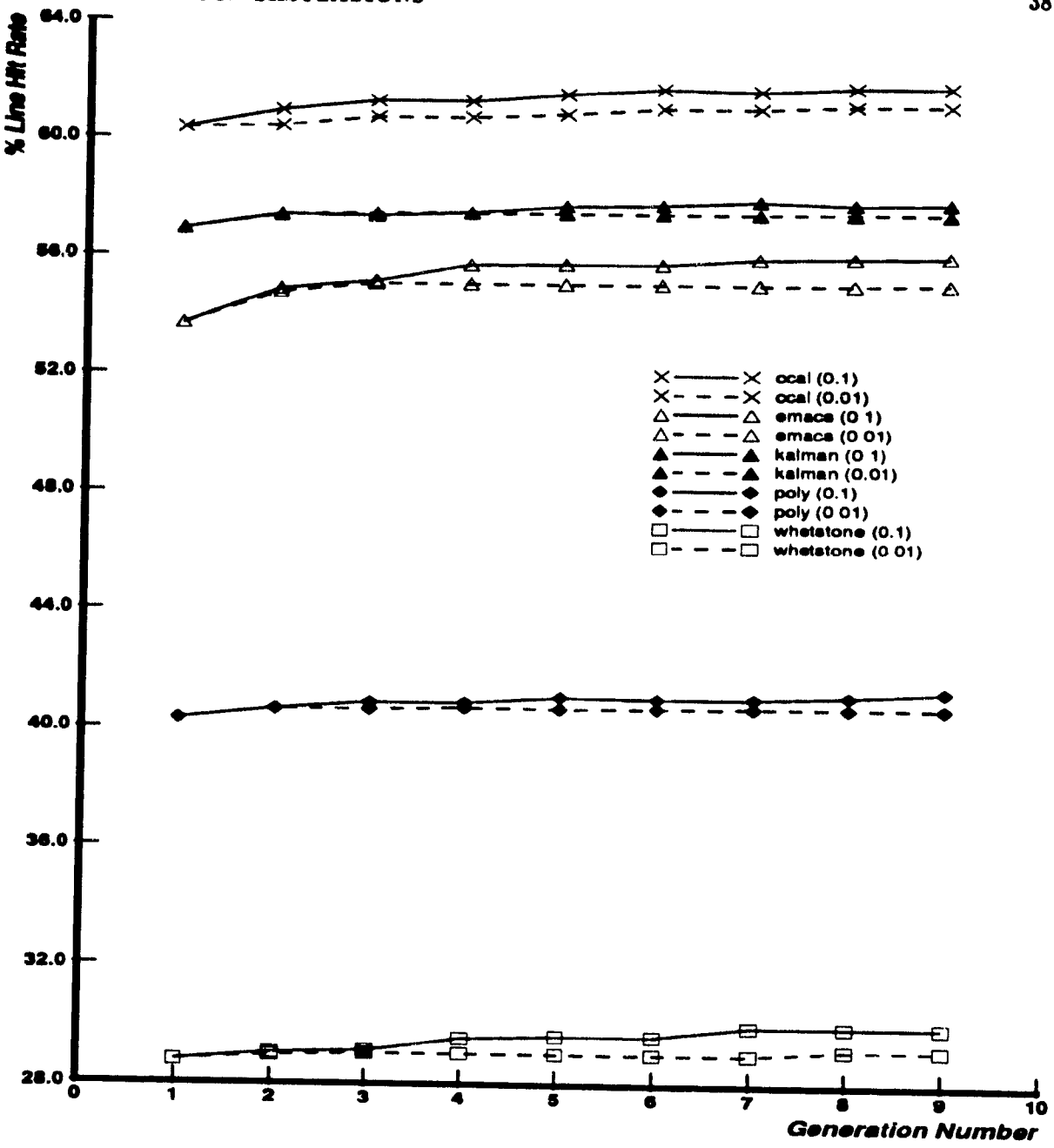

Figure 5.2: Comparison of Performance of Best String by Generation for a Mutation Rate of 0.1 (solid lines) and 0.01 (dashed lines) for a 512 byte D-cache, 4-way associativity, 16 byte lines.

have chosen [9] [19] [14] [20]. Section5.2.4 gives additional empirical evidence that 100 is a reasonable choice for population size.

Another factor to consider when choosing the population size is the length of strings in the population. 100 appears adequate even for large string sizes. For example, if a 2-bit count is used in *LRU-Count* or *FIFO-Count*, each string in the population has 512 bits. Using a 1-bit count requires strings of only 32 bits. Likewise if an 8-bit miss history is used in *LRU-History* or *FIFO-History* each string in the population has 512 bits, while a 4-bit history requires only 32-bit strings. As can be seen in Figures 5.3 and 5.4, the improvement in the longer strings is far more dramatic. (Results corresponding to longer strings all have solid lines, while those corresponding to shorter strings all have dashed lines.)

It is also interesting to note that in the early generations, the performance of the longer strings is generally worse than that of the shorter ones. This is because most of the additional possible actions of long strings are bad. Later, however, the performance of the longer strings overtakes that of the shorter ones. This is reassuring. Since the longer strings have all the information available in the shorter ones, plus additional information, they should always be able to do at least as well. As can be seen in Figures 5.3 and 5.4, by the end of 9 generations, the performance of the longer strings is better than that of the corresponding shorter string in all but one case, *LRU-Count* and the *poly* benchmark.

Even in this case, if the number of generations is increased to 11, the performance is almost identical to that of the shorter string (30.92% line hit rate for the shorter strings versus 30.88% for the longer strings). Furthermore, in most cases the performance of the longer strings is still improving after 9 generations while that of the shorter strings is flat.

Despite the generally better performance when using a deeper history and larger counts, the simulations reported below all use the shorter versions, i.e. 1-bit counts and 4-deep miss histories when in combination with LRU or FIFO. The stand-alone history also uses a 4-deep record of hits/misses and lines accessed. Shorter versions require less hardware to implement and would likely be slightly faster as well. Furthermore their simplicity does not sacrifice much overall performance. For the *kalman* benchmark, the additional data provides a mean improvement of only 0.08% in overall hit rate from 87.70% to 87.78%, while for the *poly* benchmark the difference is only 0.03% from 90.93% to 90.96%.

The *number of generations* can be determined by viewing the rate of improvement

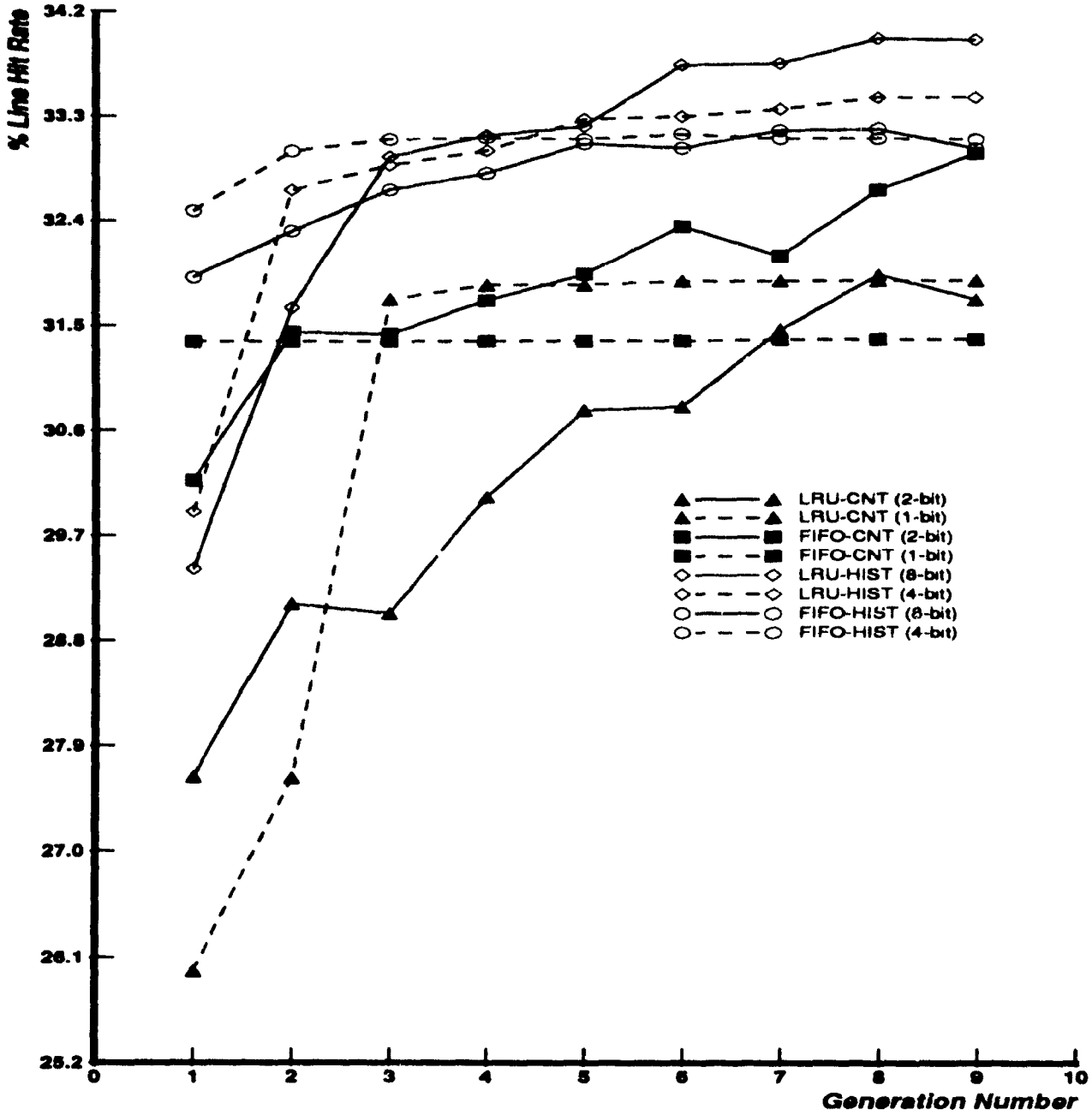

Figure 5.3: Improvement in Best Strategy for *kalman* by Generation. Solid lines represent longer *history* or *count* record, dashed lines represent shorter. Results for 512 byte I-cache, 4-way associativity, 32 byte lines.

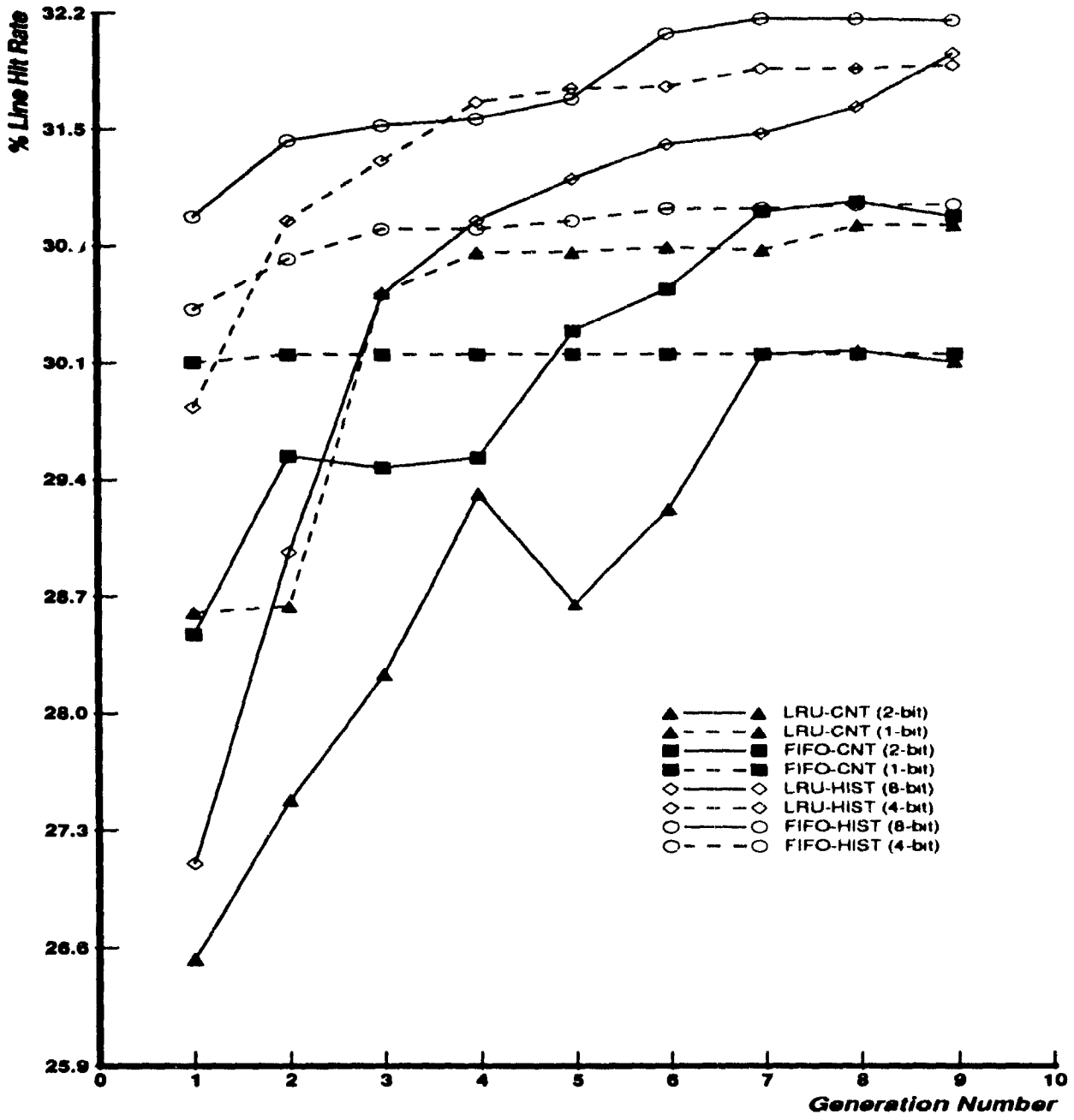

Figure 5.4: Improvement in Best Strategy for *poly* by Generation. Solid lines represent longer ***history*** or ***count*** record, dashed lines represent shorter. Results for 512 byte I-cache, 4-way associativity, 32 byte lines.

between generations. When little or no improvement occurs, the simulation can be stopped. The improvement over a number of generations is illustrated in Figures 5.5 to 5.10. These depict the improvement over 9 generations for an instruction and data cache for three of the benchmarks: *ccal*, *emacs*, and *whetstone*. The results from the other benchmarks are similar.

Note that occasionally performance is flat for many generations, then suddenly jumps. The improvement of *FIFO-Count* is relatively flat up to the 5th generation in Figures 5.6 and 5.7, but makes a sudden improvement in the 6th generation. In most of the other cases the performance improvement is relatively flat within a small number of generations.

Despite the fact that the overall performance of these genetic algorithm techniques was worse with data caches than with instruction caches, the data caches generally exhibited marginally more improvement from initial to final generation than did the instruction caches. This can be seen graphically by comparing Figures 5.5– 5.7 to Figures 5.8– 5.10.

5.2.2 Results for Individual Benchmarks

Overall the improvement of these new approaches over traditional methods was moderate. Figures 5.11 to 5.14 (page 49) show the *line hit rates* obtained for instruction and data caches for 16 and 32-byte lines. Several points can be inferred from these Figures.

The most obvious result is that the genetic algorithm approaches provide substantially better improvement over LRU and FIFO on instruction caches than on data caches. Possible reasons for this are discussed below.

It is also clear that the performance improvement of the genetic algorithm approaches is generally about the same for both 16 and 32 byte lines. One useful way to measure the performance of the genetic algorithm approach is to measure how much of the possible gain from LRU to OPT is attained. Call this the *LRU-OPT gain*.

For example if LRU has a *line hit rate* of 30% and OPT has a line hit rate of 50%, and a genetic algorithm approach achieves a *line hit rate* of 35%, then the *LRU-OPT gain* is $\frac{35-30}{50-30} = 25\%$. *LRU-OPT gain* can be applied to other measures as well, for example *overall hit rate*. However, unless otherwise indicated it refers to *line hit rate*. As a practical matter

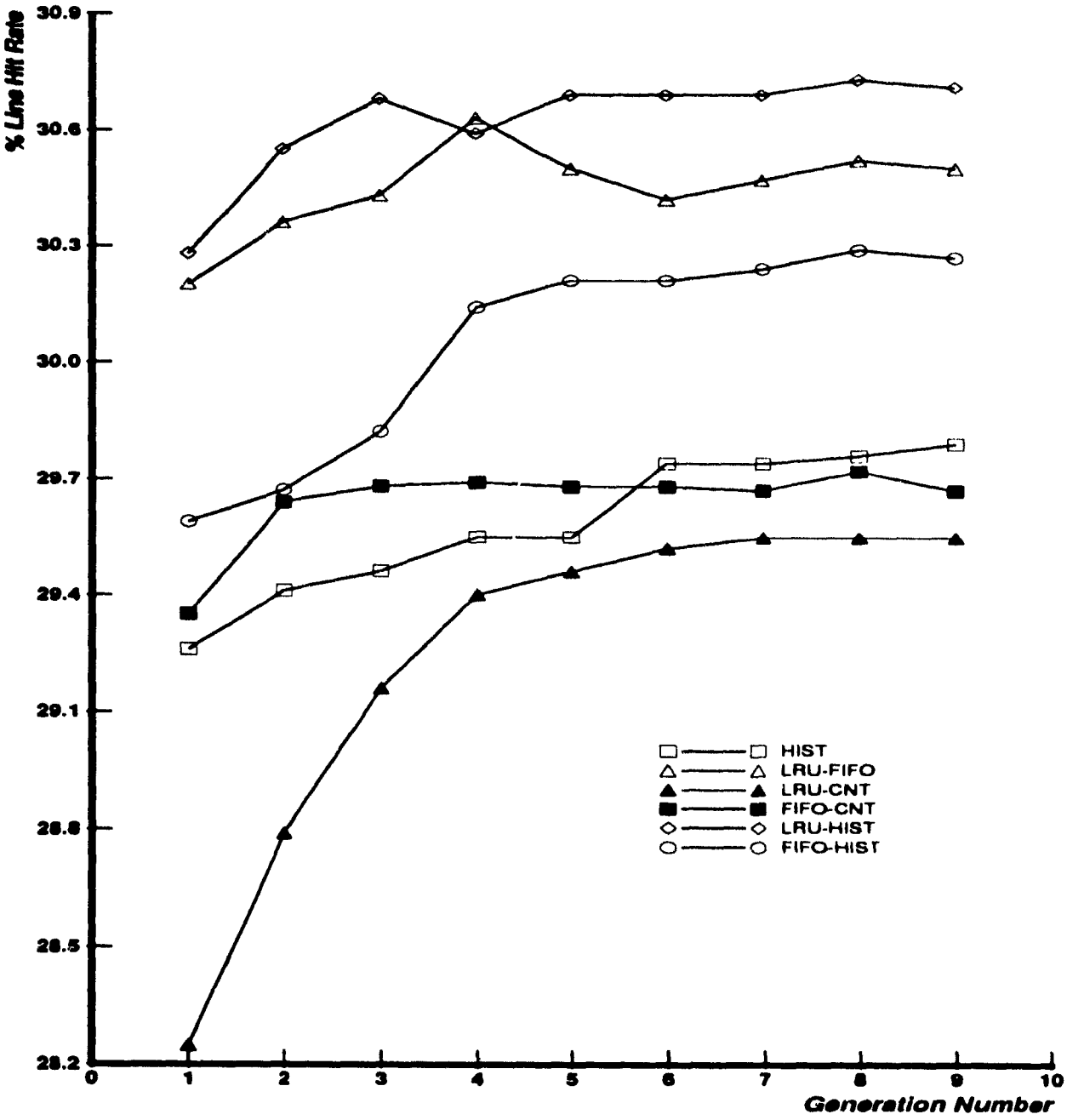

Figure 5.5: Improvement in Best Strategy for *ccal* by Generation for 512 byte I-cache, 4-way associativity, 32 byte lines.

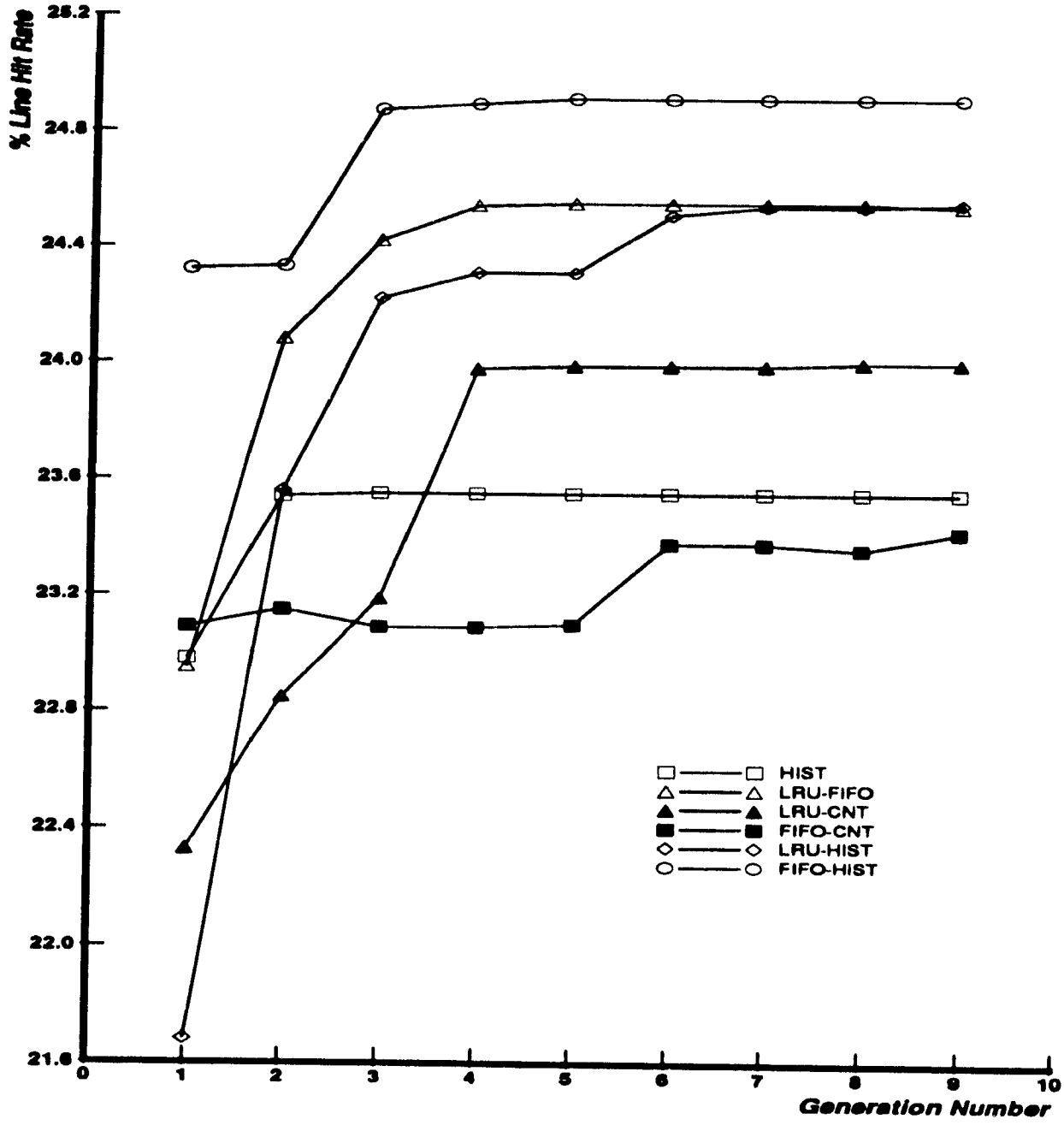

Figure 5.6: Improvement in Best Strategy for *emacs* by Generation for 512 byte I-cache, 4-way associativity, 32 byte lines.

