McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

Genetic Algorithms and Cache Replacement Policy

Erik R. Altman†
Vinod K. Agarwal†
Guang R. Gao

ACAPS Technical Memo 55
September 11, 1992

†Dept. of Electrical Enginering, McGill University, 3480 University Street, Montréal, Québec

Abstract

This paper proposes a novel methodology for cache replacement policy based on techniques of *genetic algorithms*. The key innovation is to allow the cache to be tuned to a given workload by using an appropriate combination of LRU, FIFO, or other information such as a history of hits and misses in a set or the number of hits to each line. By employing genetic algorithms the search for the appropriate combination can be performed effectively and efficiently. In order to allow tuning of individual benchmarks, we propose the use of programmable replacement policies.

In this paper, we use an optimal replacement policy, known as OPT. OPT gives the theoretical limit of any replacement policy which does not prefetch. OPT has been found to provide a miss rate about 70% that of LRU for the same size and configuration of cache [20]. Our objective is to reduce the large *LRU-OPT gap* in miss rate and we use the *LRU-OPT gain* — the fraction of the *LRU-OPT gap* we eliminate — to measure our improvement.

The main results of this paper are: 1) A methodology is proposed which employs genetic algorithms to efficiently search for good replacement policies for a desired workload. 2) An experimental study has been performed on a range of benchmark programs. Our method yields policies which are capable of *LRU-OPT gains* of 20%-30% or more. 3) Four tests have been performed which show that our use of genetic algorithms is robust and effective. 4) Our method can be implemented with a hardware complexity only slightly greater than that of traditional policies.

Contents

1 Introduction **1**

2 Genetic Algorithms **3**

3 Cache Replacement Families **5**
3.1 Types of Families and Applying the Genetic Algorithm 5
3.2 Hardware 8

4 Results **8**
4.1 Methodology 8
4.2 Comparison of Replacement Families 9
4.3 Performance on Larger Systems 12

5 Robustness and Comparative Performance **14**
5.1 Robustness of Genetic Algorithm Approach 14
5.2 Comparison of our Results to Other Work 15

6 Conclusion **18**

Bibliography **20**

List of Figures

1 Crossover for two pairs of parents 4
2 Block diagram of hardware for 4-bit *LRU-History Family* (Standard LRU Hardware not Shown) 7
3 Improvement in Best Policy by Generation 10
4 Differences in Miss Rate with FIFO for 512-byte Caches 12
5 Differences in Miss Rate with FIFO for 8-kbyte Caches 13
6 Exhaustive Search of 3-bit LRU-History Family, 512-byte Data Cache, Whetstone Benchmark 15
7 Comparison of GODS, *LHBS*, and OPT for 8-kbyte Instruction Cache 16
8 Comparison of GODS, *LHBS*, and OPT for 8-kbyte Data Cache 17
9 Comparison of Shadow Cache, *LHBS*, and OPT for 8-kbyte Instruction Cache . . 17
10 Comparison of Shadow Cache, *LHBS*, and OPT for 8-kbyte Data Cache 17

List of Tables

1 Mean *LRU-OPT Gains* for 6 Small Benchmarks 10
2 Miss Rates for six small benchmarks and 3 replacement policies 11
3 Number of Family Members examined by Genetic Algorithm to find optimum . . 16

1 Introduction

There have been many investigations of the performance of different cache replacement policies [23, 6, 10, 13, 17, 24, 25, 11, 20, 28]. These have found that LRU generally performs significantly (10%) better than FIFO or Random replacement [23]. (For an exception, see [25].) Innovative variants have been suggested, for example Shadow Cache [20] and more recently, GODS [11]. As well, adaptive caching policies such as FIDO [18] have been proposed, although these tend to be geared towards applications higher in the memory hierarchy, such as disk paging and database objects. However, LRU or simplified variants of it, such as partition-LRU remain dominant [23].