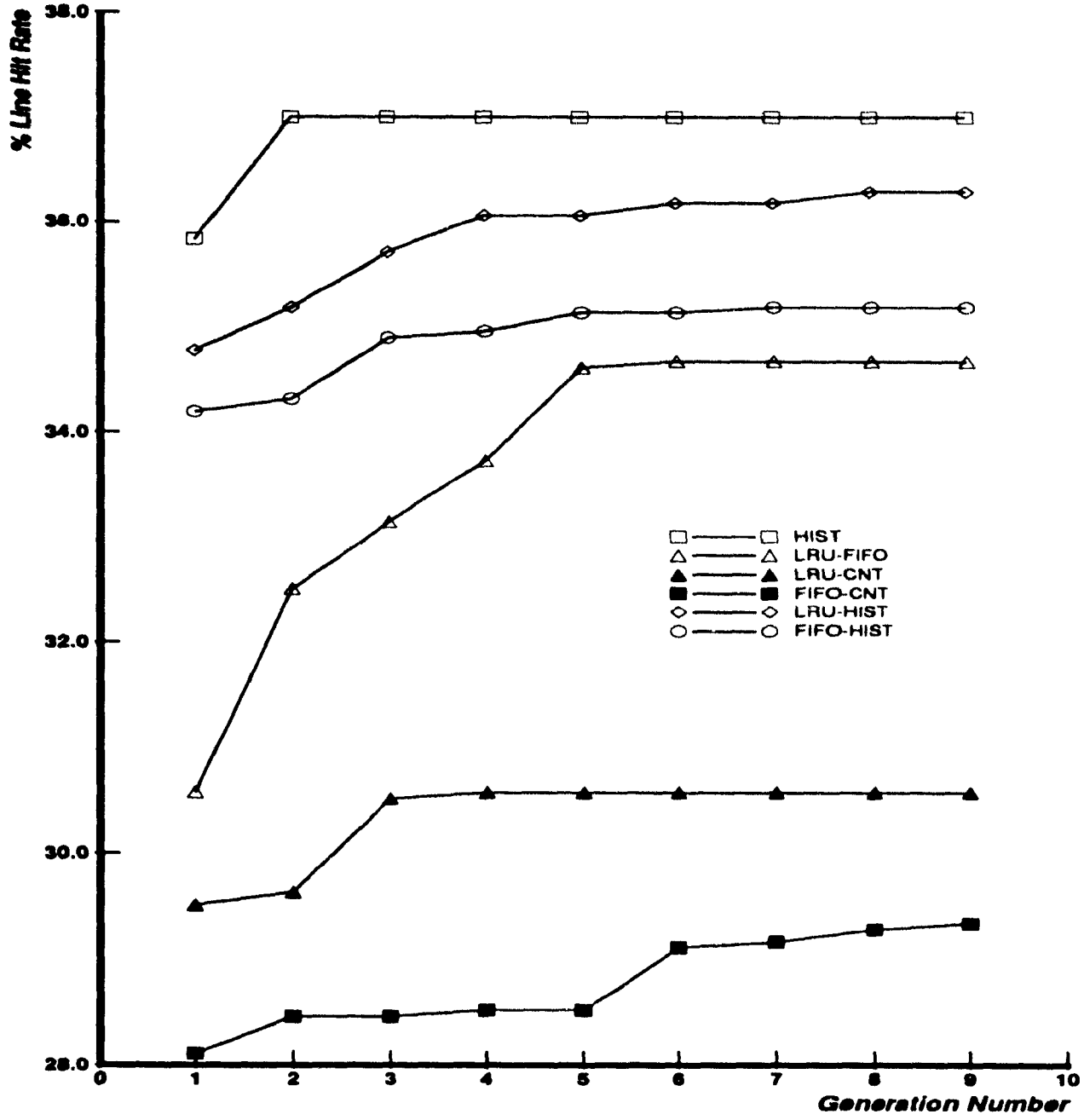

Figure 5.7: Improvement in Best Strategy for *whetstone* by Generation for 512 byte I-cache, 4-way associativity, 32 byte lines.

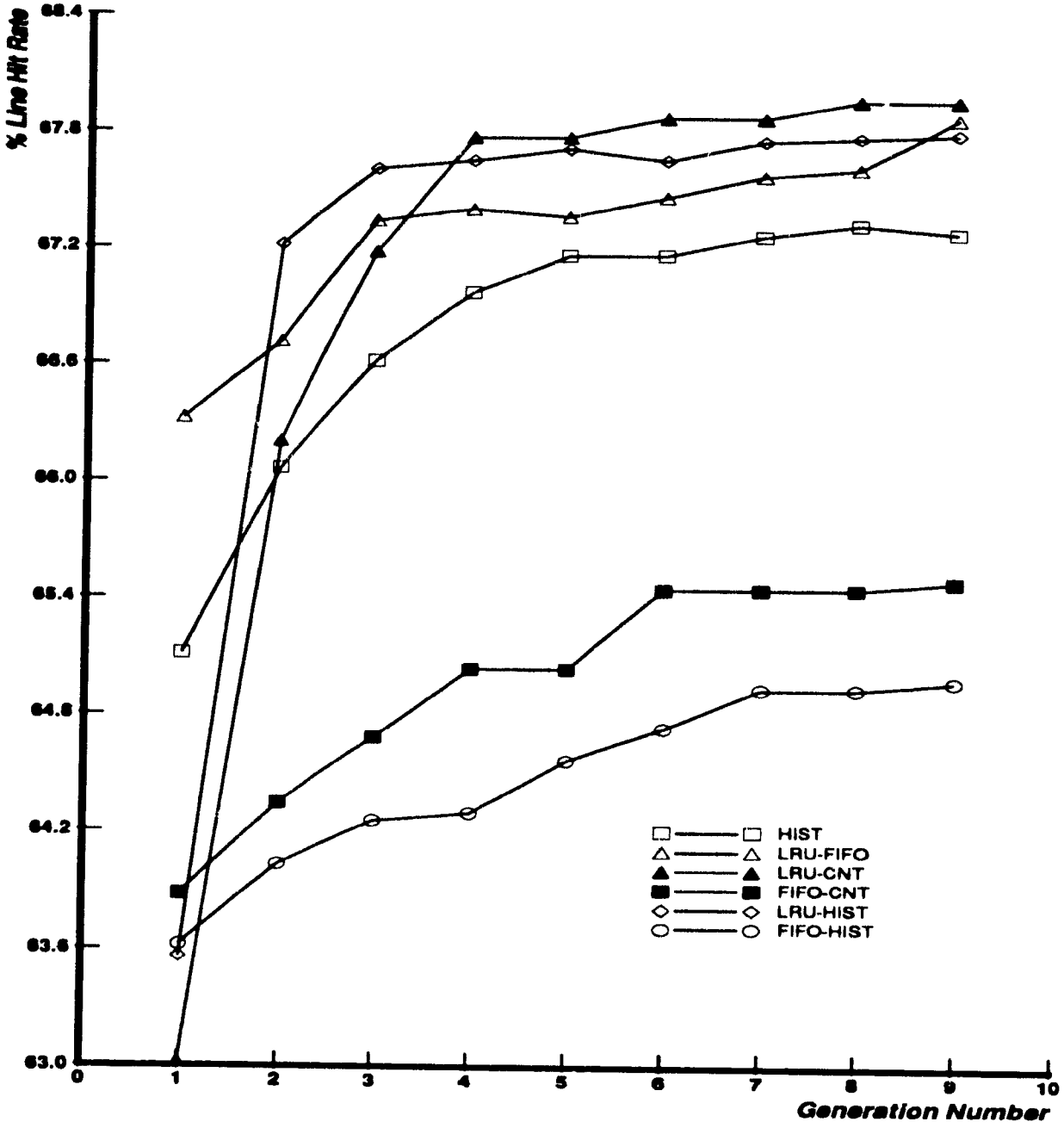

Figure 5.8: Improvement in Best Strategy for *ccal* by Generation for 512 byte D-cache, 4-way associativity, 32 byte lines.

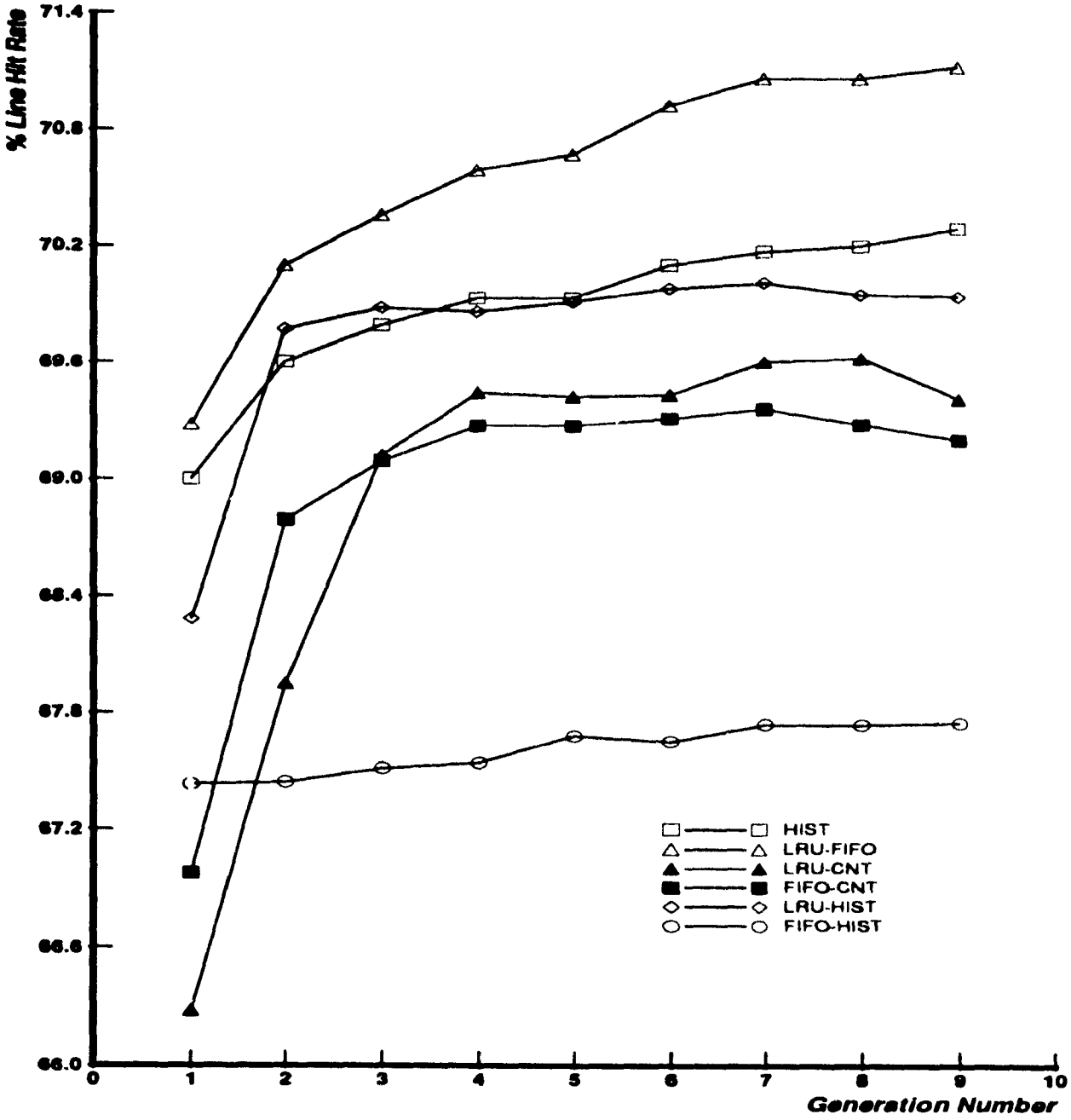

Figure 5.9: Improvement in Best Strategy for *emacs* by Generation for 512 byte D-cache, 4-way associativity, 32 byte lines.

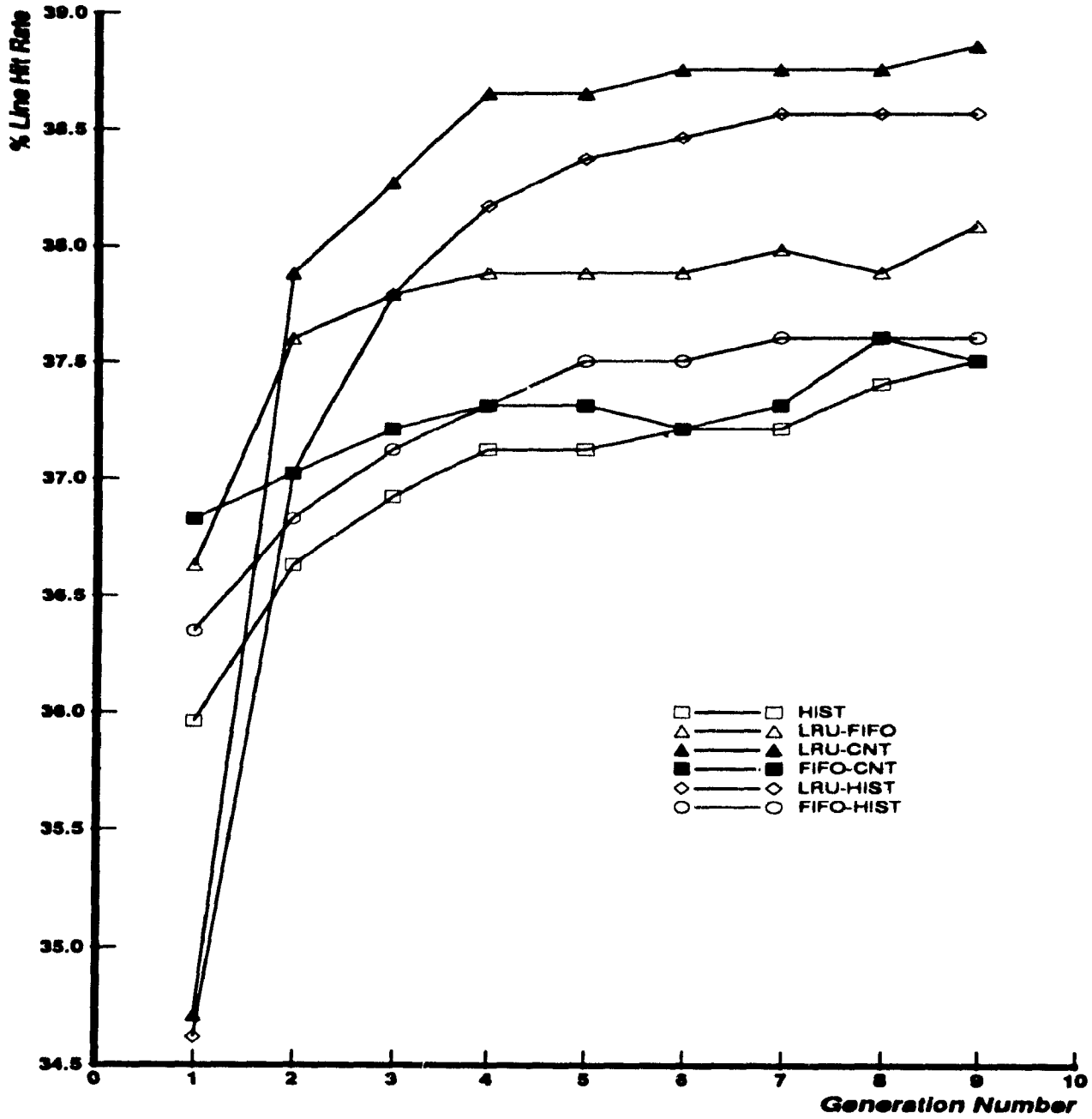

Figure 5.10: Improvement in Best Strategy for *whetstone* by Generation for 512 byte D-cache, 4-way associativity, 32 byte lines.

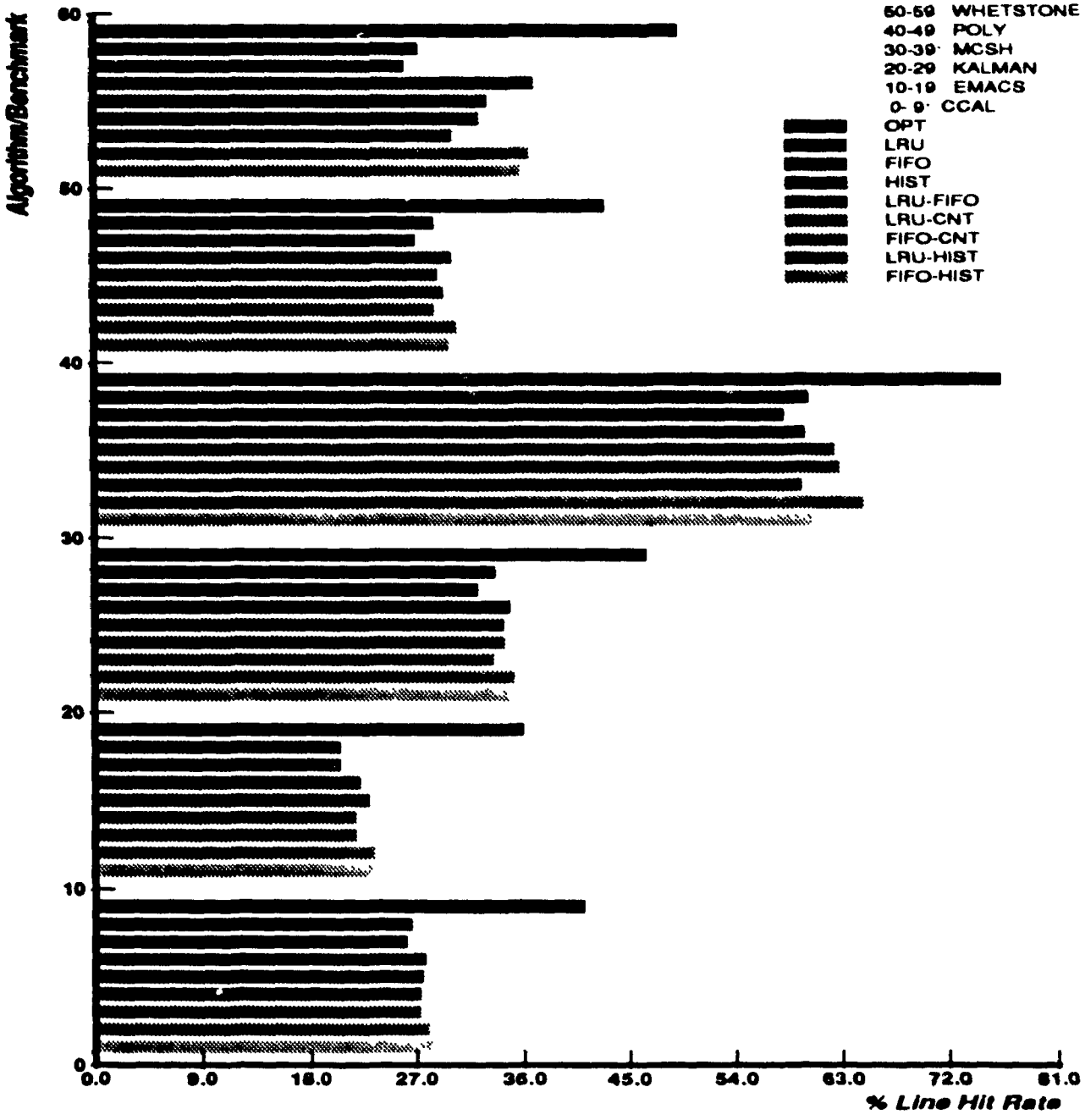

Figure 5.11: *Line Hit Rates* for 512 byte I-cache, 4-way associativity, 16 byte lines.

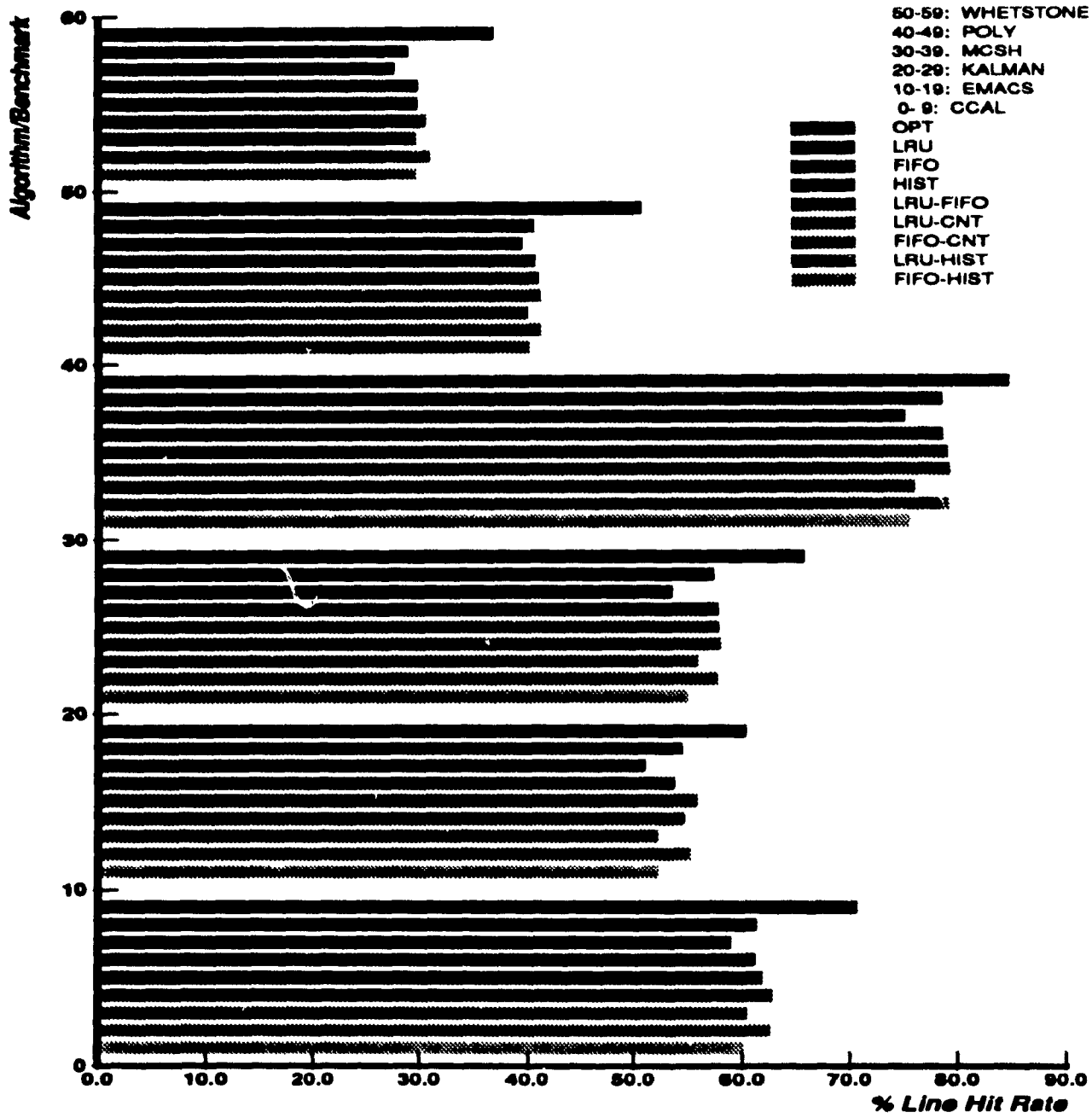

Figure 5.12: *Line Hit Rates* for 512 byte D-cache, 4-way associativity, 16 byte lines.

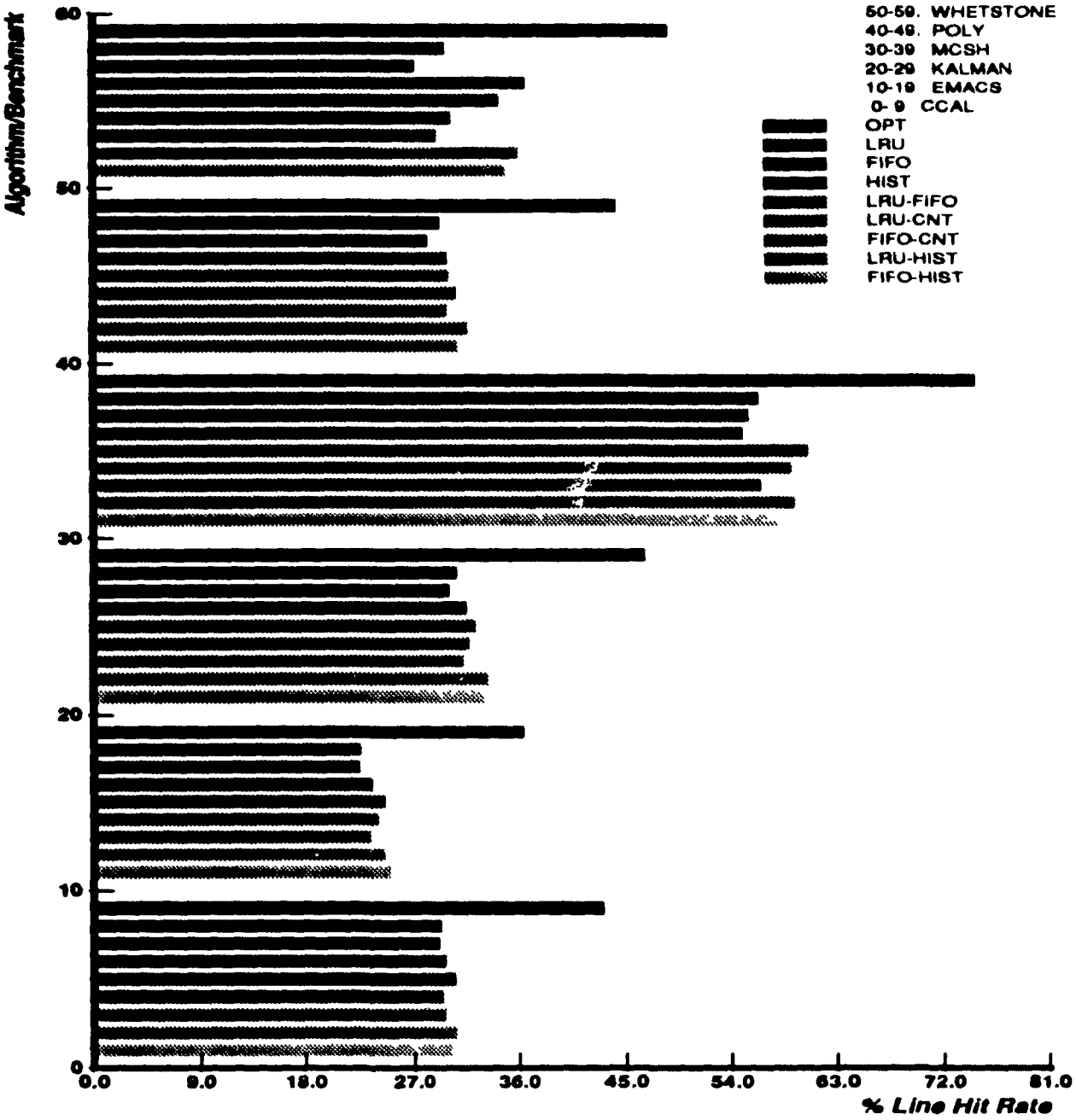

Figure 5.13: *Line Hit Rates* for 512 byte I-cache, 4-way associativity, 32 byte lines.

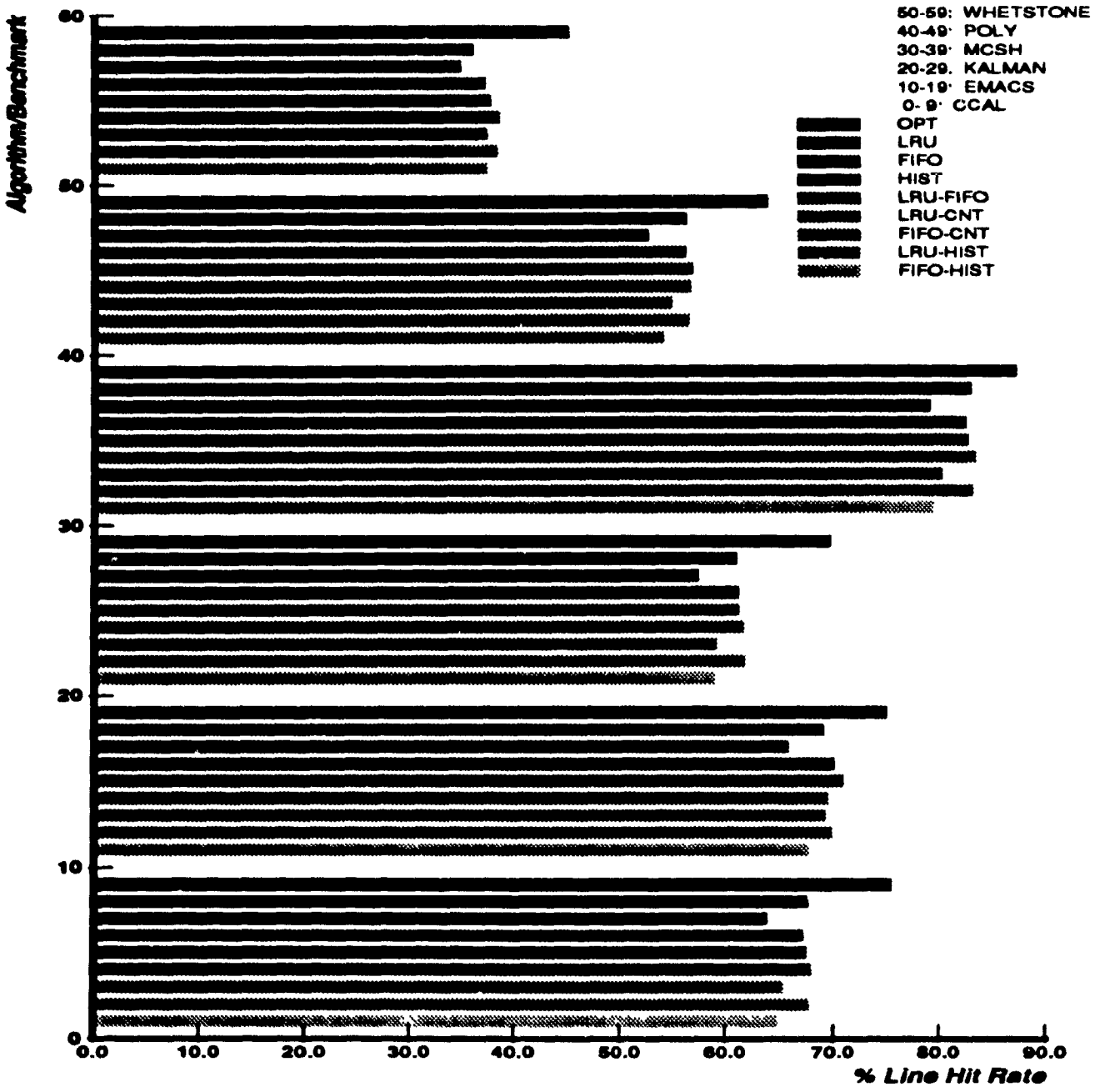

Figure 5.14: *Line Hit Rates* for 512 byte D-cache, 4-way associativity, 32 byte lines.

the *LRU-OPT gain* is generally about the same for the *line hit rate* and the *overall hit rate*.

In the discussion below it is important to keep in mind that the *absolute* difference in performance between LRU and OPT is generally quite small. The mean difference in overall hit rate is 2.2% for instruction caches and 2.3% for data caches. Thus a (hypothetical) 15% *LRU-OPT gain* in *overall hit rate* would correspond to only a 0.3% absolute improvement in hit rate.

While 0.3% is small, it represents 15% of what is possible. It is also larger than the mean gap of 0.2% between LRU and FIFO for instruction caches in these benchmarks. Furthermore, larger, more complex benchmarks may have wider absolute gaps between the hit rates of LRU and OPT. For example if the hit rates were 75% for LRU and 85% for OPT, an *LRU-OPT gain* of 15% would correspond to a 1.5% absolute improvement. And as reported below, the *LRU-OPT gain* is in some instances over 40%. Such a gain would yield a 4% absolute improvement in this hypothetical case.

Overall the best performing genetic algorithm approach, *LRU-History*, has a mean *LRU-OPT gain* of 18% and a mean absolute improvement in overall miss rate of 0.4% for an instruction cache with 32 byte lines. Averaging over all the approaches, instruction caches with 16 byte lines attain a mean *LRU-OPT gain* of 12% while those with 32 byte lines have a mean *LRU-OPT gain* of 10%. For data caches with 16 byte lines, mean performance actually drops—the *LRU-OPT gain* is -1% for 16 byte lines and -4% for 32 byte lines. For more detailed information on the performance of each approach on each algorithm see Tables 5.4 to 5.7.

The best *LRU-OPT gain* for instruction caches was 44%. This result occurred using the *whetstone* benchmark and the *history* method and a cache with 16-byte lines. For data caches, the best *LRU-OPT gain* was 30%. This was obtained using the *emacs* benchmark and the *LRU-FIFO* method and a cache with 32 byte lines.

The 44% *LRU-OPT gain* on *whetstone* provides a 1.7% increase in the overall hit rate from 87.1% to 88.8%. The 30% increase on *emacs* corresponds to 0.3% increase in overall hit rate from 94.3% to 94.6%. A greater increase in overall hit rate in a data cache is actually attained by *whetstone* using *LRU-Count*. In a cache with 32-byte lines the overall hit rate is increased by 0.7% from 82.6% to 83.3%.

In viewing Tables 5.4 to 5.7, there is no clear ranking among the genetic algorithm

	LRU-FIFO	LRU-Count	LRU-Hist	Hist	FIFO-Count	FIFO-Hist	MEAN
CCAL	7%	5%	10%	8%	5%	13%	8%
EMACS	16%	9%	19%	11%	9%	17%	13%
KALMAN	5%	6%	12%	9%	-1%	9%	7%
MCSH	14%	16%	29%	-2%	-4%	2%	9%
POLY	2%	6%	13%	11%	0%	9%	7%
WHETSTONE	26%	23%	42%	44%	13%	39%	31%
Mean	12%	11%	21%	13%	4%	15%	12%

Table 5.4: *LRU-OPT Gains* for 512 byte I-cache, 4-way associativity, 16 byte lines.

	LRU-FIFO	LRU-Count	LRU-Hist	Hist	FIFO-Count	FIFO-Hist	MEAN
CCAL	5%	15%	13%	-1%	-9%	-12%	2%
EMACS	23%	4%	11%	-13%	-41%	-40%	-9%
KALMAN	5%	7%	4%	5%	-17%	-29%	-4%
MCSH	7%	11%	10%	1%	-42%	-50%	-11%
POLY	4%	6%	6%	1%	-6%	-5%	1%
WHETSTONE	11%	22%	26%	12%	9%	10%	15%
Mean	9%	11%	12%	1%	-18%	-21%	-1%

Table 5.5: *LRU-OPT Gains* for 512 byte D-cache, 4-way associativity, 16 byte lines.

	LRU-FIFO	LRU-Count	LRU-Hist	Hist	FIFO-Count	FIFO-Hist	MEAN
CCAL	9%	1%	10%	3%	2%	6%	5%
EMACS	14%	10%	14%	7%	6%	17%	11%
KALMAN	10%	7%	17%	5%	4%	15%	9%
MCSH	23%	16%	17%	-8%	1%	8%	10%
POLY	5%	9%	15%	4%	4%	10%	8%
WHETSTONE	24%	2%	33%	37%	-4%	27%	20%
Mean	14%	7%	18%	8%	2%	14%	10%

Table 5.6: *LRU-OPT Gains* for 512 byte I-cache, 4-way associativity, 32 byte lines.

	LRU-FIFO	LRU-Count	LRU-Hist	Hist	FIFO-Count	FIFO-Hist	MEAN
CCAL	-1%	3%	1%	-5%	-29%	-35%	-11%
EMACS	30%	5%	12%	15%	1%	-26%	6%
KALMAN	2%	7%	9%	3%	-23%	-25%	-5%
MCSH	-8%	10%	5%	-12%	-66%	-87%	-27%
POLY	8%	6%	3%	-1%	-18%	-27%	-5%
WHETSTONE	19%	28%	25%	12%	15%	15%	19%
Mean	8%	10%	9%	2%	-20%	-31%	-4%

Table 5.7: *LRU-CPT Gains* for 512 byte D-cache, 4-way associativity, 32 byte lines.

approaches other than *LRU-History*. The *FIFO-Count* method is the most consistently bad, while *FIFO-History* does quite well for instruction caches, but quite poorly for data caches. The other three methods, *LRU-FIFO*, *LRU-Count*, and *History* do moderately well on everything.

A variety of other observations can made from Tables 5.4 to 5.7 and Figures 5.11 to 5.14.

Combining LRU and FIFO information in the *LRU-FIFO* policy almost always yields an improvement over both LRU and FIFO performance. For an instruction cache with 32 byte lines, *LRU-FIFO* has a mean *line hit rate* of 35.6% versus 33.2% for LRU and 32.3% for FIFO. For data caches the gap is narrower for LRU and wider for FIFO. With 32 byte lines *LRU-FIFO* has a mean *line hit rate* of 63.2% versus 62.4% for LRU and 59.1% for FIFO.

Comparing the performance of *History* in Tables 5.4 and 5.6 shows that *History* is an exception to the general rule that behavior with 16 and 32 byte lines is quite similar. For an instruction cache the *LRU-OPT gain* falls from 13% with 16 byte lines to 8% with 32 byte lines.

This is to be expected: since longer lines exploit spatial locality, the line reference history is more likely to contain only a small number of *distinct* lines when the line size is longer. And if only 1 or 2 lines are accessed in the history records, it is difficult to choose which line to replace. Surprisingly, data cache behavior defies this logic and shows an increase in *LRU-OPT gain* from 1% to 2%.

To alleviate the problem of *History* faring more poorly with longer line sizes, alternatives could be used, such as recording every other line accessed, or only recording instances when the line referenced is different than the previous line referenced in the set. However, these measures add complexity. Furthermore they grow quite similar to LRU: LRU records exactly the most recent reference to any given line.

For instruction caches, combining hit-miss history information for the set with LRU or FIFO information (*LRU-History* and *FIFO-History*) generally does better than combining the counts of hits to each line with LRU or FIFO information (*LRU-Count* and *FIFO-Count*). Maintaining a hit-miss history gives an indication that the working set is

changing—if several misses occur in succession, it is likely that the program is moving to a new phase of work. The hit-miss history provides this information, and may allow the cache to quickly rid itself of lines that it would otherwise keep.

For data caches, *LRU-History* and *LRU-Count* fare approximately equally when combined with LRU. However, *FIFO-Count* does somewhat better than *FIFO-History*. Perhaps this is because LRU has (recency of) usage information, while FIFO provides no usage information. Hence augmenting LRU with a usage count adds little useful information, while augmenting FIFO provides usage information needed for improvement.

Furthermore the improvement should be particularly noticeable in a data cache where it is important to be able to distinguish "scratch" variables which are used only once. This view is supported by the fraction of misses when it is best *not* to store the newly referenced line in cache. This fraction is much higher for data caches than for instruction caches, as is discussed in Section 5.3.

As can be seen in Tables 5.4 and 5.6, FIFO-History always outperforms simple LRU for instruction caches. In fact on average FIFO-History obtains an *LRU-OPT gain* of 15%. In most cases FIFO-Count also outperforms simple LRU for instruction caches, although marginally, obtaining an *LRU-OPT gain* of 2% to 4%.

However for data caches, both FIFO-History and FIFO-Count do significantly worse than simple LRU. This is likely caused by three factors.

1. The *line hit rate* in data caches is substantially higher than in instruction caches—62.4% on average versus 33.2% for LRU and 32 byte lines Hence existing algorithms already perform well in data caches.

   (Interestingly instruction caches do slightly better in *overall hit rate*—88.6% on average versus 91.0% for the same LRU, 32 byte line case. To achieve the better overall performance, the instruction caches are clearly exploiting the greater temporal locality present in the instruction stream.)

2. The next reason is similar. The ratio of the LRU *line hit rate* to OPT's *line hit rate* is higher in data caches than instruction caches. Again taking the example of a cache having 32 byte lines, LRU's *line hit rate* is 88.8% of OPT on average for a data cache, while the ratio is 66.6% for an instruction cache.

3. The last reason is that plain FIFO does better compared to plain LRU in instruction caches than in data caches. On average FIFO's *line hit rate* is 94.8% of LRU's for a data cache with 32 byte lines, while for instruction caches the ratio is 97.2%. Hence hybrid FIFO techniques start at a significant disadvantage with respect to LRU. However in all cases, the two hybrid FIFO techniques perform better than plain FIFO.

5.2.3 Results for a Multitasking Suite of Benchmarks

The basic parameters used here are the same as those used for individual benchmarks in Section 5.2.2. In particular the mutation rate is 10%, counts are 1-bit, and histories are 4-deep. There are three additional parameters that are required for simulating multiple traces:

1. The task switch interval is 20,000 addresses. This means that 20,000 addresses from the first benchmark are processed, then 20,000 from the second, etc. until the sixth and last benchmark, after which the first benchmark is restarted at the point where it was left. This simulates a multitasking operating system.

   There is actually one additional nuance to this scheme. In order to better simulate equal time slices in a multitasking operating system, a method originally suggested in [38] was used. Whenever a miss occurs, the assumed miss latency is subtracted from the 20,000 addresses to be processed in that interval. Here the miss latency is assumed to be 6 cycles. Hence if a benchmark had 25 misses during its interval, only $20000 - 25 \times 6 = 19850$ addresses would actually be processed.

2. A physical address cache is modelled, and the cache is not flushed on task switches. With a physical address cache it is possible to have different benchmarks use the same physical memory. In such a case, any cache entries corresponding to the overlapping memory must be invalidated. However, the total resident size (under 1 megabyte) of the suite of benchmarks used here is sufficiently small as to all fit in the physical memory of most modern machines. Hence the effect of overlapping physical addresses is ignored.

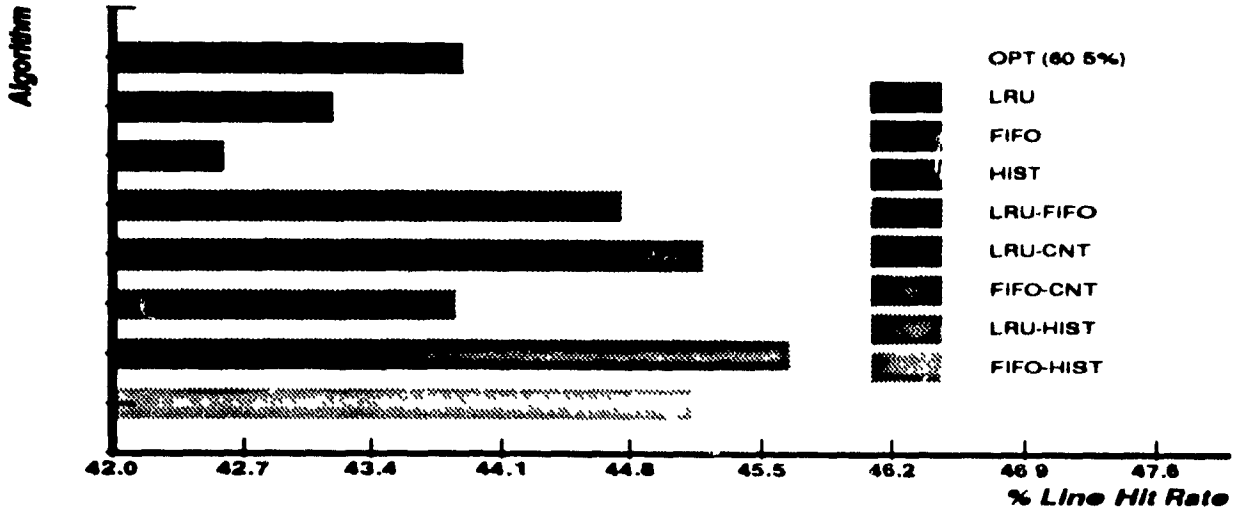

Figure 5.15: Results by Algorithm for a Multitasking Suite of All Benchmarks for 512 byte I-cache, 4-way associativity, 32 byte lines.

3. The order of the benchmarks also has some effect on the resulting hit rate. The order used here is *emacs*, *ccal*, *mcsh*, *kalman*, *poly*, *whetstone*. This choice is largely arbitrary.

Several of the genetic algorithm approaches attain a significantly higher *line hit rate* than achieved by LRU. Results for an instruction cache with 32 byte lines are shown in Figure 5.15, while results for a data cache with 32 byte lines are in Figure 5.16.

As was the case with the individual benchmarks, *LRU-History* is again the best overall genetic algorithm approach. For an instruction cache with 32 byte lines, it achieves a *line hit rate* of 45.7% for the overall suite of benchmarks. This compares to only 43.9% for simple LRU. The *LRU-OPT gain* is 11%.

For a data cache, the numbers are 73.9% *line hit rate* for *LRU-History* versus 73.7% for simple LRU. The *LRU-OPT gain* is 3%, Actually *LRU-Count* does as well for data caches, also achieving a 73.9% *line hit rate*. As with the individual benchmarks, improvement is better in instruction caches.

The *overall hit rates* are, of course, somewhat closer: Instruction caches yield a 92.2%

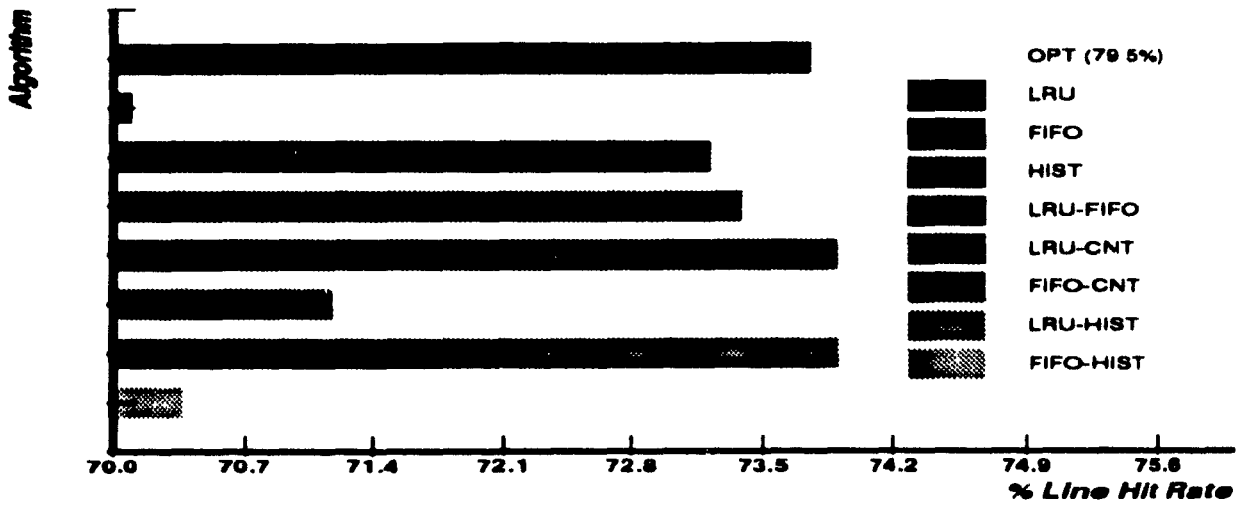

Figure 5.16: Results by Algorithm for a Multitasking Suite of All Benchmarks for 512 byte D-cache, 4-way associativity, 32 byte lines.

hit rate for LRU and 92.4% for *LRU-History* giving an *LRU-OPT gain* of 9%. Data caches yield 92.11% for LRU, and 92.14% for *LRU-History*, giving an *LRU-OPT gain* of 2%.