The use of a software cache simulator and an address trace from a benchmark of interest, permits determination of the optimal hit rate achievable by any replacement policy on that benchmark. This optimal policy is often termed OPT. Although not implementable in a real cache because of its need for future information, OPT provides a yardstick against which other replacement policies can be measured. OPT has been found to provide a miss rate about 70% that of LRU for the same size and configuration of cache [20]. This large *LRU-OPT gap* leaves significant room for improvement.

In this paper we propose several new "families" of replacement policies. Most of the families that we propose seek to combine information used in traditional replacement policies. For example, in one family, we allow the cache to make use of LRU information and and a short history of hits and misses in each cache set. Another family makes use of both LRU and FIFO information in choosing which line to replace. The question is how to simply and effectively combine two types of information. (The brief Example near the start of Section 3.1 on page 6 illustrates how families represent combinations.)

Each family member provides a different combination. For instance in the LRU-FIFO family, one family member always replaces the LRU line unless that line is also the newest ("Last In") in the cache. Normal LRU and FIFO replacement are both members of the LRU-FIFO family as well. The family of possible replacement policies obtained by combining LRU and FIFO information is surprisingly large—2^{48} for a 4-way associative cache [3]. However this number is typical of the family size encountered even for relatively simple combinations.

Given such a large family size, some method is needed to efficiently search within the family for those policies that do well (significantly better than LRU). In most cases there is a wide range of performance within a family, with many policies producing results far worse than LRU. One method of search is to try and select family members (replacement policies) based on a model of program referencing behavior. This method would likely provide an interesting avenue for inquiry. However it has several drawbacks 1) we do not know of a model which predicts all the nuances of program behavior precisely 2) this method does not provide for systematic exploration of the entire search space 3) a relatively small number of policies could be investigated, 4) interesting and well-performing possibilities could easily be missed, and 5) programs with unusual referencing behavior would likely not benefit.

To try and avoid these problems and because we believe that the most effective way to capture program behavior is through "learning", we employed a genetic algorithm technique to search the families for good replacement policies. Genetic algorithms have been used successfully in a wide variety of applications including VLSI circuit layout [8, 22], adaptive filter design [12], the travelling salesman problem [14, 27], iterated prisoner's dilemma [4], and job shop scheduling [7]. Other areas of application range from cellular biology to demographics [15].

The basic notion in our a genetic algorithm approach is to mimic evolution's mechanisms in the search for a good replacement policy. For example take our LRU-FIFO family [1]. We start with a random population of 100 family members. We then see which members are fittest, i.e. we see which produce the highest hit rates for the benchmark(s) in question. (Hit rates are determined via a cache trace simulation.) The fittest members survive to reproduce and form a new generation. In reproducing, the fittest members pair off and exchange parts of their genetic code. The resulting offspring hopefully receive the best features of each parent. If not, they will likely be of low fitness and not survive to reproduce. Offspring also occasionally mutate.

This reproduction and mutation proceeds for generation after generation, with good replacement policies eventually evolving from the original random population of family members. Note that during the course of evolution, the offspring produced are always members of the LRU-FIFO family: this model of evolution does not allow the creation of new species. (The result of a mutation is merely the creation a distant relative in the LRU-FIFO family.) An LFU-FIFO (LFU=Least Frequently Used) family member could never result. The genetic algorithm merely acts to search (evolve) through the vast LRU-FIFO family looking for fit replacement policies.

We used this genetic algorithm approach to search through several families of replacement policies looking both for the best policy within a family, and ultimately looking for the family with the best policy. Best, of course, depends on the benchmark in question. We tried to find the best policy both for individual benchmarks and for a suite of benchmarks meant to typify a diverse workload.

In our experiments we first used very small 512-byte caches and a group of 6 generally small benchmarks. We found good replacement policies for these benchmarks, both individually and collectively. After this proof-of-concept, we applied our most successful techniques to 8 larger benchmarks and more realistic 8-kbyte caches.

The best common replacement policy found for a suite of benchmarks provided marginal improvement over LRU—the *LRU-OPT gain* was approximately 9% for an instruction cache. However, the hardware implementation of this policy is only slightly more complex than LRU and may be worthwhile in some instances.

The replacement policies found for individual benchmarks show that there are substantial gains to be made in allowing a cache to load a separate replacement policy for each program.