In addition to their performance for the entire suite of benchmarks, it is important to know how well replacement strings developed using the *suite* of benchmarks do on the *individual* benchmarks. It is possible that *LRU-History* might do much better than simple LRU for one benchmark, but significantly worse for the others. If the performance of *LRU-History* on the one benchmark were sufficiently high, the poor performance on the other benchmarks would be masked.

Luckily this is not the case. The performance of the individual benchmarks with the best *LRU-History* algorithm found are shown in Figures 5.17 and 5.18, instruction and data cache results respectively. Actually separate algorithms were used for instruction and data caches. As can be seen in Figure 5.17, the performance of *LRU-History* is superior to simple LRU on each individual benchmark for an instruction cache. For a data cache, *LRU-History* is better in 4 of the 6 benchmarks, and slightly worse in the other two.

Of course, the *LRU-History* algorithm was found by applying a genetic algorithm to

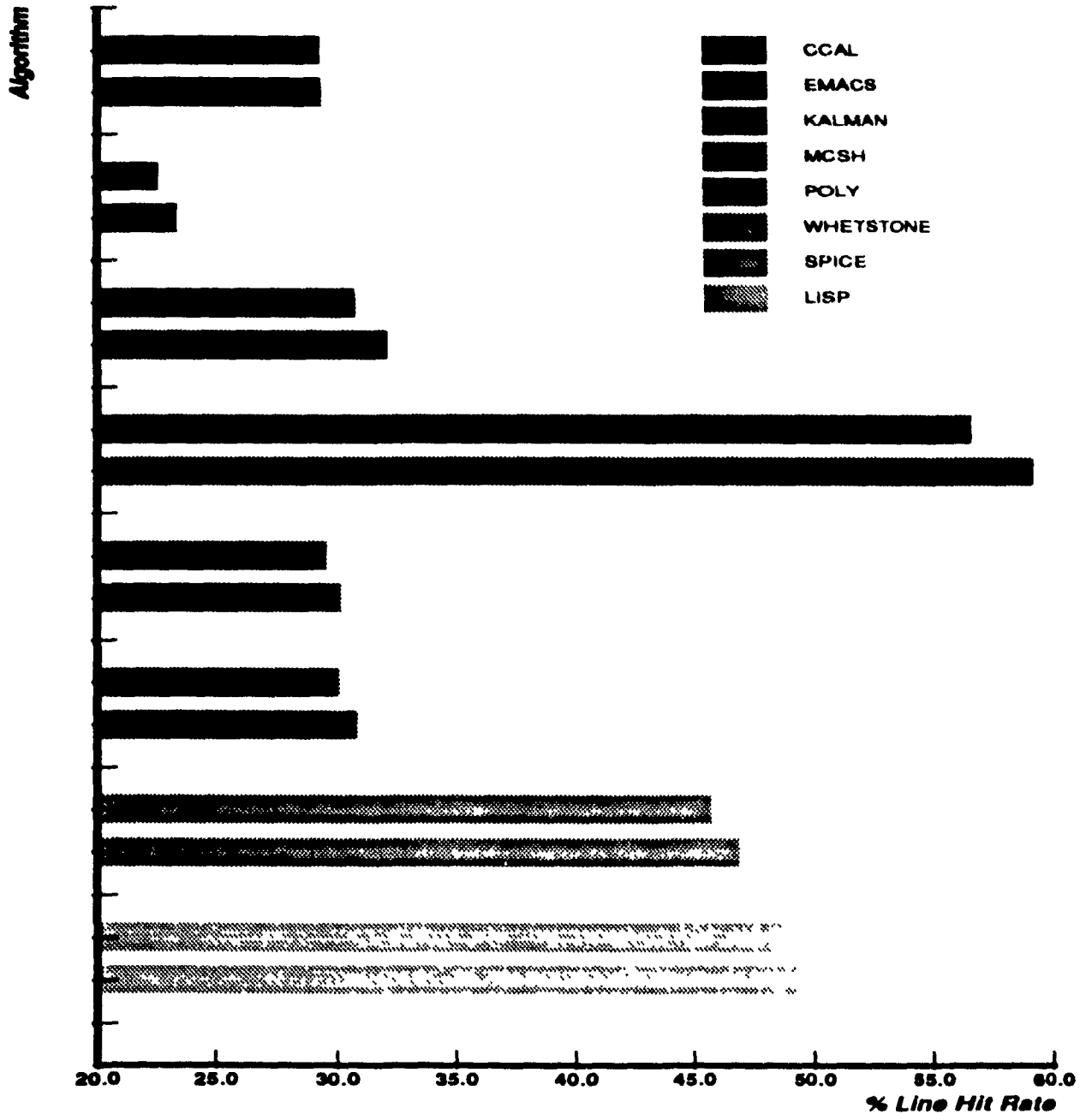

Figure 5.17: Results of using the overall best *LRU-History* algorithm on individual benchmarks. Upper Bar is LRU, Lower is *LRU-History*. 512 byte I-cache, 4-way associativity, 32 byte lines.

Figure 5.18: Results of using the overall best *LRU-History* algorithm on individual benchmarks. Upper Bar is LRU, Lower is *LRU-History*. 512 byte D-cache, 4-way associativity, 32 byte lines.

these 6 benchmarks. Hence it might be argued that this algorithm should do well on *these* 6 benchmarks, but that the algorithm might not do well on benchmarks for which it was not specifically trained. To investigate this possibility, two additional traces were used. These traces were also used in [1]. The two traces are

- **SPICE**, an execution of the circuit modelling program.
- **LISP**, an execution of a LISP interpreter.

SPICE is a floating point intensive numeric program, while *LISP* uses symbolic manipulations. In short these are two quite different applications For both benchmarks, the *LRU-History* algorithms proved superior to simple LRU—for both instruction and data caches. This can be seen at the bottom of Figures 5.17 and 5.18.

As noted in Section 5.1 on *Methodology and Details*, one of the reasons for using genetic algorithms to optimize hit rate for individual benchmarks was to provide a rough upper bound on how well this overall approach can do. Here with the overall approach, we are optimizing for the suite of benchmarks and not any one benchmark. Hence we expect the resulting algorithm to perform more poorly than the specially optimized algorithms of Section 5.2.2.

The performance of the individual and suite approaches can best be measured in a manner similar to the *LRU-OPT gain* discussed in Section 5.2.2. For example, the *kalman* benchmark has a *line hit rate* of 30.8% using simple LRU, of 33.4% using an *LRU-History* algorithm optimized specifically for *kalman*, and of 32 1% using the *LRU-History* algorithm optimized for the entire suite of benchmarks. The *individual-suite ratio* is then $\frac{32.1-30.8}{33.4-30.8} = 50.0\%$. In other words roughly half the gain realizable by using the *LRU-History* method is actually achieved.

The results for all the benchmarks are presented in Table 5.8. They vary wildly. The data cache for the *mcsh* benchmark actually did better with the overall algorithm than with the specially tailored algorithm. Perhaps it was able to better learn certain cases important to *mcsh* by seeing them in the other benchmarks. As noted previously, a data cache for the *ccal* benchmark and the *poly* benchmark actually does better with simple LRU than the

	Individual-Suite Ratio	
	INSTRUCTION	DATA
CCAL	3.0%	-816.7%
EMACS	25.6%	11.3%
KALMAN	50.0%	33.8%
MCSH	83.7%	142.1%
POLY	25.9%	-42.3%
WHETSTONE	11.3%	23.5%
Mean	33.3%	-108.1%

Table 5.8: *Individual-Suite Ratios* for *LRU-History*.

overall *LRU-History* algorithm. This causes them to have large negative *individual-suite ratios*.

It is also interesting to note that the best algorithm for instruction caches is significantly different than the best for data caches. When the best *LRU-History* algorithm for data caches (for the suite of benchmarks) is used with an instruction cache, the performance is worse than simple LRU—43.8% *line hit rate* versus 43.9% *line hit rate*.

Likewise when the best *LRU-History* algorithm for instruction caches (for the suite of benchmarks) is used with a data cache, its performance is substantially worse than simple LRU—71.3% *line hit rate* versus 73.7% *line hit rate*.

Finally it should be noted that there is wide variety in the performance of different strings in a given population. Figure 5.19 shows the best, worst, and mean performance by generation of the *LRU-History* technique applied to an instruction cache for the entire suite of benchmarks. Note that the performance of the worst strategy does not show much improvement over the generations and actually often declines. This is largely due to the high mutation rate (10%). With such a high rate, unfit mutants are likely to be produced at each generation.

Also note that the best string from the entire simulation has a *line hit rate* almost 3 times greater than the worst string from the entire simulation (45.7% versus 16.1%). A

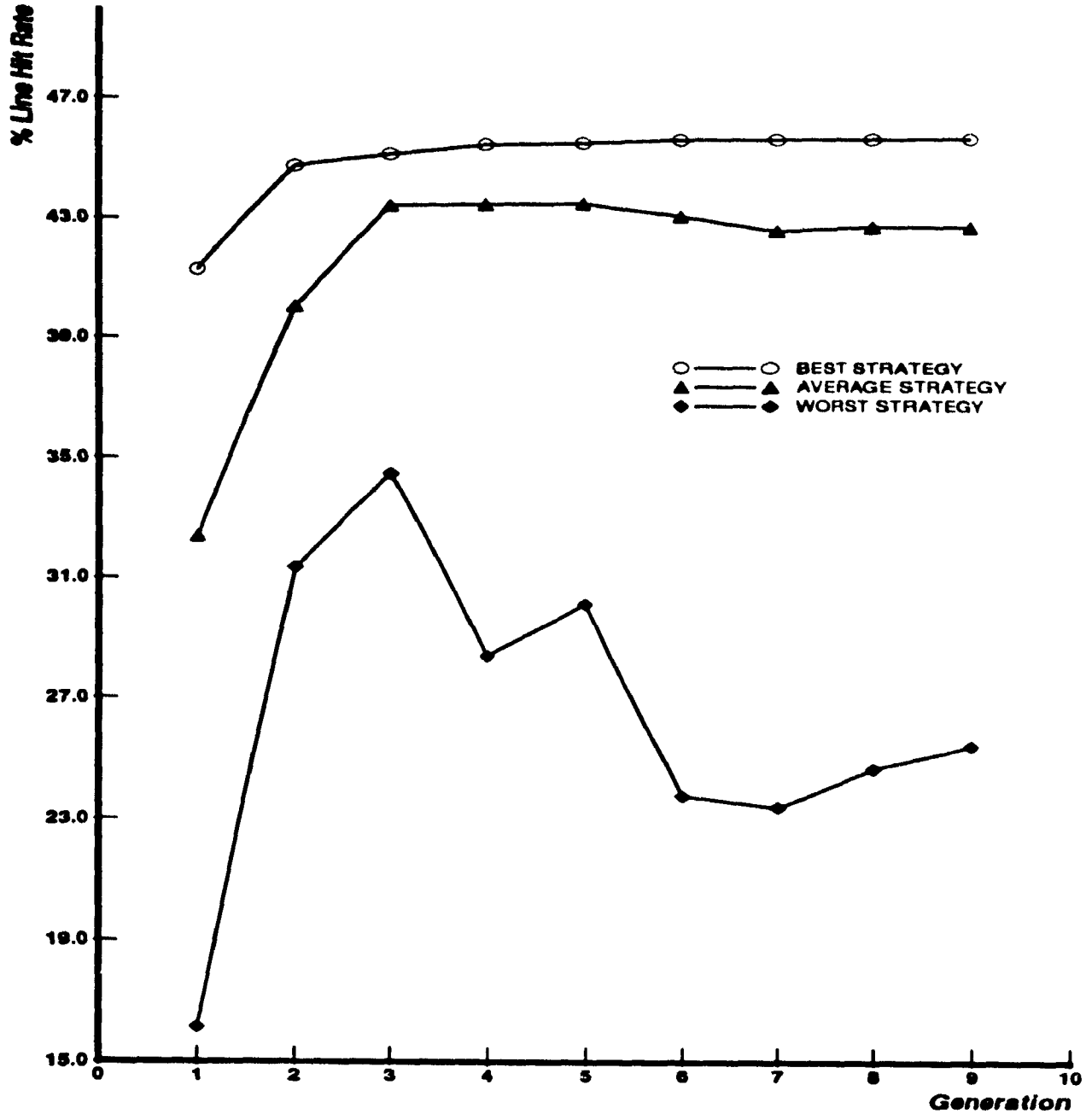

Figure 5.19: Best, Worst, and Mean Performance of *LRU-History* Strings on Suite of Benchmarks for 512 byte I-cache, 4-way associativity, 32 byte lines.

ratio of 3 between best and worst is actually quite small. The ratios range from 1.1 for a data cache with *FIFO-History* to 1506 for an instruction cache with *LRU-Count*. In many cases the *line hit rates* are less than 1% for the worst strings. In other words, when the cache tries to access a different line in a set than it accessed the previous time, it is almost never there! The best and worst strings for each approach are plotted in Figure 5.20 for instruction caches and Figure 5.21 for data caches. Note that the X-axis has a log scale.

Curiously *LRU-History* and *FIFO-History* have by orders of magnitude, the smallest ratios for both instruction and data caches. It is not clear why the bad strings from these two approaches should do so much better than the bad strings from the other approaches.

5.2.4 Random Performance

In order to corroborate that the performance of the genetic algorithm approach is better than random chance, two checks were made. *First*, a histogram was made charting the generations at which the best strings occurred. If the genetic algorithm were ideal, performance would increase at each generation, and the best strings for the entire simulation would always occur in the last generation, i.e. generation 9. On the other hand, if the genetic algorithm approach behaves as a random search, then the occurrence of best strings should be uniformly distributed among the generations with a mean in the middle, generation 5.

Luckily for this work, the results fall much closer to the ideal genetic algorithm case than to the random case. For instruction caches, the mean generation with the best string is 6.8, while for data caches the mean generation is 7.6. Values for each approach are given in Table 5.9. Overall *LRU-History* finds it best strings latest. As noted previously, *LRU-History* is also generally the best performing of the six techniques tried.

Figure 5.22 shows the distribution of "best strings" across generations. As the mean values suggest, the vast majority of bests occurred in generations 7, 8, and 9. Note that these results are in some sense conservative: if two or more generations produced "best strings" of equal value, and these were the best strings of the whole simulation, then the earlier generation was used in tabulating these figures.

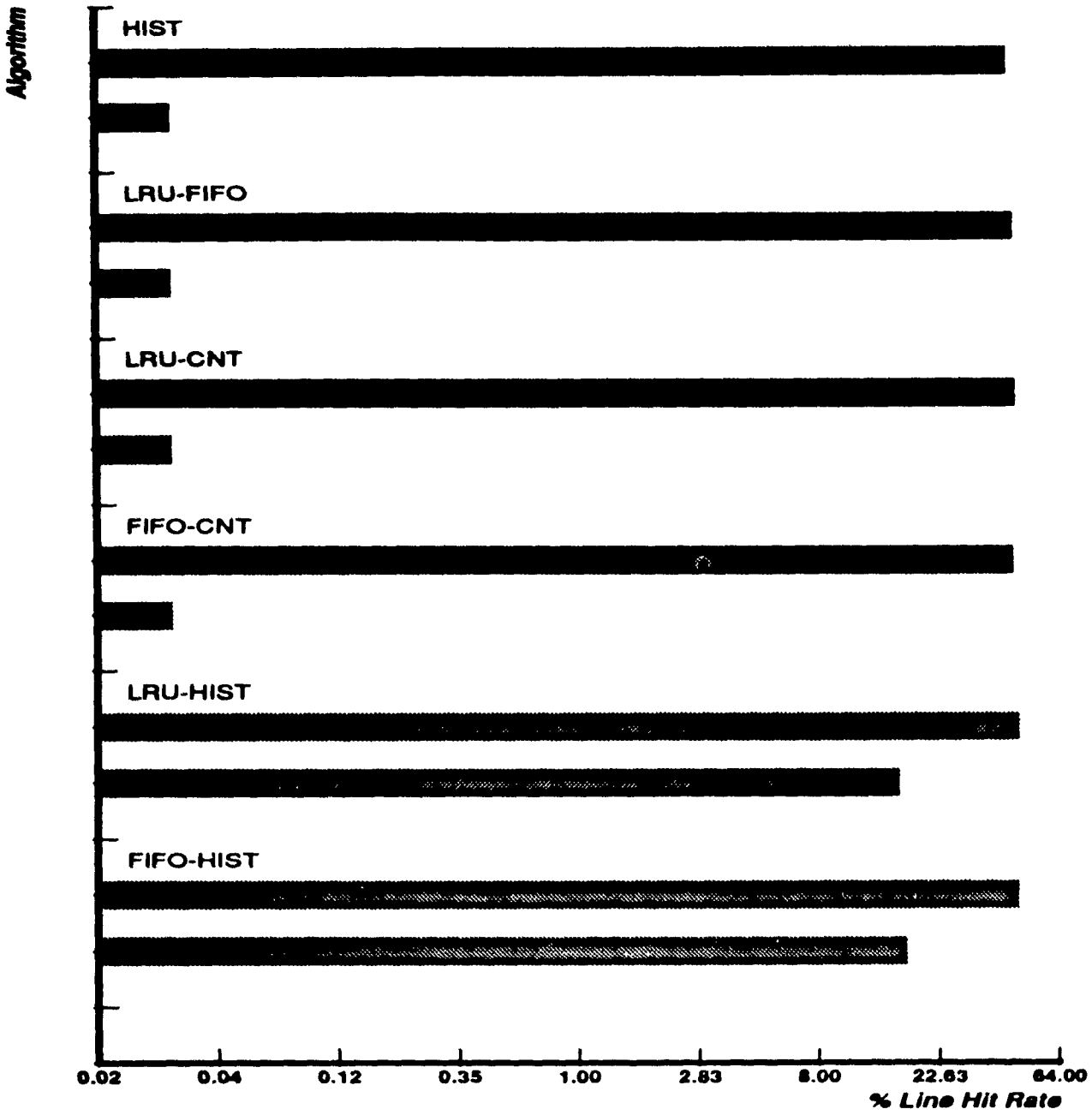

Figure 5.20: Performance of Best and Worst Strings for Suite of Benchmarks for 512 byte I-cache, 4-way associativity, 32 byte lines.

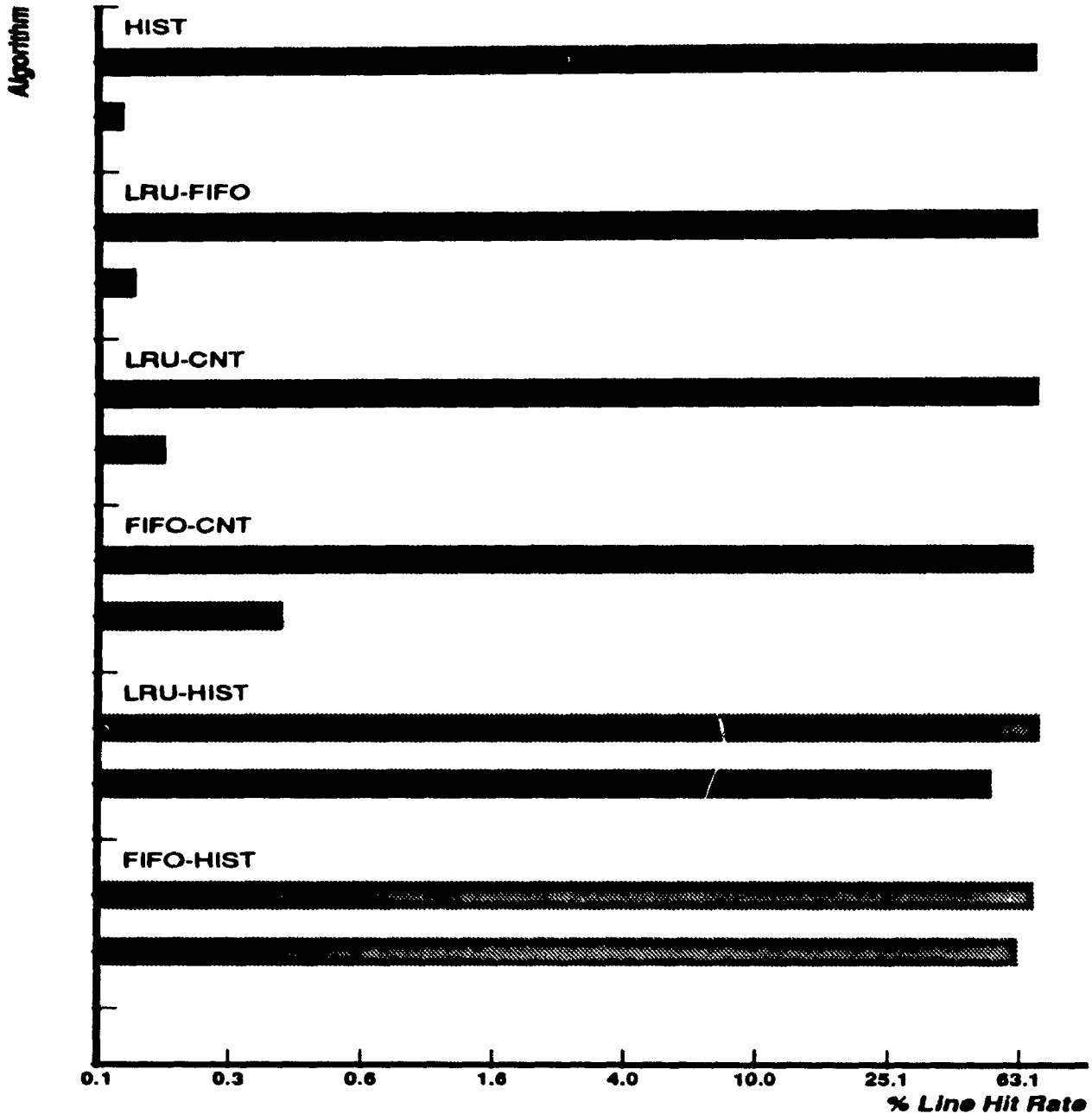

Figure 5.21: Performance of Best and Worst Strings for Suite of Benchmarks for 512 byte I-cache, 4-way associativity, 32 byte lines.

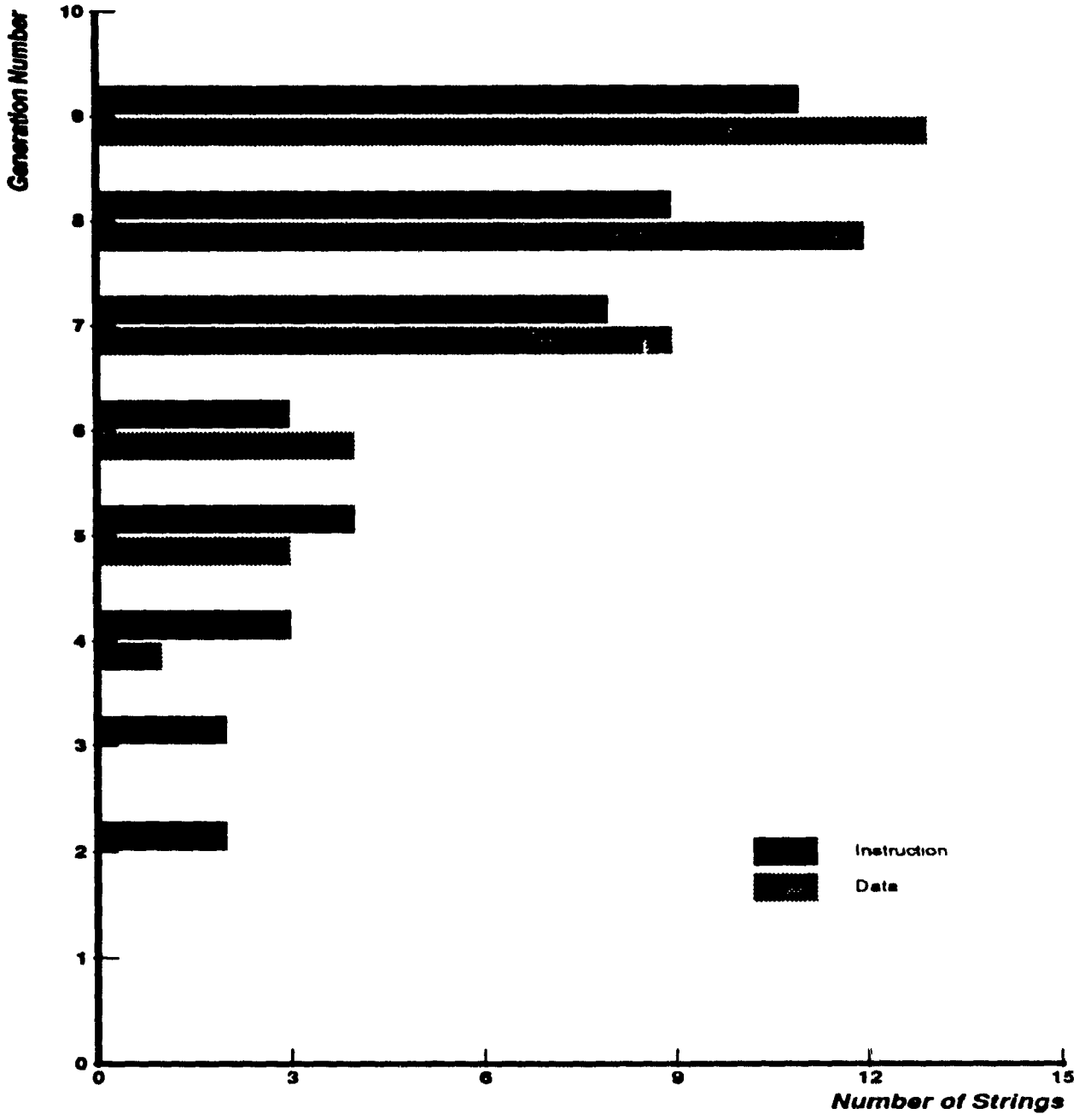

Figure 5.22: Distribution of Generations at which *Best Strings* Occurred.

	History	LRU-FIFO	LRU-Cnt	FIFO-Cnt	LRU-Hist	FIFO-Hist
Instruction	5.7	7.1	6.7	6.3	8.6	6.6
Data	7.7	8.3	7.9	6.6	7.6	7.6

Table 5.9: Mean Generation at which *Best String* Occurred.

The *second* method used to check the usefulness of the genetic algorithm approach was to run simulations using the same basic techniques—*History*, *LRU-FIFO*, *LRU-History*, etc.—but instead of starting with an initial population and using the genetic algorithm approach to improve performance, a random population of strategy strings was created. The size of the population was 800, roughly the number of *different* strings produced in all the generations of a genetic algorithm simulation.

In all but 4 of the 84 benchmark/algorithm combinations, the best string produced using the genetic algorithm approach was superior to the best performance of string found in a random population. In those 4 cases, the performance of the random string was the same, not better than the best string found using a genetic algorithm approach.

The percentage difference in *line hit rate* between the random and genetic algorithm approaches are shown in Figures 5.23 and 5.24, for instruction and data caches respectively. Note that a log scale is employed because of the wide range differences—from 0.01% for an instruction cache using the *History* method used on a suite of benchmarks to 12.36% for an instruction cache using the *LRU-Count* method on the *mcsh* benchmark.

Note that the 4 cases with no difference in performance are not depicted. Three cases occurred with an instruction cache. They are *History/whetstone*, *FIFO-Count/poly*, and *FIFO-Count/kalman*. The data cache case occurred for *FIFO-Count/mcsh*. From this and from viewing Figures 5.23 and 5.24, it is clear that *FIFO-Count* and *History* do not gain much from the genetic algorithm approach. However, *LRU-Count* and *LRU-History* appear to derive significant benefit. The mean difference in *line hit rates* across benchmarks and between random and genetic algorithm approaches is shown in Table 5.10.

Not surprisingly techniques such as *LRU-Count*, in which a genetic algorithm approach finds a much better string than found by random search, tend to show much more im-

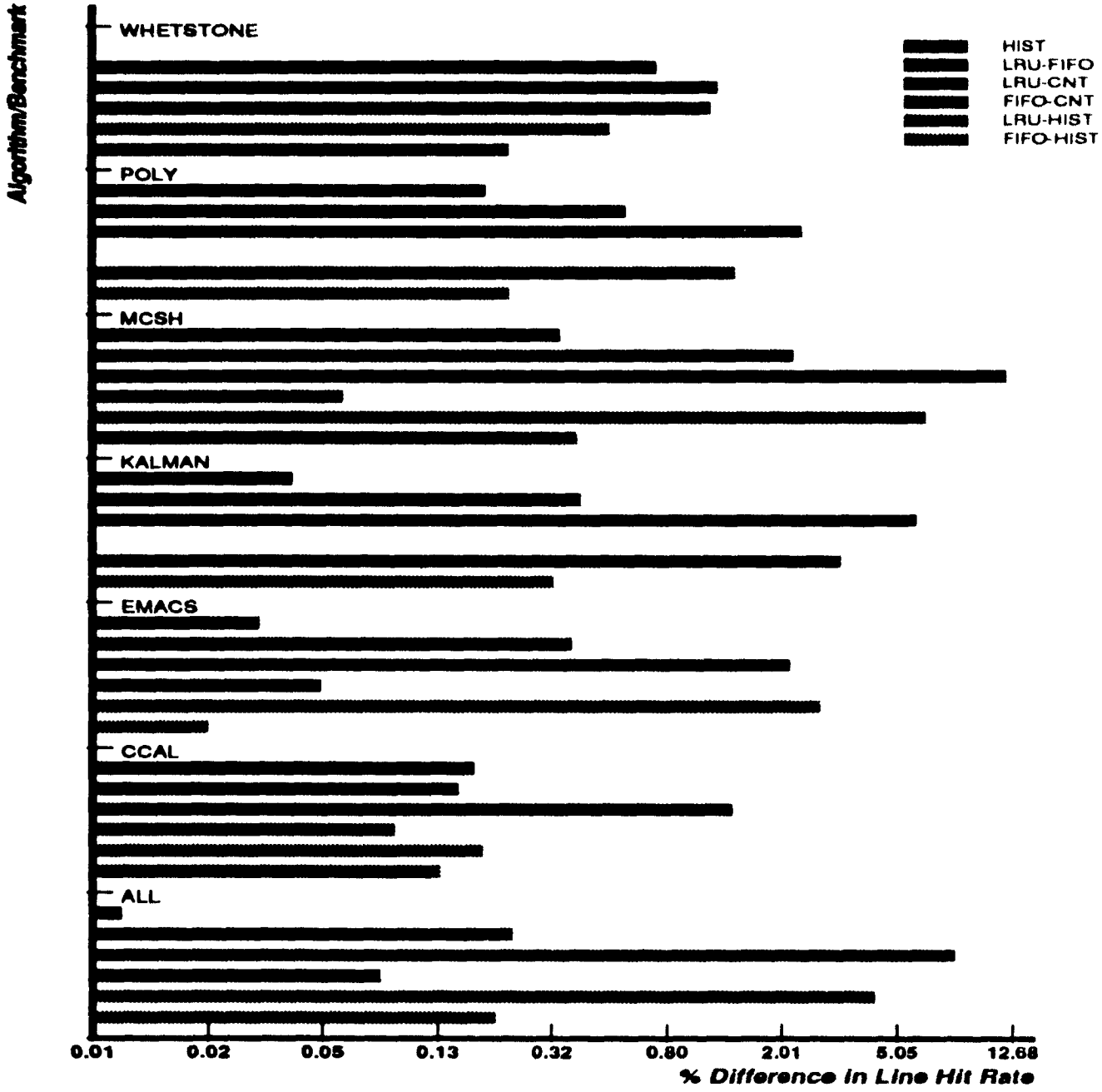

Figure 5.23: Percentage Differences in *Line Hit Rates* between Best Strings Generated by Genetic Algorithm Approach and Best Strings Generated by Random Approach for a 512 byte I-cache, 4-way associativity, 32 byte lines.

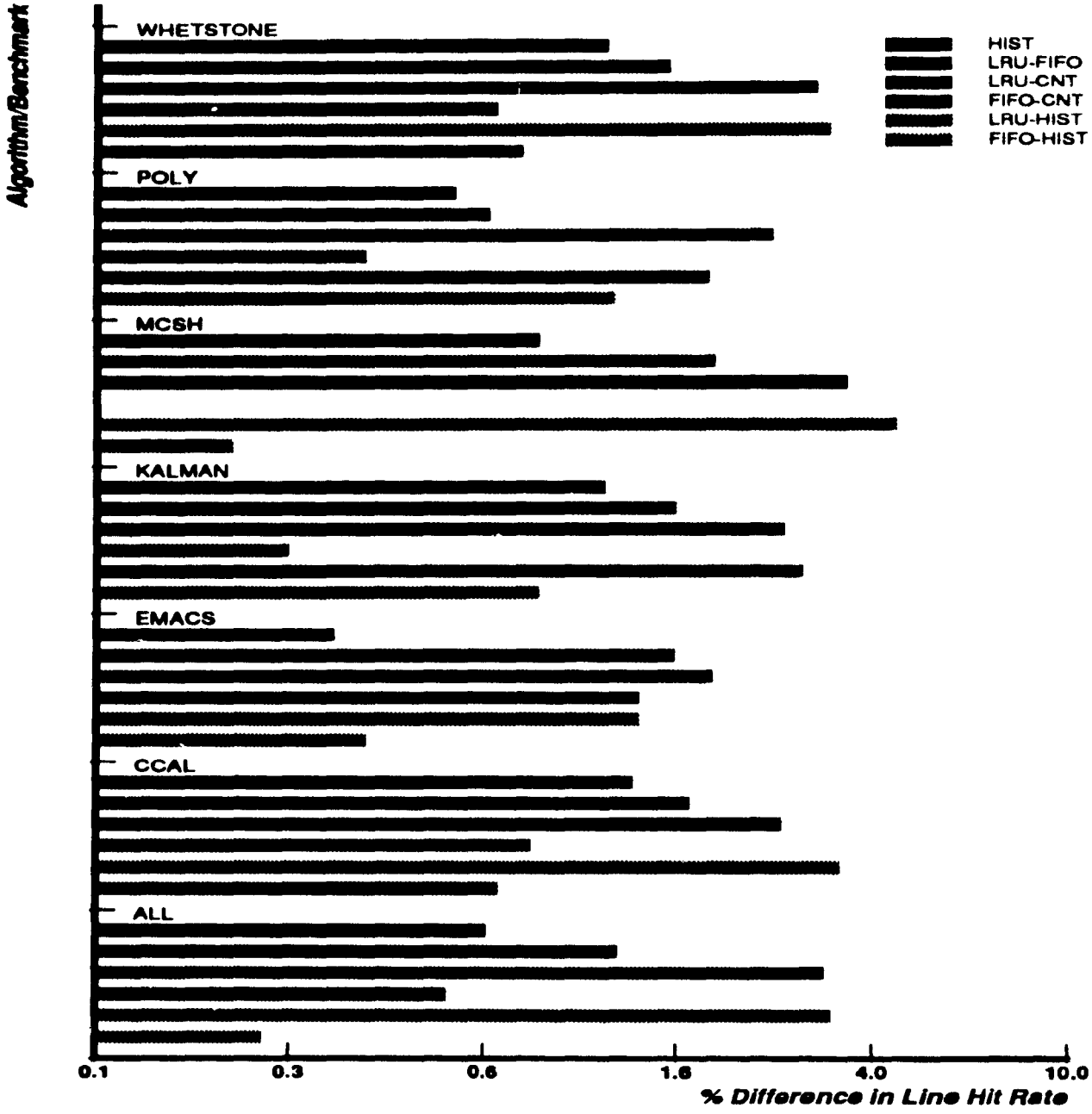

Figure 5.24: Percentage Differences in *Line Hit Rates* between Best Strings Generated by Genetic Algorithm Approach and Best Strings Generated by Random Approach for a 512 byte D-cache, 4-way associativity, 32 byte lines.

	History	LRU-FIFO	LRU-Cnt	FIFO-Cnt	LRU-Hist	FIFO-Hist
Instruction	0.11	0.69	4.80	0.21	2 70	0.22
Data	0.84	1.44	2.78	0.56	2.93	0.60

Table 5.10: Mean Percentage Differences in *Line Hit Rates* between Best Strings Generated by Genetic Algorithm Approach and by Random Approach.

provement from the first to the last generation than those methods such as *History* where the genetic algorithm and random approaches perform more similarly This is graphically depicted in Figures 5.25 and 5.26.

It is worthwhile reiterating that the choice of parameters makes a difference here. For example, when a mutation rate of 1% is used instead of 10%, random selection finds a better string for 3 of the 6 individual benchmarks when using the *LRU-FIFO* approach on an instruction cache. However, as already noted, with a mutation rate of 0.1, the genetic algorithm approach found a better string for all 6 benchmarks. Clearly when the mutation rate is too low, the genetic algorithm simulation can become mired in a far from optimal solution. In such a case, randomly selecting strategies can work better, because a more diverse portion of the solution space is explored.

Finally, use of a large random population gives a good basis upon which to test the choice of another of the parameters to the genetic algorithm: *the population size*. After simulating the initial generation of 800 randomly chosen strings, an additional generation can be created using the standard genetic algorithm operators. This was done and the performance measured. In 67 of the 84 benchmark/algorithm combinations, the standard approach of simulating 100 strings for 9 generations was better. In 5 cases the two approaches produced strings of equal capability, and in 12 cases, simulating 800 strings for 2 generations was better. Table 5.11 shows the mean differences in performance. Clearly use of a smaller population for more generations is a better choice overall.

This is accentuated by the fact that 1600 strings (2 generations $\times$ 800 strings per generation) were simulated for the large population versus only 900 for the small population

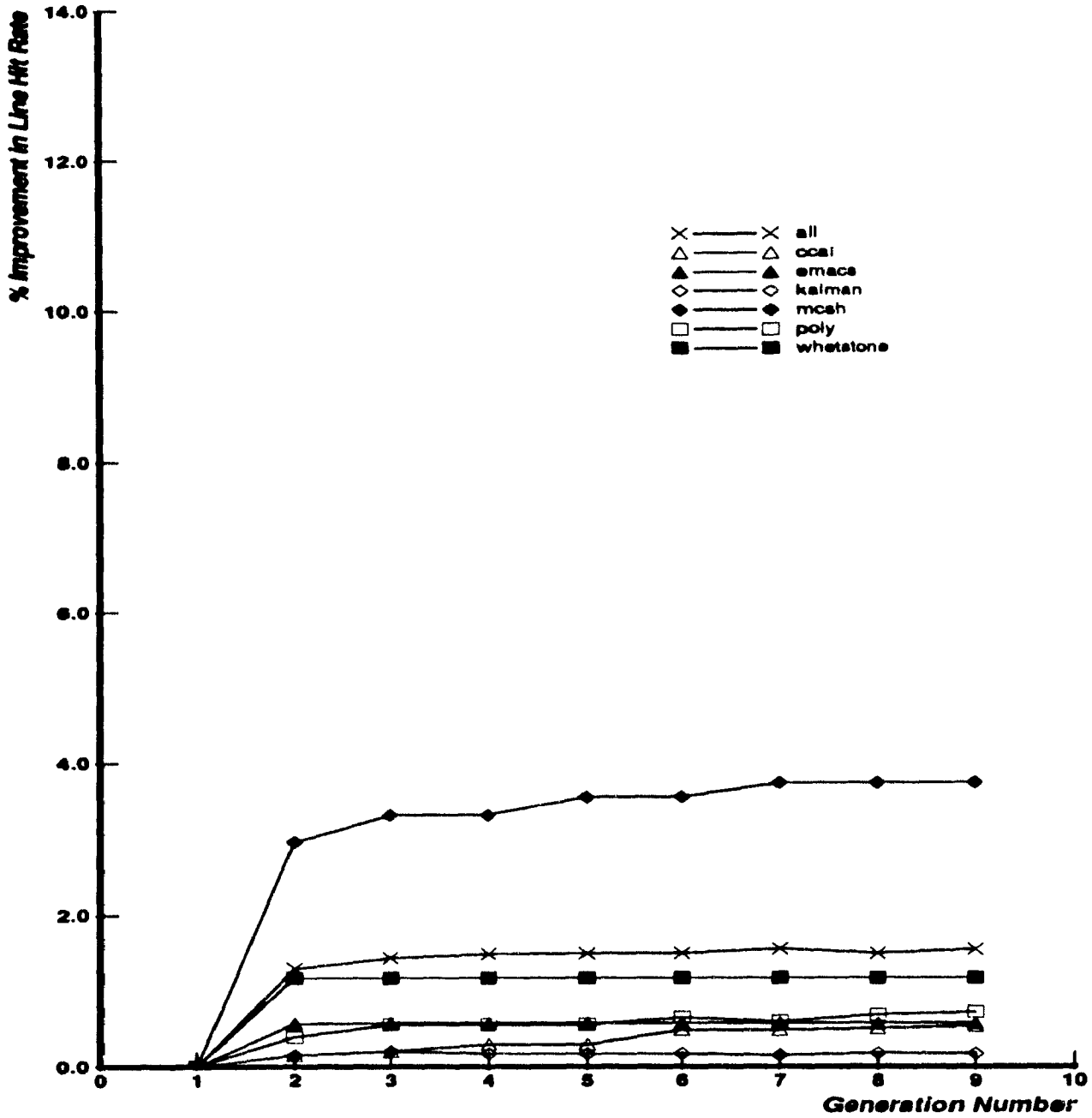

Figure 5.25: Improvement in *Line Hit Rate* by Generation using the *History* Method for a 512 byte I-cache, 4-way associativity, 32 byte lines.

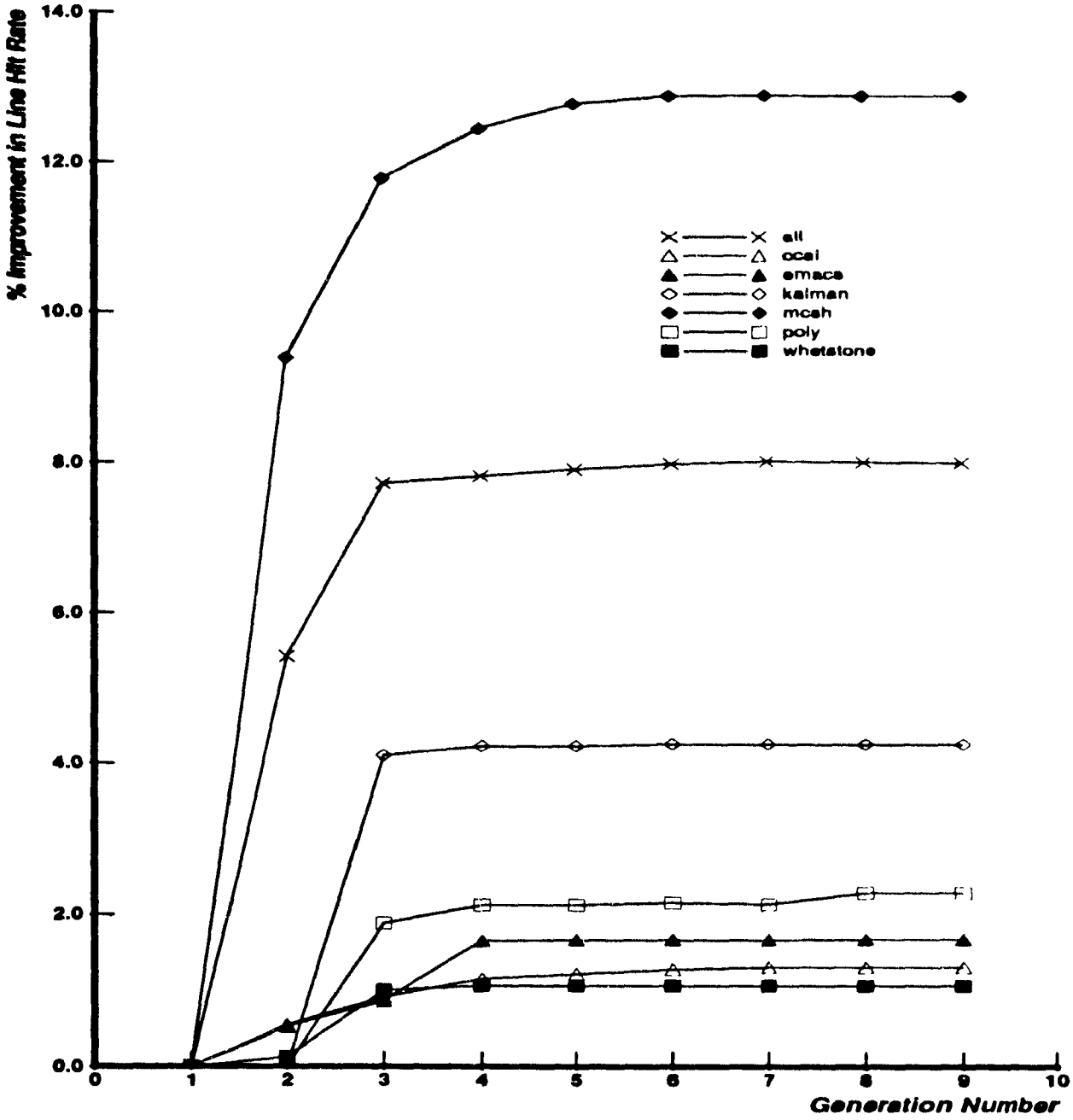

Figure 5.26: Improvement in *Line Hit Rate* by Generation using the *LRU-Count* Method for a 512 byte I-cache, 4-way associativity, 32 byte lines.

	History	LRU-FIFO	LRU-Cnt	FIFO-Cnt	LRU-Hist	FIFO-Hist
Instruction	0.02	0.19	0.91	0.10	0.36	0.08
Data	0.30	0.58	0.29	0.04	0.32	0.27

Table 5.11: Mean Percentage Differences in *Line Hit Rates* between Best Strings Generated by Population of 100 for 9 Generations *and* by a Population of 800 for 2 Generations.

(9 generations × 100 strings per generation). Actually both the 900 and the 1600 are somewhat overstated, as some strings survive from generation to generation. Hence the number of *unique* strings simulated is somewhat less than 900 or 1600.

5.3 OPT Match Rate

Misses for replacement methods other than OPT can be broken into two groups:

1. Misses at which OPT also missed
2. Misses at which OPT did not miss.

For misses in the first group, the choice of line replacement can be compared to OPT. The fraction of misses in the first group is often quite large. For these benchmarks and an instruction cache, a mean of 75% of LRU misses are in the first group. For FIFO the number is 74%. For data caches the numbers are 80% for LRU and 72% for FIFO.

Note that at each point where OPT misses, LRU also misses [29]. Of course, LRU misses at many additional points as well. To show that LRU misses every time OPT does, assume that this is not true; that there are some occasions when OPT has a miss and LRU does not.

To clarify this, see Figure 5.27 in which the same address stream is depicted under both LRU and OPT replacement. Assume that at point *2*, *A* is accessed and that it results in a

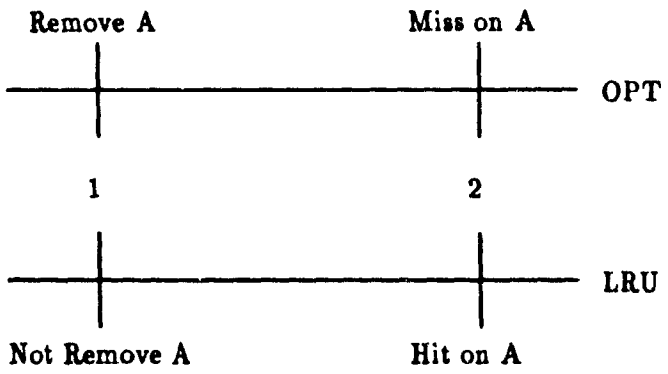

Figure 5.27: Misses under LRU and OPT.

miss for OPT, but a hit for LRU. Thus OPT must have replaced *A* at an earlier point, *1*, at which LRU chose to leave A in cache. However, since OPT chose to remove *A* at point *1*, every other line in the set must be accessed in the interval *1–2*. This is because OPT removes the line whose reference is furthest in the future. If OPT removes *A*, it must be because it is referenced further in the future than any other line in the set.

However, if every other line in the set is accessed in the interval *1–2*, then there is insufficient room to hold all of these lines and *A*. Hence before point *2*, LRU must replace *A* in the cache as the *least recently used* line. But if LRU has replaces *A*, it will have a miss at point *2*, just like OPT. But this contradicts the premise that OPT has a miss, while LRU does not. Thus it must be the case that LRU has a miss whenever OPT does. Note that this does not hold true for other replacement methods, such as FIFO.

Figures 5.28 to 5.31 provide a complete breakdown of how the various methods compare to OPT. Each histogram bar is broken into four parts:

1. *Chosen:* Instances where the replacement policy chose the same line to replace as did OPT.

2. *Not Bring:* Instances where OPT chose not to bring the new line into cache. This option of not bringing in a new line is technically possible for other replacement algorithms, but generally results in decreased performance. Hence other algorithms

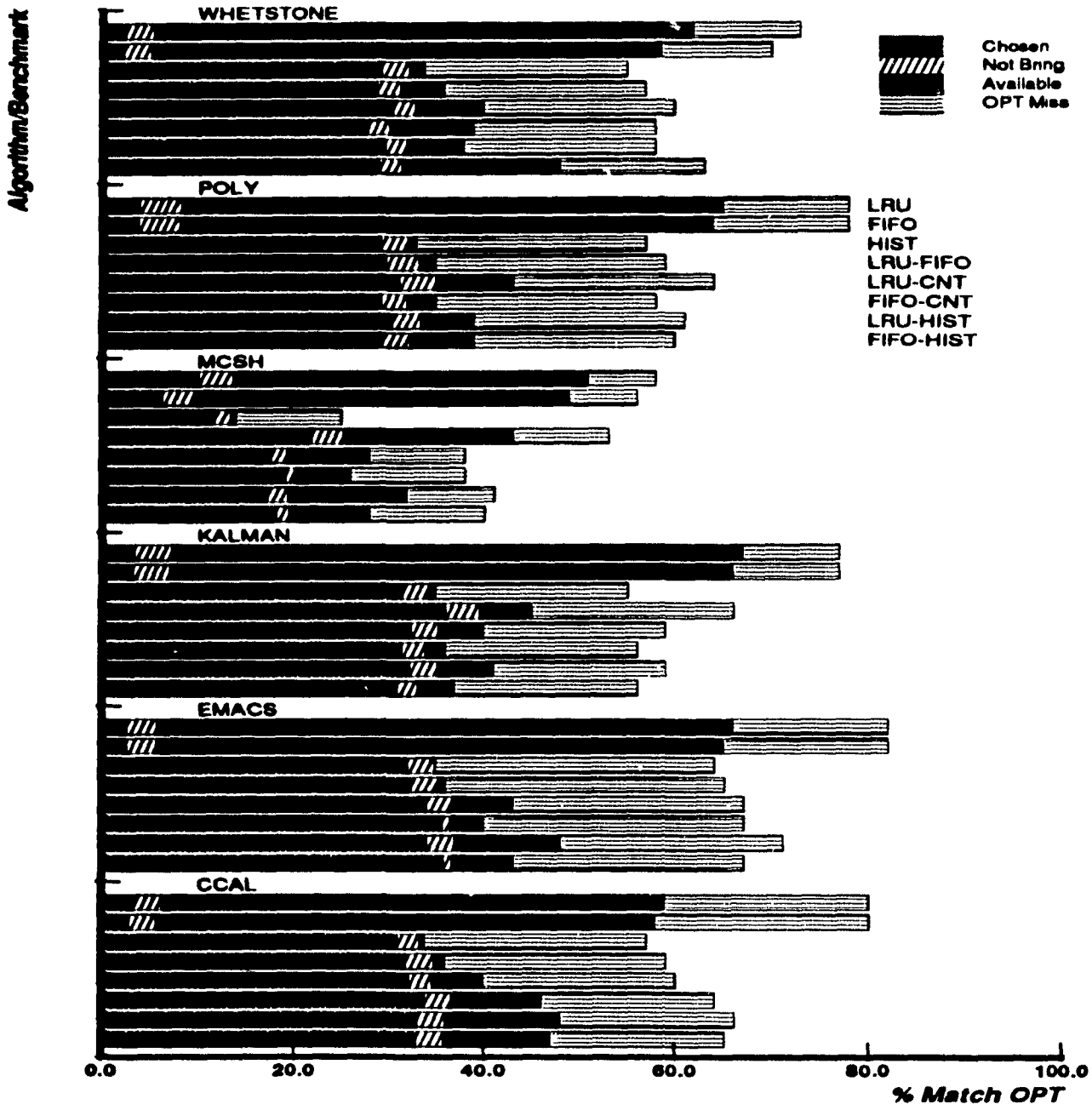

Figure 5.28: Behavior of Replacement Policies Compared to OPT. Genetic Algorithm Policies have *OPT Match Rate* Maximized. 512 byte I-cache, 4-way associativity, 32 byte lines.

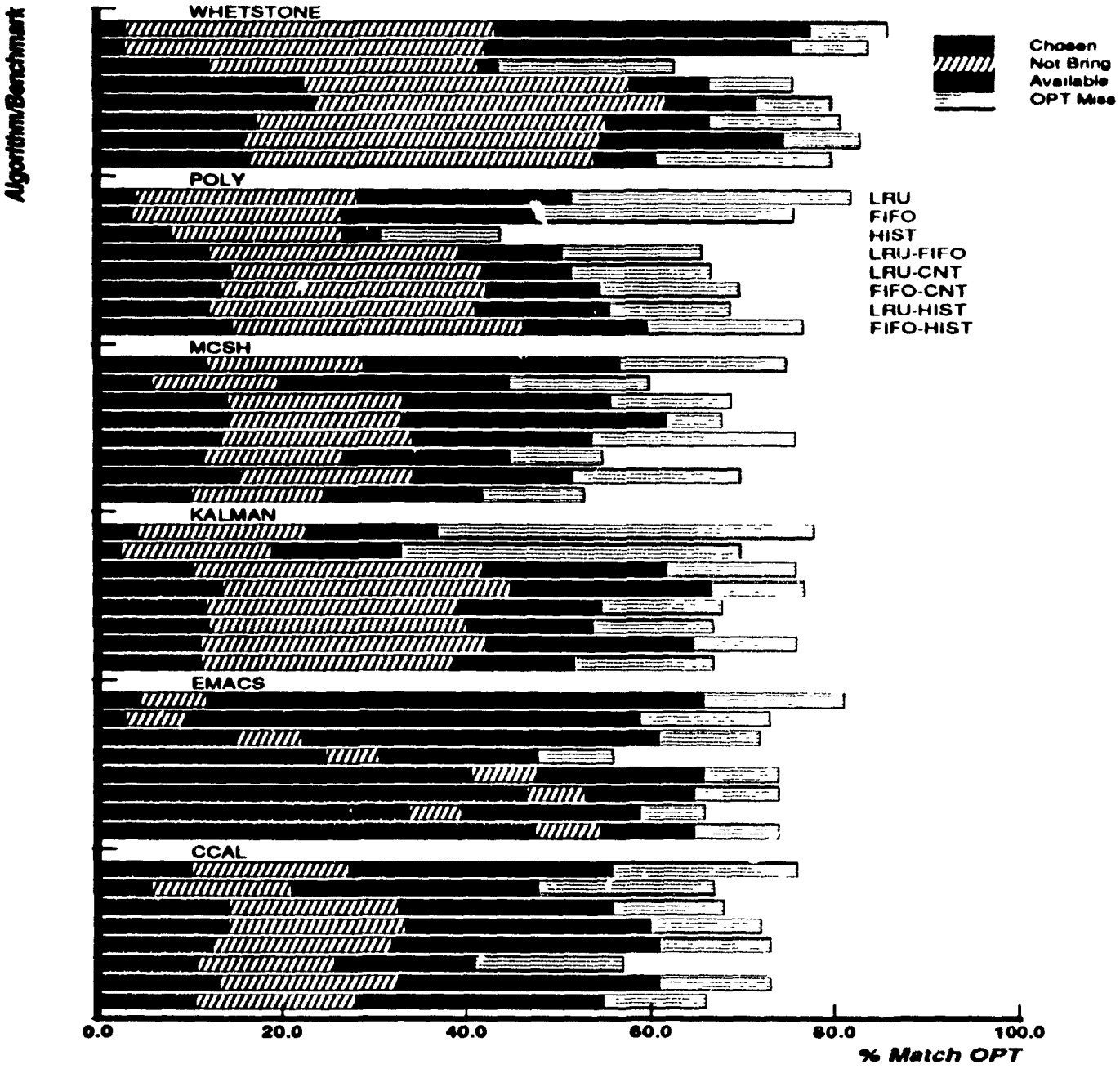

Figure 5.29: Behavior of Replacement Policies Compared to OPT. Genetic Algorithm Policies have *OPT Match Rate* Maximized. 512 byte D-cache, 4-way associativity, 32 byte lines.

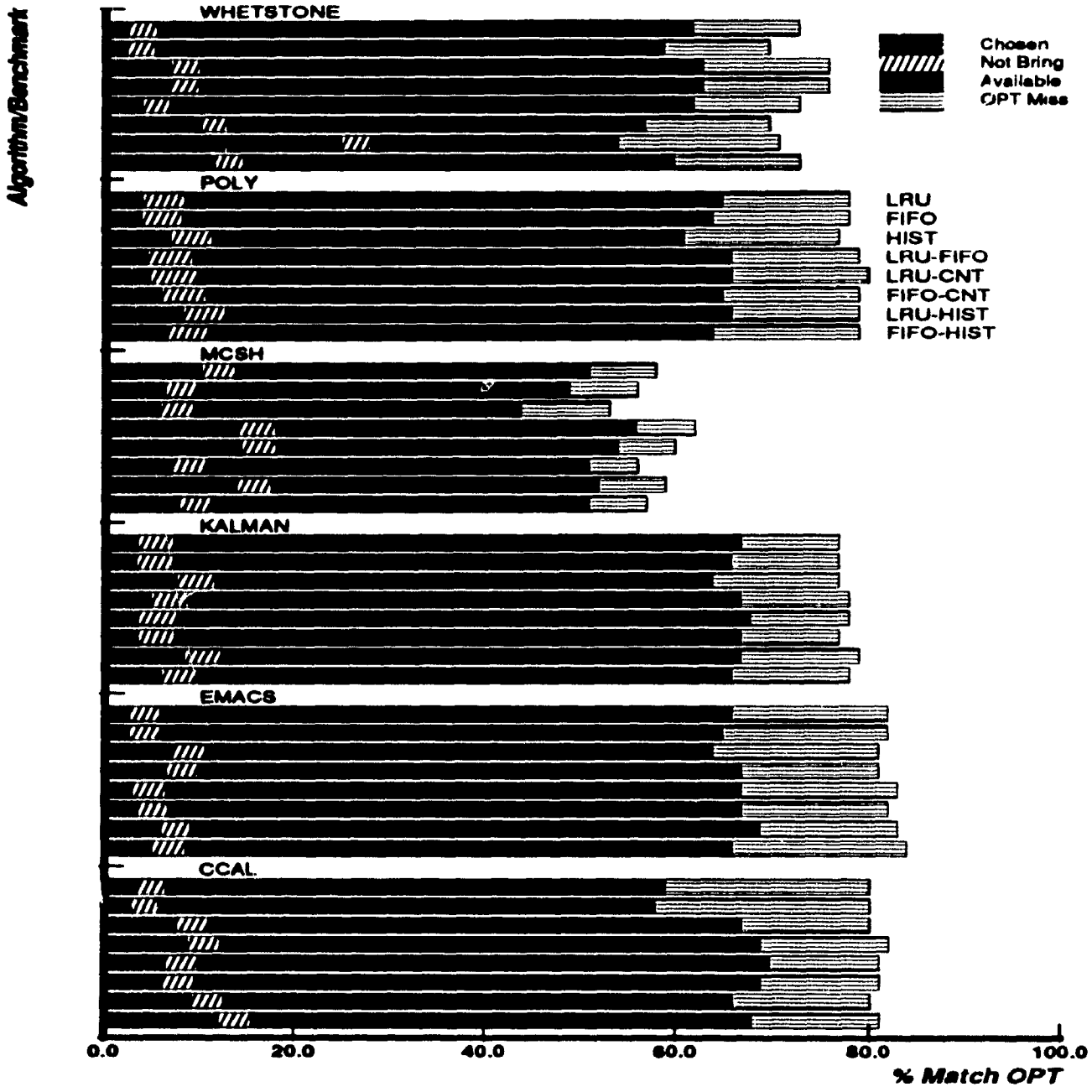

Figure 5.30: Behavior of Replacement Policies Compared to OPT. Genetic Algorithm Policies have *Line Hit Rate* Maximized. 512 byte I-cache, 4-way associativity, 32 byte lines.

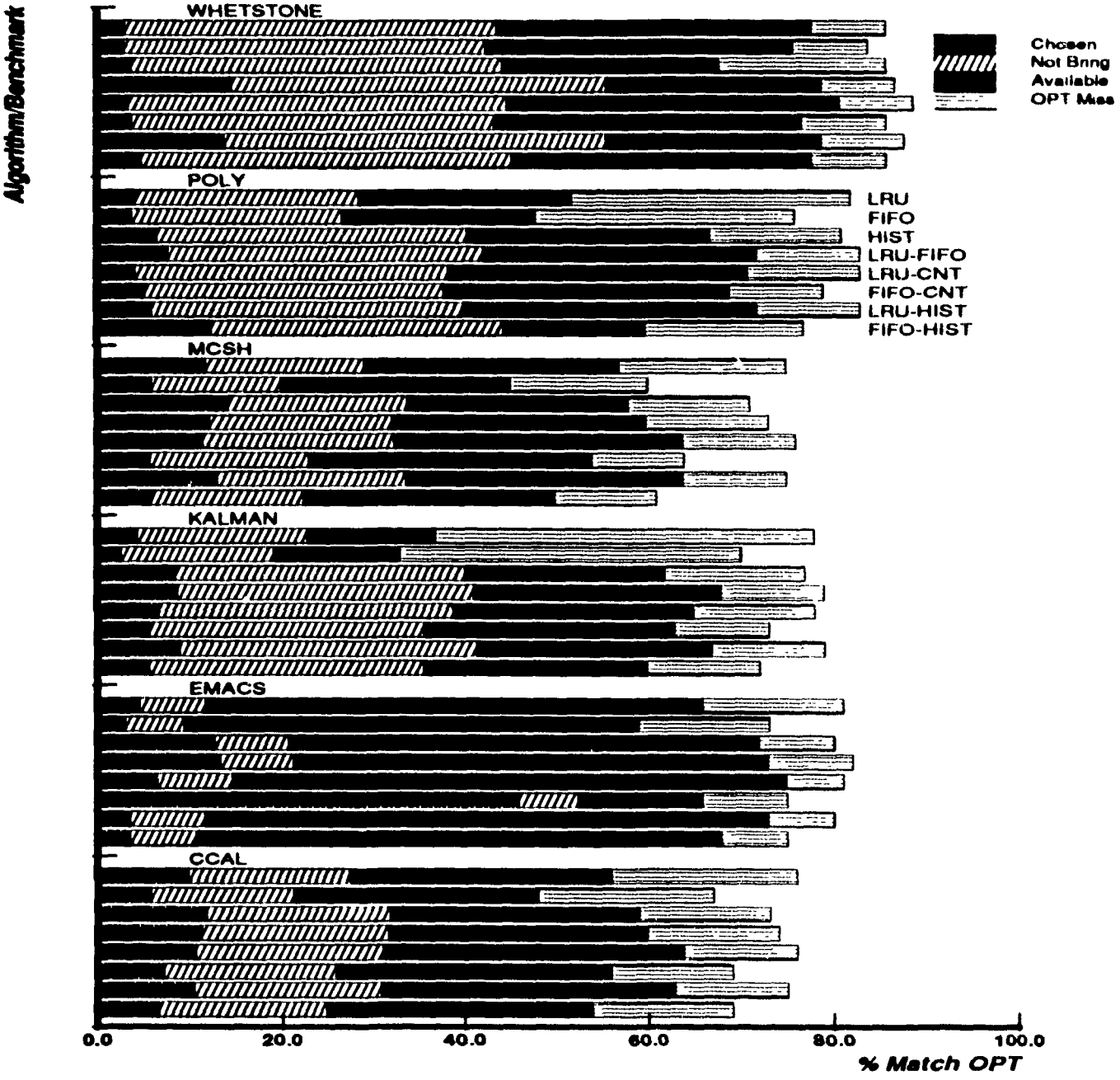

Figure 5.31: Behavior of Replacement Policies Compared to OPT. Genetic Algorithm Policies have *Line Hit Rate* Maximized. 512 byte D-cache, 4-way associativity, 32 byte lines.

always bring in the new line.

3. *Available:* Instances where the line OPT chose to replace is present in cache or as the new line being referenced. In other words, these are all instances where the replacement policy *can* behave as OPT did. The first two instances are subsets of this one.

4. *OPT Miss:* Instances where OPT also missed at this point, i.e. the group 1 misses at the start of this Section. *OPT Miss* is the overall length of the histogram bar, and the first three instances are subsets of this one. Since there is an OPT miss for every LRU miss, the length of the LRU histogram indicates the number of OPT misses.

One of the chief goals in this section is to maximize the size of the first part of the histogram for *History*, *LRU-FIFO*, etc, i.e. **the goal is maximize the fraction of misses when the genetic algorithm approaches behave exactly as OPT would.** The fraction of misses when OPT is correctly mimicked is used as the metric in this Section, as *line hit rate* was used in Section 5.2. The results of this goal of "matching OPT" can be seen in Figures 5.28 and 5.29 for instruction and data caches respectively.

Before continuing, it should be pointed out that in this section, results are presented from individual benchmarks only. None are from the suite of all benchmarks. This was necessary because of the way in which the simulator was coded. To compare other methods to OPT, OPT is run and a record of its selections made. Then the other replacement policies are run and their choices compared to OPT's. To make this comparison, it must be possible to align the results from the OPT run and the subsequent runs.

This is no problem for single benchmarks. However recall that in Section 5.2.3, it was stated that multitasking is simulated by varying the number of addresses processed for a benchmark in its "time slice" depending on the number of misses. The result is that different replacement policies process slightly different streams of addresses depending on when misses occur. These different streams make it impossible to align the address references of OPT with those of other replacement policies.

Despite this omission, the individual benchmarks provide a variety of useful and interesting results. By looking at the *Chosen* category in Figures 5.28 and 5.29, it is immediately clear that the genetic algorithm replacement policies can be tailored to do far better than

	INSTRUCTION		DATA	
	Maximize OPT Match Rate	Maximize Line Hit Rate	Maximize OPT Match Rate	Maximize Line Hit Rate
LRU	4.4	4.4	6.4	6.4
FIFO	3.6	3.6	4.0	4.0
History	27.6	7.0	12.4	9.8
LRU-FIFO	30.2	7.6	17.0	11.5
LRU-Count	29.8	6.0	19.5	7.3
FIFO-Count	29.6	6.1	18 7	12.3
LRU-History	29.6	11.8	17.0	9.4
FIFO-History	29.4	8.2	18.5	6.6

Table 5.12: *OPT Match Rates*: Percentage of Misses in which Different Algorithms Replaced the Line OPT Would Have.

standard LRU or FIFO. While LRU chooses the line OPT would have 4.4% of the time and FIFO 3.6% of the time on average for an instruction cache, *LRU-FIFO* makes this choice 30.2% of the time! When the *History* method is used on *emacs* in an instruction cache, the OPT replacement line is *chosen* almost every time it is *available*. (See Figure 5 28) The means for each method are displayed in Table 5.12.

For data caches the differences are also large, but not quite as dramatic LRU matches OPT 6.4% of the time on average, while the best genetic algorithm policy, *LRU-Count* matches OPT 19.5% of the time.

Given the vast superiority of the genetic algorithm policies, a natural question arises: Is their performance in matching OPT an inherent part of these policies or simply the result of having used genetic algorithms to maximize the OPT Match Rate. The answer appears to lie somewhere in the middle.

Figures 5.30 and 5 31 show the same replacement policies, but in these Figures the "genetic algorithm" policies were optimized for *line hit rate*, not *OPT match rate* The result is that the genetic algorithm policies do slightly better than LRU and FIFO at

matching OPT, but not nearly so well as when they were specifically optimized to match OPT. For instance, *LRU-History* is the best policy for instruction caches in this case. It has an *OPT match rate* of 11.8% versus 4.4% for LRU. The version of *LRU-History* optimized for *OPT match rate*, matches OPT 29.6% of the time. The mean performance of the different methods is summarized in Table 5.12.

Reviewing Figure 5.28, where policies are optimized for *OPT match rate* in an instruction cache, reveals that although the genetic algorithm replacement policies can do far better than LRU and FIFO at matching OPT's choice of replacement line, they do uniformly worse in making sure that OPT's choice is *available* in cache. The genetic algorithm policies achieve their high rate of matching OPT by almost always choosing the line OPT would have when it is available, *not* by making sure that OPT's choice is available.

Figure 5.30, where policies are optimized for *line hit rate*, indicates that failure to make OPT's choice available is not a basic failing of the genetic algorithm policies. In this Figure, the OPT line is *available* approximately as often as it is for LRU or FIFO, but as already noted, it is *chosen* at a slightly higher rate.

From Figure 5.28, it is clear that for LRU and FIFO the optimal replacement is *available* approximately half the time in an instruction cache, even if the 3% - 4% of instances where no line should be *replaced* are excluded. It is somewhat surprising then, that both LRU (4.4%) and FIFO (3.6%) *choose* so poorly.

Data caches behave quite differently in terms of whether or not the OPT line is available for replacement. As can be seen in Figures 5.31 and 5.29, most of the genetic algorithm policies make the OPT line *available* approximately as much as do LRU and FIFO. However when the genetic algorithm policies are optimized to match OPT's choice, a smaller fraction of their misses are also *Opt Misses*.

This difference in data cache behavior may be partially due to the far higher number of times when it is best not to bring a new line into cache. Under LRU, it is best *not* to bring the new line into a data cache 27.3% of the time on average. (As can be seen in Figures 5.28 to 5.31, the percentage does not vary much from one replacement policy to another.) For an instruction cache, not bringing in a new line is best only 3.6% of the time under LRU. The high percentage in data caches is likely due to *scratch* variables which are used only once.

Given the large number times when it is best not to bring a new line into a data cache, it is worthwhile to briefly re-examine the standard approach of always bringing in the new line. First note the inherent problem: To know when *not* to bring in a line is difficult for simple algorithms working at runtime, especially when they must make use of a brief summary of *past* references to the cache. Unfortunately *past* references are likely to contain few clues about how an entirely new line will be referenced.

Furthermore the principle of temporal locality assumes that if a line is referenced, it will soon be referenced again. Very good knowledge is needed to know when this principle should be violated. (Compiler directives could indicate such "dead" lines, but that is beyond the scope of this work.)

Because of these difficulties, the easiest cases were tried. The *LRU-Count* method was used on the two benchmarks with the highest fraction of lines which are best not brought into cache: *whetstone* with 41.3% and *poly* with 34.3%. The results were slightly worse than the corresponding results where the line was always brought in. For *whetstone*, the *line hit rates* of the best strings were 36.9% with the new line *not* always brought in, versus 38.9% with the new line *always* brought in. For *poly* the numbers were 56.0% to 56.9%. However this *line hit rate* for *whetstone* is slightly higher than the LRU *line hit rate* of 36.3%. For *poly*, LRU is slightly better at 56.4%.

Since *poly* and *whetstone* are the two "best" cases for not bringing in lines, it does not seem likely that these approaches are adequate to determine whether or not to bring in lines. Hence this avenue will not be pursued further.

Thus far, no mention has been made of the hit rate of the genetic algorithm approaches *when optimized to match OPT's choice of replacement line.* Unfortunately, despite their vastly superior performance in mimicking OPT, the overall hit rate in almost all cases is lower than LRU. For instruction caches, the overall hit rate is almost always less than FIFO as well. This comparison is made graphically in Figures 5.32 and 5.33 for instruction and data caches. For each replacement policy, Table 5.13 provides the mean hit rate across benchmarks.

Since the genetic algorithm approaches can mimic OPT quite well, but have a relatively low hit rate, and since LRU and FIFO mimic OPT quite poorly, but have a relatively high hit rate, it appears that mimicking OPT's choice of replacement line is generally not

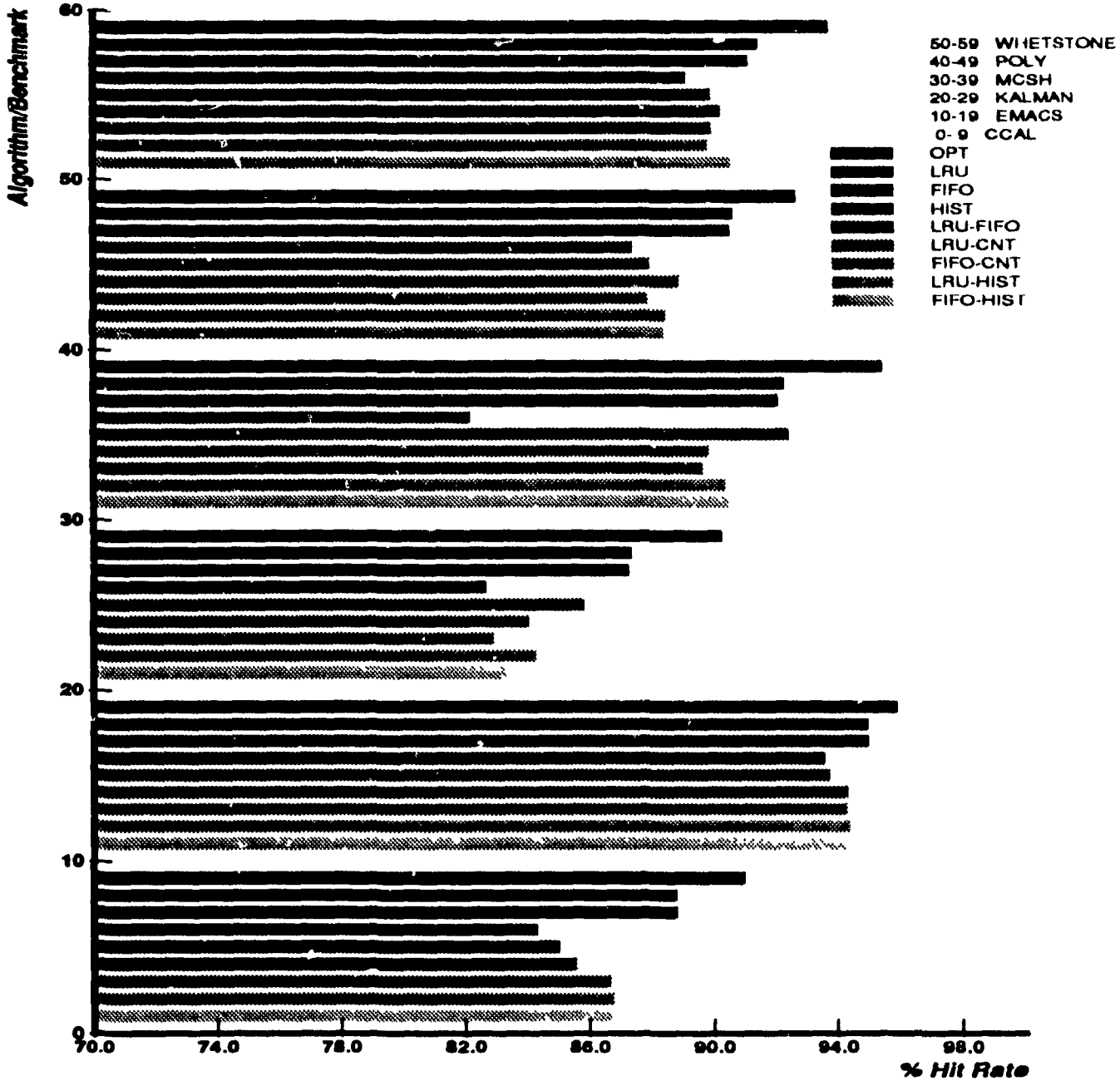

Figure 5.32: Overall Hit Rates When Genetic Algorithm Policies Maximize *OPT Match Rate*. 512 byte I-cache, 4-way associativity, 32 byte lines.

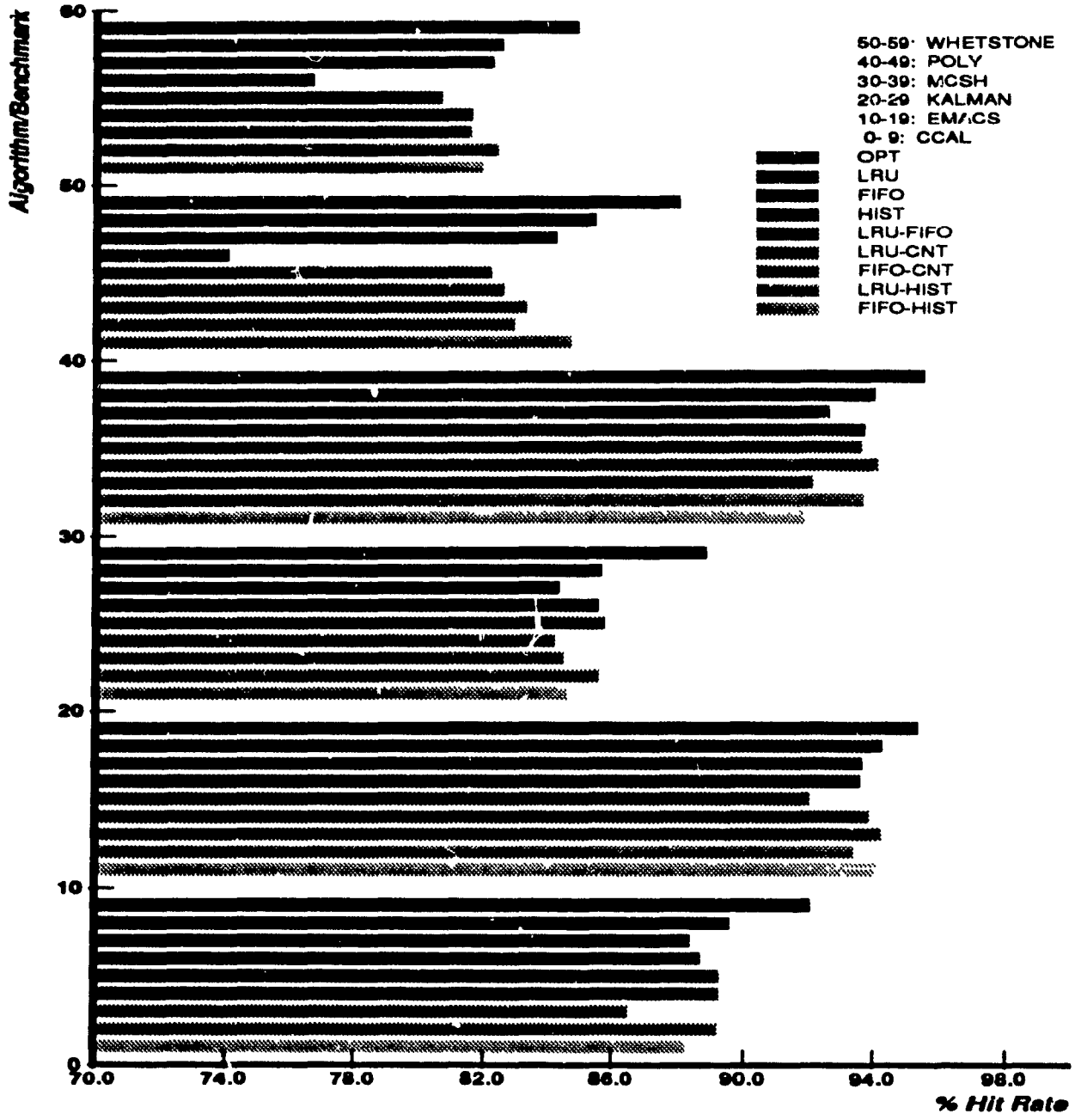

Figure 5.33: Overall Hit Rates When Genetic Algorithm Policies Maximize *OPT Match Rate*. 512 byte D-cache, 4-way associativity, 32 byte lines.

	INSTRUCTION	DATA
LRU	91.0	88.6
FIFO	90.8	87.6
History	86.6	85.4
LRU-FIFO	89.1	87.3
LRU-Count	88.9	87.6
FIFO-Count	88.6	87.0
LRU-History	89.0	87.9
FIFO-History	89.0	87.5

Table 5.13. Mean Overall Hit Rates When Genetic Algorithm Policies Maximize *OPT Match Rate.*

important in achieving a high hit rate.

5.4 History

In this section three aspects of the *history* method are explored further:

1. The representation of strings is of great importance. This is explored next in Section 5.4.1.

2. The *history* approach allows at least one simple heuristic to be used as a replacement policy. A comparison of the performance of this heuristic to the performance of strings found using the genetic algorithm approach is covered in Section 5.4.2.

3. The *history* approach allows many variants. Some were mentioned briefly in Section 5.2.2. Section 5.4.3 compares the performance of two variants. In one case the history includes a record of hits and misses as well as lines referenced. In the other case only the lines referenced are recorded.

5.4.1 Canonical Form

As mentioned in Sections 4.1 and 4.4, the *history* method uses a more natural representation for strings than that described in Chapter 4. Use of this natural representation changes the performance of the genetic algorithm, sometimes for better, sometimes for worse. This is discussed in more detail later.

The key to this canonical representation is to note tha. many history sequences are equivalent. For example, assume in the previous 4 accesses that the history of lines accessed was 1,3,3,1. For the algorithm this is really no different than if the lines accessed had been 2,4,4,2 or 2,3,3,2, or many others.

Realizing this, it makes sense to reduce all equivalent patterns to the same pattern and let the algorithm work with the canonical form. It turns out that the compression ratio, $\frac{Total\ Strings}{Number\ of\ Canonical\ Strings}$ is

$$\lim_{m \to \infty} Compression\ Ratio = K!$$

See Appendix A.2 for a proof.

Several compression ratios are given in Table 5.14. In particular, note that for the case used in these simulations, associativity $K = 4$ and $m = 4$ previous accesses, the compression ratio is 17.07. This means that instead of needing strategy strings of 512 bits, strings of only 30 bits are required!.

The actual number of canonical forms, $f(K, m)$ is given by

$$f(K,m) = \sum_{r=1}^{K} \left[\left(\frac{r^m}{r!} \right) \left(\sum_{c=0}^{K-r} \frac{(-1)^c}{c!} \right) \right] \tag{5.1}$$

This equation can be derived using the theory of group actions on a set as described in [35]. Assume an alphabet of K letters and words of length m. Then two words are said to be in the same *orbit* if and only if one word can be obtained from the other by a permutation of letters. An *orbit* corresponds to the notion of a canonical form above. The goal is then

K	m	Canonical Forms	Total Forms	Compression Ratio	$\frac{m+1}{m}$ Ratio
1	n	1	1	1.00	1.00
2	1	1	2	2	
2	2	2	4	2	2
2	3	4	8	2	2
2	4	8	16	2	2
2	5	16	32	2	2
2	6	32	64	2	2
2	7	64	128	2	2
2	8	128	256	2	2
2	9	256	512	2	2
2	10	512	1024	2	2
2	∞			2!=2	2
3	1	1	3	3	
3	2	2	9	4 50	2.00
3	3	5	27	5.40	2.50
3	4	14	81	5.79	2.80
3	5	41	243	5 93	2 93
3	6	122	729	5.98	2.98
3	7	365	2187	5.99	2.99
3	8	1094	6561	5.997	2.997
3	9	3281	19683	5.999	2.999
3	10	9842	59049	5.9997	2.9997
3	∞			3!=6	3
4	1	1	4	4	
4	2	2	16	8 00	2.00
4	3	5	64	12.80	2.50
4	4	15	256	17.07	3.00
4	5	51	1024	20.08	3.40
4	6	187	4096	21.90	3.67
4	7	715	16384	22.91	3.82
4	8	2795	65536	23.45	3.91
4	9	11051	262144	23.72	3.95
4	10	43947	1048576	23.86	3.98
4	∞			4!=24	4

Table 5.14: Number of Canonical Forms for History Replacement.

to count the number of different orbits for arbitrary K and m. See Appendix A.1 for a proof.

As an aside, note that the second term in Equation 5.1 is related to e:

$$\lim_{(K-r)\to\infty} \sum_{c=0}^{K-r} \frac{(-1)^c}{c!} = \frac{1}{e}$$

That the number of canonical forms is related to e, is a surprising and beautiful result.

As noted in Section 4.4 this small number of canonical forms does not help in reducing the hardware complexity of the history approach. To use this "canonical" method, the cache would require combinational logic to reduce the history to canonical form, determine which line to replace based on the canonical form, and map the answer back to the original form. It is almost certainly easier to have hardware logic calculate the replacement line directly from the original history.

Finally it is interesting to note in Table 5.14 for a given K, that the ratio of the number of canonical forms of length m to the number of length $m+1$ increases by a factor of K as $m \to \infty$. In other words

$$\lim_{m\to\infty} \frac{Number\ of\ Canonical\ Forms(m+1)}{Number\ of\ Canonical\ Forms(m)} = K$$

See Appendix A.3 for a proof of this.

Table 5.15 provides some results of using the canonical form for *history*. The Table compares the *line hit rates* obtained using the canonical form and the non-canonical form. In both cases, the simulations use the standard parameters described in Sections 5.1 and 5.2.1 were used. (The *line hit rates* for the canonical form correspond to the simulations described earlier in Section 5.2.2).

Table 5.15 indicates that the non-canonical representation is the superior choice for instruction caches, while the canonical representation is best for data caches. That the non-canonical form is better for instruction caches is somewhat surprising. The motivation for developing the canonical form was the equivalency of different histories. Why should a

	INSTRUCTION		DATA	
	Canonical	Non-Canonical	Canonical	Non-Canonical
CCAL	29.8	31.9	67.3	65.1
EMACS	23.6	25.1	70.3	69.4
KALMAN	31.6	33.1	61.5	59.7
MCSH	55.2	58.8	82.8	80 8
POLY	30.2	32.3	56 3	54.8
WHETSTONE	37.0	37.0	37.5	37.2
Mean	34.6	36.4	62 6	61 2
SUITE	42.6	45.3	73.2	71 4

Table 5.15: Line Hit Rates using Canonical and Non-canonical Representation 512 byte Instruction and Data caches with 4-way associativity, 32 byte lines.

representation which allows multiple forms of supposedly equivalent histories ever perform better than a representation which has only one form for each equivalent set of histories? The canonical form has a much smaller solution space to explore, and chooses replacement lines consistently.

If the line history is $0,3,1,2$ and a string of the canonical form chooses to replace line 1, then if the line history is $2,1,0,3$, the string will choose line 0. On the other hand, a string using a non-canonical representation could replace line 1 in the first case and line 3 in the second.

One possible explanation for the superior performance of the non-canonical form is that supposedly *equivalent* histories are not actually equivalent. It could be that the genetic algorithm is sometimes able to develop a replacement policy which dedicates certain cache lines to certain types of values. For example, from the history of accesses, the algorithm may sometimes be able to detect that a given line corresponds to an innermost nested loop, while another corresponds to transient initialization code. In such a case, the cache may want to manage the *inner loop* line differently than the *initialization* line. To this end, the cache might try to reserve line 0 for inner loop lines, and line 3 for initialization lines. In

such a case, the meaning of $0,3,1,2$ would indeed be different than $2,1,0,3$.

This reasoning could also account for the difference between instruction and data caches. The history of accesses to an instruction cache may be *simple* enough that the genetic algorithm can determine a need for specialized lines as just described. For data caches, the history may be more complicated, not allowing the genetic algorithm to dedicate lines for special purposes. In this case, the consistency and reduced complexity of the canonical approach could be important to finding good solutions.

There is another possible explanation for the superiority of the non-canonical form in instruction caches. Certain sets in the cache may tend to use different physical lines intensively. The non-canonical representation could then be optimized so that different sets essentially have different replacement policies. However this explanation is sufficient only for individual benchmarks. Presumably in the multitasking suite of benchmarks the varied use of sets by the different benchmarks makes this optimization difficult. Since the non-canonical form is better even for multitasking (45.3% to 42.6% *line hit rate*), this explanation is inadequate.

Other explanations undoubtedly exist. More work is needed to determine the precise causes of the sometimes superiority of the non-canonical form.

Finally not only is the non-canonical form better than the canonical form of *history* in instruction caches, it is better than every other approach in Section 5.2.2 as well. The mean *line hit rate* for the individual benchmarks using *LRU-History* is 36.1%. The mean for the non-canonical form is 36.4%. The multitasking performance of the non-canonical form is slightly inferior to *LRU-History* (45.7% versus 45.3% *line hit rate*), but is superior to every other approach tried in Section 5.2.3.

5.4.2 Genetic Algorithms versus Least Recent History

Given the general success of LRU as a replacement policy, it is natural to make the *history* method attempt to mimic LRU. This can be done by always replacing the oldest line referenced. If some lines have not been accessed, then the line to replace is chosen arbitrarily from among those not accessed.

	LRU	GA History, $m = 4$	LRH, $m = 4$	LRH, $m = 8$
CCAL	29.4	29.8	27.6	28.8
EMACS	22.6	23 6	20 4	22 9
KALMAN	30.8	31.6	28 7	32 9
MCSH	56.6	55 2	48 2	55.9
POLY	29.6	30.2	26.9	30.7
WHETSTONE	30.1	37.0	29.6	35 1
Mean	33.2	34.6	30.2	34.7
SUITE	43.9	42 6	38 3	43 7

Table 5.16: Line Hit Rates for *Least Recent History* and other Replacement Policies 512 byte I-cache with 4-way associativity, 32 byte lines.

Note that for associativity, K, only $K-1$ distinct lines need be accessed to fully simulate LRU. Also note that to approximate LRU, a history need only record which lines have been accessed in a set. Any *hit/miss* information is extra and not used by LRU

The performance of this *Least Recent History* or *LRH* approach varies. For an *instruction* cache and a standard $m = 4$ deep history, *LRH* performs significantly worse than LRU. As is indicated in Table 5.16, the mean *line hit rate* for LRU is 33 2%, but only 30.2% for *LRH*.

The poor performance of *LRH* in this case accentuates the value of the genetic algorithm approach used in Sections 5.2.2 and 5.2.3 For the same $m = 4$ deep history, the genetic algorithm approach is able, for each benchmark individually and for the suite of benchmarks, to find replacement strategies which do far better than *LRH* The mean *line hit rate* for the genetic algorithm strategies is 34.6% versus 30.2% for *LRH*. It is worth emphasizing that *LRH* appears a plausible heuristic for a replacement policy However, by efficiently exploring a wider portion of the solution space, the genetic algorithm approach does far better.

The depth of history appears to be an important factor in obtaining good performance in instruction caches. If the depth is increased from the previous $m = 4$ accesses to the

	LRU	GA History, $m = 4$	LRH, $m = 4$	LRH, $m = 8$
CCAL	67.7	67.3	65.7	65.4
EMACS	69 3	70.3	69.1	68.3
KALMAN	61.3	61.5	60.3	60.7
MCSH	83.3	82 8	82 2	82.6
POLY	56.4	56.3	55.4	54.3
WHETSTONE	36.3	37.5	35.5	34.7
Mean	62.4	62.6	61.4	61.0
SUITE	73.7	73.2	72.5	72.7

Table 5.17: Line Hit Rates for *Least Recent History* and other Replacement Policies. 512 by'e D-cache with 4-way associativity, 32 byte lines.

set to $m = 8$, the mean *line hit rate* jumps from 30.2% to 34.7%. A reason for this leap is suggested in Table 5.18. When $m = 4$, the history on average contains 3 or 4 distinct lines only 3% of the time. For $m = 8$ this rises to 26%. With the shorter history, there is apparently insufficient information on which to make a judgment much of the time.

Note that with K-way associativity, the history can contain at most K distinct lines. Recall from Chapter 4 that the history records which of the K lines in a *cache set* have been accessed, as opposed to which memory lines Hence whether $m = 4$ or $m = 8$, the maximum number of distinct lines is 4, since $K = 4$.

The behavior of *LRH* in data caches is quite different:

- As can be seen in Table 5.17, the mean *line hit rate* of *LRH* with $m = 4$ (61.4%) is reasonably close to LRU (62.4%) and the genetic algorithm strategies (62.6%). A likely reason for this is the larger number of distinct lines accessed in data caches than instruction caches. Table 5.19 indicates that on average the history contains 3 or 4 distinct lines on 30% of data cache misses. Even for an instruction cache with $m = 8$, the number is only 26%. Hence a data cache employing *LRH* has more information on which to make a decision than an instruction cache.

	Lines Accessed							
	$m = 4$				$m = 8$			
	1	2	3	4	1	2	3	4
CCAL	65	33	2	0	22	48	26	4
EMACS	66	33	1	0	30	49	18	3
KALMAN	58	38	4	0	23	46	27	4
MCSH	76	23	1	0	27	58	13	2
POLY	61	35	5	0	23	49	24	4
WHETSTONE	54	44	3	0	21	52	22	5
Mean	63	34	3	0	24	50	22	4
SUITE	68	30	2	0	26	52	19	3

Table 5.18: Percentage Distribution of Distinct Lines Accessed in Set When Miss Occurs using LRH replacement. 512 byte I-Cache with 4-way associativity, 32 byte lines.

	Lines Accessed							
	$m = 4$				$m = 8$			
	1	2	3	4	1	2	3	4
CCAL	33	38	25	4	15	29	35	21
EMACS	68	18	12	2	28	44	18	10
KALMAN	33	29	33	6	14	28	32	27
MCSH	41	39	17	3	15	33	35	17
POLY	43	26	26	5	17	34	27	22
WHETSTONE	35	18	37	10	26	14	29	32
Mean	42	28	25	5	19	30	29	22
SUITE	44	31	22	4	19	33	30	19

Table 5.19: Percentage Distribution of Distinct Lines Accessed in Set When Miss Occurs using LRH replacement. 512 byte D-Cache with 4-way associativity, 32 byte lines.

- The mean *line hit rate* actually goes *down* from 61.4% to 61.0% when the depth is increased from $m = 4$ to $m = 8$. As just indicated, this could be partially because an $m = 4$ deep history has sufficient information on which to make a decision.

  Furthermore, there is an increased likelihood that a line is dead if it has not been accessed in the previous 4 accesses to a set. Even if not dead, recency of usage may not be the best replacement metric for "old" lines. If several lines have not been accessed in the previous 4 accesses to a set, the best policy can be to randomly replace a line from among them, as LRH with an $m = 4$ does. This possibility is in accordance with [39], where it was found that for instruction caches, random replacement can sometimes be superior to LRU.

5.4.3 Two History Variants

The *history* approach used in Sections 5.2.2 and 5.2.3 maintained a record only of lines accessed. The *history* did not include any *hit/miss* information. This Section examines the effect of this choice.

Table 5.20 compares *line hit rates* for the two history variants. As has often been the case, results differ dramatically for instruction and data caches. For instruction caches, the mean *line hit rate* improves from 34.6% to 36.0% when *hit/miss* information is included in the history. Each benchmark and the suite of benchmarks shows a significant improvement.

The 36.0% mean *line hit rate* for the *individual* benchmarks compares favorably to the 36.1% of *LRU-History*, the best method found in Section 5.2.2. Thus *history* with a *hit/miss* record may be suitable for an instruction cache in which the replacement policy is stored in a RAM and is modifiable.

The 43.8% *line hit rate* for the suite of benchmarks (simulating multitasking), however is still worse than every other approach tried in Section 5.2.3, except FIFO (43.2%). Thus neither *history* variant seems appropriate for an instruction cache in which the replacement policy is hard-wired into the cache.

As can be seen in Table 5.20, the mean performance in data caches actually declines when *hit/miss* information is included in the history! Clearly the genetic algorithm approach

	INSTRUCTION		DATA	
	Lines Only	Lines & Hits/Misses	Lines Only	Lines & Hits/Misses
CCAL	29.8	30.6	67.3	66.5
EMACS	23.6	25.7	70.3	70.4
KALMAN	31.6	33.4	61.5	61.5
MCSH	55.2	56.4	82.8	81.8
POLY	30.2	31.4	56.3	55.6
WHETSTONE	37.0	38.2	37.5	37.5
Mean	34.6	36.0	62.6	62.2
SUITE	42.6	43.8	73.2	72.1

Table 5.20: Line Hit Rates for *History* of Line References and for *History* of Line References *and* Hits/Misses. 512 byte Instruction and Data Caches with 4-way associativity, 32 byte lines.

is a bit deficient here. Since the *hit/miss* record is additional information, it is always possible to find at least as good a solution by ignoring the additional information.

Since performance improves significantly for instruction caches, it seems likely that *hit/miss* information is a useful for instruction caches, but generally not so for data caches, and merely confuses the genetic algorithm. This is corroborated by the fact that for instruction caches, *LRU-History*, i.e. LRU augmented with *hit/miss* information, is significantly better than the other methods used in Sections 5.2.2 and 5.2.3. However for data caches the performance of *LRU-History* is comparable to other approaches such as *LRU-Count*. Some reasons for this were given on page 56.

Finally, a history depth of $m = 4$ is used in Table 5.20, as it is throughout this document. Note that different depths (m_1 and m_2) of history could be kept for *hits/misses* and *lines referenced*. Limited trials were made using different m_1 and m_2, but results did not appear to differ significantly from those reported in Table 5.20. Still this might make an interesting area for further inquiry.

5.5 Shadow Cache

Shadow cache [33] is a hybrid replacement policy somewhat similar to those described here. The shadow cache method augments a normal cache using LRU replacement. The augmentation is for each set to keep a larger number of tags (see Chapter 2), than actual lines. For example with a $K = 4$-way associative cache, each set would normally keep 4 lines and their corresponding 4 tags. The shadow cache augments the number of tags kept, perhaps keeping 4 additional tags in a *shadow directory*. The cache is then managed as an 8-way associative cache.

By keeping the extra tags, the cache can "distinguish between *transient* lines that must be flushed from cache quickly and lines that become active after long periods of inactivity [*shadow misses*]...As each new item is loaded into cache, a bit is set to indicate whether the item was a transient miss or a shadow miss...[The cache manager] can tend to retain the lines that were in the shadow [directory] in favor of the lines that were transient misses, and in this way it will tend to flush transients from the cache more quickly than an LRU algorithm will flush them [41]."

Thus the shadow cache augments normal LRU information with additional historical information about what lines have been accessed. This is quite similar to some of the methods used here, such as *LRU-History*, where LRU information is augmented by information about the history of hits and misses to the set.

Table 5.21 compares *line hit rates* obtained using *LRU-History* to a slightly more effective variant of shadow cache suggested by Puzak [34]. Under this variant, non-LRU decisions are limited to the LRU and next-to-LRU entries. Puzak found that using the shadow replacement policy on the MRU and next-to-MRU entries reduced performance by unwisely flushing lines brought in on transient misses.

As can be seen in Table 5.21, the *LRU-History* method fares very well versus shadow cache. In every case except a data cache with a multitasking workload, *LRU-History* performs better than the shadow cache. In that case, the performance is equal.

LRU-History's superiority is especially marked for instruction caches. The mean *line hit rate* of *LRU-History* is 36.1% versus only 33.5% for the shadow cache. The difference is even

	INSTRUCTION			DATA		
	LRU	LRU-History	Shadow	LRU	LRU-History	Shadow
CCAL	29.4	30.7	30.4	67.7	67.8	67.4
EMACS	22.6	24.6	23.3	69.3	70.0	69.9
KALMAN	30.8	33.4	31.4	61.3	62.0	61.0
MCSH	56.6	59.7	56.8	82.3	83.5	83.7
POLY	29.6	31.9	28.6	56.4	56.7	56.4
WHETSTONE	30.1	36.3	30.6	36.3	38.6	36.3
Mean	33.2	36.1	33.5	62.4	63.1	62.5
SUITE	43.9	45.7	41.6	73.7	73.9	73.9

Table 5.21: Line Hit Rates for LRU, LRU-History, and Shadow Caches. 512 byte Instruction and Data caches with 4-way associativity, 32 byte lines.

greater when multitasking is simulated: 45.7% versus 41.6%. Furthermore *LRU-History* is superior to LRU in every case, while in 4 cases the shadow cache actually performs worse than LRU.

5.6 Optimization by Set

If a cache stores its replacement policy in RAM, then it is may be reasonable for each set in the cache to have its own RAM and consequently its own replacement policy. In this way, the replacement policy can truly be tailored to a specific program, subroutine, loop, etc..

To test the potential of such an approach, *LRU-History* was used to find a good replacement policy for each individual set for each benchmark. The results of this approach are shown in Table 5.22. They show a significant improvement over the performance obtained when the entire cache uses the same replacement policy.

	INSTRUCTION				DATA			
	Line Hit Rate		LRU-OPT Gain		Line Hit Rate		LRU-OPT Gain	
	Set	Cache	Set	Cache	Set	Cache	Set	Cache
CCAL	32.3	30.7	21	10	68.6	67.8	11	1
EMACS	26.6	24.6	29	14	70.4	70.0	19	12
KALMAN	35.1	33.4	27	17	63.2	62.0	22	9
MCSH	60.4	59.7	21	17	83.7	83.5	11	5
POLY	33.0	31.9	23	15	57.3	56.7	12	3
WHETSTONE	38.5	36.3	44	33	40.1	38.6	43	25
Mean	37.6	36.1	27	18	63.9	63.1	20	9

Table 5.22: Performance of *LRU-History* with Common Cache Replacement Policy and with Individual Set Replacement Policies. 512 byte Instruction and Data caches with 4-way associativity, 32 byte lines.

In Section 5.2.2, *LRU-History* was the best method overall. Nevertheless the mean *LRU-OPT gain* of the *line hit rate* improves from 18% to 27% for instruction caches and more than doubles from 9% to 20% for data caches. In other words, instruction caches obtain more than a quarter of the possible gain in *line hit rate* from LRU to OPT, while data caches obtain one fifth.

The overall hit rates show similar improvement. For LRU, the mean overall hit rate for instruction caches is 91.0%, while OPT achieves 93.2%. The value for *LRU-History by Set* is 91.6%, i.e. 27% of the 2.2% difference between LRU and OPT is bridged. For data caches, the corresponding hit rates are

- 88.6% LRU
- 90.9% OPT
- 89.1% *LRU-History by Set*

Here 20% of the 2.3% difference is bridged.

Luckily, the cost of implementing this approach is relatively low. As outlined in Section 4.4, each set requires 2, 4-bit RAM's to specify the LRU rank of the line to be replaced. In other words, 1 byte per set of additional storage is sufficient to implement *LRU-History by Set.*

Chapter 6

Conclusion

Harold Stone has stated, "We are likely to gain only from ten to 30 percent of the available improvement because the hardware cannot have perfect knowledge of the future [41]." The results found here bear this out:

1. Improvement was made over traditional replacement policies. When a separate replacement policy was used for each benchmark, *LRU-History* was able to bridge 18% of the 2.2% mean difference in hit rates between LRU and OPT in an instruction cache and 9% of the 2.3% mean difference in a data cache.

2. Use of a different *LRU-History* replacement policy for each benchmark *and* for each set allowed 27% of the difference to be bridged in an instruction cache and 20% in a data cache.

3. Smaller improvements were made when a common replacement policy was used for all benchmarks while simulating multitasking. *LRU-History* was able to bridge 9% of the 2.3% difference between LRU and OPT in an instruction cache and 2% of the 1.7% difference in a data cache.

4. It was found that LRU and FIFO replace the same line as OPT on approximately 5% of misses. Using genetic algorithms, replacement policies were found for each method (*LRU-History*, *LRU-Count*, etc.) which raised this to approximately 30% for

instruction caches and 18% for data caches. However, the resulting hit rates were generally lower than those of LRU or FIFO.

5. Performance of *LRU-History* was significantly better than *shadow cache*, another approach which augments LRU information with historical information.

6. Performance of replacement policies found using the genetic algorithm approach was in all cases equal or better to the performance of policies generated randomly.

7. Representation of strings had a significant effect on the quality of replacement policy found by the genetic algorithm.

8. The *history* approach allows both heuristic replacement policies such as LRH, and replacement policies found by applying genetic algorithms. The genetic algorithm policies substantially outperformed the LRH heuristic. In many cases *history* also performed as well or better than LRU.

9. Allowing the cache not to bring in newly referenced lines was found to be detrimental, even in the most suitable benchmarks.

The primary goal of using genetic algorithm techniques to improve upon existing cache replacement policies, has been realized. Since improvement was better in instruction caches, any future work might best be concentrated there.

There are a number of avenues for future work:

1. More sophisticated genetic techniques could be tried, for example recession and dominance, niche operators, and crossover techniques which permit bit function to be independent of position.

2. Larger caches and longer address traces could be used. Of particular interest are benchmarks with a large absolute difference in hit rates between LRU and OPT.

3. The difference in performance between *canonical* and *non-canonical* forms of history could be investigated further.

4. Maximizing the fraction of time a replacement policy chose the same line as OPT to replace did not improve the overall hit rate. Perhaps maximizing the fraction of time that the OPT replacement line is *available* would yield better results.

5. The most ambitious project would be to actually implement a system using one of the methods described here. An instruction cache could be built using a hard-coded *LRU-History* replacement policy. Alternatively, a cache could be built with a RAM based replacement policy. An optimizing compiler could then be enhanced to use genetic algorithm simulations to find good replacement policies.

Appendix A

Proofs

A.1 Derivation of Number of Orbits

Assume an *alphabet* of K characters and *words* of length m. Then two words are said to be in the same *orbit* if and only if one word can be obtained from the other by a permutation of letters in the alpnabet [1]. An *orbit* corresponds to the notion of a canonical form in Section 5.4.1. The goal is then to count the number of different orbits for arbitrary K and m.

To this end, start with two more definitions:

W_m = All words in K letters of length m. Note $|W_m| = K^m$.

S_K = Group of permutations on K letters acting on W_m. Note $|S_K| = K!$.

Burnside's Formula then gives the number of orbits [35]:

$$f(K,m) = \#\ of\ orbits = \frac{1}{|S_K|} \sum_{\sigma \epsilon S_K} \#\ of\ forms\ fixed\ by\ \sigma$$

$$= \frac{1}{K!} \sum_{\sigma \epsilon S_K} \#\ of\ forms\ fixed\ by\ \sigma$$

[1] Many thanks to Sheila Sundaram for help in constructing this proof.

Since $\sigma \epsilon S_K$, what words (or canonical forms)

$$w = w_1 w_2 \ldots w_m$$

are fixed by σ?

$$\begin{aligned} \sigma(w) &= \sigma(w_1)\sigma(w_2)\ldots\sigma(w_m) \\ &= w_1 w_2 \ldots w_m \end{aligned}$$

So w is fixed by $\sigma \Leftrightarrow \sigma(w_i) = w_i, \ \forall i = 1, \ldots, m.$

So if σ has r fixed points, there are r^m words fixed by σ. Clearly r can vary from 0 to the size of the alphabet, K. Thus Burnside's Formula can be rewritten:

$$f(K,m) = \frac{1}{K!}\sum_{r=0}^{K} \#\{\textit{permutations with } r \textit{ fixed points}\} \times r^m$$

Now another definition is needed. A *derangement* is a permutation with no fixed point. Let $d(n)$ be the number of derangements on n points. It is well known [40], that

$$d(n) = n!\sum_{c=0}^{n} \frac{(-1)^c}{c!}$$

There are $\binom{K}{r}$ ways to choose the r fixed points. For each of these ways, none of the other $K - r$ points are fixed. The number of permutations with none of the other $K - r$ points fixed is just the number of derangements $d(K - r)$. Hence Burnside's Formula can now be re-expressed as

$$f(K,m) = \frac{1}{K!}\sum_{r=0}^{K} r^m \binom{K}{r} d(K-r) \qquad \text{(A.1)}$$

$$= \sum_{r=1}^{K} \frac{r^m}{r!}\,\frac{d(K-r)}{(K-r)!} \qquad \text{(A.2)}$$

Finally substituting formula for the number of *derangements* gives

$$f(K,m) = \sum_{r=1}^{K} \left[\left(\frac{r^m}{r!} \right) \left(\sum_{c=0}^{K-r} \frac{(-1)^c}{c!} \right) \right]$$

which is the same formula given in in Equation 5 1.

A.2 Proof of Compression Ratio

The compression ratio follows from Equation A.2 and the fact that with an alphabet of K characters and words of length m, there are K^m total words.

$$\begin{aligned}
\frac{1}{Compression\ Ratio} &= \frac{f(K,m)}{K^m} \\
&= \sum_{r=1}^{K} \frac{1}{r!} \left(\frac{r}{K} \right)^m \frac{d(K-r)}{(K-r)!} \\
&= \left[\sum_{r=1}^{K-1} \frac{1}{r!} \left(\frac{r}{K} \right)^m \frac{d(K-r)}{(K-r)!} \right] + \frac{1}{K!} \left(\frac{K}{K} \right)^m \frac{d(0)}{(0)!} \\
&= \left[\sum_{r=1}^{K-1} \frac{1}{r!} \left(\frac{r}{K} \right)^m \frac{d(K-r)}{(K-r)!} \right] + \frac{1}{K!}
\end{aligned}$$

Note in the sum that $r < K$. Thus

$$\lim_{m \to \infty} \left(\frac{r}{K} \right)^m = 0$$

This makes the summation 0, and hence the compression ratio is $K!$.

A.3 Proof of Increase in Canonical Forms

As $m \to \infty$, there are K times as many canonical forms in a string of length m as in a string of length $m-1$. The approach used to show this is similar to that used above in Section A.2 to show that the compression ratio is $K!$ and also begins with Equation A.2.

$$\begin{aligned} \textit{Increase Factor} &= \frac{f(K,m)}{f(K,m-1)} \\ &= \frac{\sum_{r=1}^{K} \frac{r^m}{r!} \frac{d(K-r)}{(K-r)!}}{\sum_{r=1}^{K} \frac{r^{m-1}}{r!} \frac{d(K-r)}{(K-r)!}} \\ &= \frac{\left[\sum_{r=1}^{K-1} \frac{r^{m-1}}{(r-1)!} \frac{d(K-r)}{(K-r)!}\right] + \frac{K^{m-1}}{(K-1)!} \frac{d(0)}{0!}}{\left[\sum_{r=1}^{K-1} \frac{r^{m-2}}{(r-1)!} \frac{d(K-r)}{(K-r)!}\right] + \frac{K^{m-2}}{(K-1)!} \frac{d(0)}{0!}} \end{aligned}$$

Dividing both the numerator and denominator by K^{m-1} yields

$$\textit{Increase Factor} = \frac{\left[\sum_{r=1}^{K-1} \frac{1}{(r-1)!} \left(\frac{r}{K}\right)^{m-1} \frac{d(K-r)}{(K-r)!}\right] + \frac{1}{(K-1)!} \left(\frac{K}{K}\right)^{m-1}}{\frac{1}{K}\left\{\left[\sum_{r=1}^{K-1} \frac{1}{(r-1)!} \left(\frac{r}{K}\right)^{m-2} \frac{d(K-r)}{(K-r)!}\right] + \frac{1}{(K-1)!} \left(\frac{K}{K}\right)^{m-2}\right\}}$$

As before in $r < K$ in both summations so

$$\lim_{m \to \infty} \left(\frac{r}{K}\right)^m = 0$$

This makes the summations 0 in both numerator and denominator. The K-th terms are both $\frac{1}{(K-1)!}$ and cancel. This leaves only

$$\textit{Increase Factor} = \frac{1}{\frac{1}{K}} = K$$

Bibliography

[1] A. Agarwal, J. Hennessy, and Horowitz M. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] A. Agarwal, R.L. Sites, and Horowitz M. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986.

[3] C. Alexander, W. Keshlear, F. Cooper, and F. Briggs. Cache Memory Performance in a UNIX Environment. *Computer Architecture News*, pages 41–70, June 1986.

[4] E.R. Altman. An Analysis of Architect's Workbench. ACAPS Technical Note 19, School of Computer Science, McGill University, Montreal, Que., March 1990.

[5] R. Axelrod. The Evolution of Strategies in the Iterated Prisoner's Dilemma. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 32–41. Pitman, 1987.

[6] L.A. Belady. A Study of Replacement Algorithms for a Virtual-Store Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[7] L.A. Belady and F.P. Palermo. On-line Measurement of Paging Behaviour by the Multivalued MIN Algorithm. *IBM Journal of Research and Development*, pages 2–19, January 1974.

[8] R.M. Brady. Optimization Strategies Gleaned from Biological Evolution. *Nature*, 817:804–806, November 1985. (Letter to the Editor).

[9] A. Brindle. *Genetic Algorithms for Function Optimization.* PhD thesis, University of Alberta, Edmonton, 1981.

[10] D.J. Cavicchio. *Adaptive Search Using Simulated Evolution.* PhD thesis, University of Michigan, Ann Arbor, 1970.

[11] J.H. Crawford. The i486 CPU: Executing Instructions in One Clock Cycle. *IEEE Micro*, 10(1):27–36, February 1990.

[12] L. Davis. Job Shop Scheduling with Genetic Algorithms. In *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, pages 136–140, 1985.

[13] L. Davis and D. Smith. Adaptive Design for Layout Synthesis. Technical report, Texas Instruments, Dallas, 1985.

[14] K.A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, Ann Arbor, 1975.

[15] R.W. Edenfield, M.G. Gallup, W.B. Ledbetter, R.C. McGarity, E.E. Quintana, and R.A. Reininger. The 68040 Processor: Part 1, Design and Implementation. *IEEE Micro*, pages 66–78, February 1990.

[16] R.J. Eickemeyer and J.H. Patel. Performance Evaluation of On-chip Register and Cache Organizations. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 64–72. IEEE, May 30 to June 2, 1988.

[17] D.M. Etter, M.J. Hicks, and K.H. Cho. Recursive Adaptive Filter Design Using an Adaptive Genetic Algorithm. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing.* IEEE, 1982.

[18] S. Forest. *Documentation for PRISONERS DILEMMA and NORMS Programs That Use the Genetic Algorithm.* University of Michigan, Ann Arbor, 1985.

[19] D.R. Frantz. *Non-linearities in Genetic Adaptive Search.* PhD thesis, University of Michigan, Ann Arbor, 1983.

[20] D.E. Goldberg. *Computer-Aided Gas Pipeline Operation Using Genetic Algorithms and Rule Learning.* PhD thesis, University of Michigan, Ann Arbor, 1983.

[21] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[22] D.E. Goldberg and R. Lingle. Alleles, Loci, and the Traveling Salesman Problem. In *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, pages 154–159, 1985.

[23] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffmann, 1990.

[24] J.H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975.

[25] W. Hollingsworth, H. Sachs, and A.J. Smith. The Clipper Processor: Instruction Set Architecture and Implementation. *Communications of the ACM*, pages 200–219, February 1989.

[26] A.C. Klaiber and H.M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53. IEEE, May 27-30 1991.

[27] R.F. Krick and A. Dollas. The Evolution of Instruction Sequencing. *Computer*, pages 5–15, April 1991.

[28] W. Mangione-Smith, S.G. Abraham, and E.S. Davidson. A Performance Comparison of the IBM RS/6000 and the Astronautics ZS-1. *IEEE Computer*, pages 39–46, January 1991.

[29] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[30] B. Maytal, S. Iacobovici, D.B. Alpert, D. Biran, J. Levy, and S.Y. Tov. Design Considerations for a General Purpose Microprocessor. *IEEE Computer*, pages 66–76, January 1989.

[31] J. Miyake, T. Maeda, Y. Nishimichi, J. Katsura, T. Taniguchi, S. Yamaguchi, H. Edamatsu, S. Watari, Y. Takagi, K. Tsuji, S. Kuninobu, S. Cox, D. Duschatko,

and D. MacGregor. A Highly Integrated 40-MIPS (Peak) 64-b RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 25(5):1199–1206, October 1990.

[32] R. Olsen. Instructions for Performing Trace-Driven Cache Simulations. ACAPS Technical Note 20, School of Computer Science, McGill University, Montreal, Que., 1990.

[33] J. Pomerene, T.R. Puzak, R. Rechtschaffen, and F. Sparacio. *Prefetching Mechanism for a High-Speed Buffer Store*, 1984. Patent Pending.

[34] T.R. Puzak. *Cache-Memory Design.* PhD thesis, University of Massachusetts, 1985.

[35] J.J Rotman. *Theory of Groups.* Allyn and Bacon, 1973.

[36] A.J. Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[37] A.J. Smith. Cache Evaluation and the Impact of Workload Choice. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 64–73. IEEE, 1985.

[38] A.J. Smith. Line (Block) Size Choices for CPU Cache Memories. *IEEE Transactions on Computers*, pages 1063–1075, September 1987.

[39] J.E. Smith and J.R. Goodman. Instruction Cache Replacement Policies and Organizations. *IEEE Transactions on Computers*, 34(3):234–241, March 1985.

[40] R.P. Stanley. *Enumerative Combinatorics*, volume 1. Wadsworth and Brooks/Cole Advanced Books and Software, 1986.

[41] H.S. Stone. *High Performance Computer Architecture.* Addison-Wesley, 1987.

[42] J.Y. Suh and D. Van Gucht. Incorporating Heuristic Information into Genetic Search. In *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 100–107, 1987.