[1] Though we use the LRU-FIFO family for illustrative purposes, it is generally not the family with the best-performing members. LRU combined with a history of hits and misses is.

Such customization doubled the *LRU-OPT gain* from 9% to 18% for an instruction cache. To implement these individually customized replacement policies, we propose the use of a programmable replacement policy. This would require that in addition to the combinational logic used for LRU, that the cache have a small RAM (only 4 bytes in many cases) into which each program's replacement policy could be loaded, for example, by the operating system.

There is yet another improvement to be made: programmable replacement policies can be extended so that each set of the cache contains its own replacement policy. Such an extension produced an even more dramatic 27% *LRU-OPT gain* for an instruction cache. Luckily the hardware cost is still quite low, only 4 bytes of RAM per set in most cases.

However, there is a caveat. The genetic approach used here shows that substantial improvement is possible through the use of programmable replacement policies. The overhead incurred in using our genetic approach to find these policies is quite high—often hours of simulation time for a single benchmark. In certain cases, such "optimization" overhead may be justified considering the substantial performance gain achieved.

The next several sections discuss the material just presented in greater detail and provide more detailed results of our experiments. Section 2 provides additional background on the genetic algorithm approach used. Section 3 describes the replacement families studied, how we apply genetic algorithms to searching them, and the hardware needed for implementation. Section 4 provides results from our experiments, Section 5 discusses the robustness and comparative performance of the genetic algorithm approach and Section 6 concludes the paper.

2 Genetic Algorithms

This section provides a brief introduction to genetic algorithms. (For an excellent overview of the whole field, see [15].) Genetic algorithms have a long history [5, 21, 9, 16], but have only recently generated widespread interest. The current interest is due to several factors including: 1) a better understanding of the foundations of genetic algorithms, 2) development of computers sufficiently powerful to handle the large simulations required in most applications, 3) the growing number of applications to which genetic algorithms have been successfully applied, 4) the need for global optimization techniques capable of handling rough surfaces with many local optima.

Genetic algorithms attempt to mimic evolution to find optimal or near optimal solutions for a given problem. A number of approaches for genetic algorithms have been proposed. Most modern approaches use a population of binary strings (or strategies) all of the same length, l-bits. Each string represents a solution, with some strings representing a better solution than others. (Some function must exist to evaluate each string.)

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

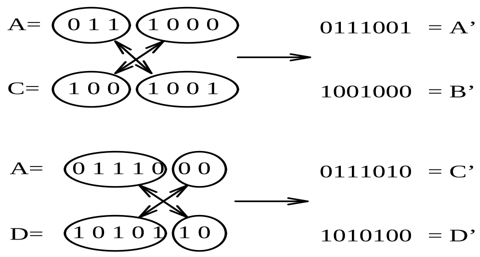


Figure 1: Crossover for two pairs of parents

strategies be evaluated to find the optimum of the function, and $l = 32$ is relatively small. Genetic algorithms attempt to reduce the search space to a manageable size at the risk of finding only an approximately optimal solution.

A simple, but typical genetic algorithm is outlined below, together with an example:

1. Generate a random population of l-bit binary *strings*. Example with $l = 7$: A=0111000, B=0101010, C=1001001, and D=1010110. In this paper each of these *strings* will correspond to a member of a replacement policy family such as *LRU-FIFO*.

2. Evaluate fitness of each string in the current population.

3. Use this fitness to determine which strings to reproduce in the next generation. 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.

4. Original generation "dies".

5. Pair off strings in new generation. Assume A/C and A/D are paired.

6. For each pair, randomly choose a bit position, p in the string. Choose 3 as the crossover position for A/C, and 5 as the crossover position for A/D.

7. Swap the bits to the right of p between the two strings in the pair *(Crossover)*. As shown in Figure 1: A'=0111001, B'=1001000, C'=0111010, and D'=1010100 .

8. Randomly mutate (with small probability) bits in resulting population. Choose the 2nd bit of C' and the 7th bit of D' to mutate. The new generation is: A'=0111001, B'=1001000, C'=0011010, and D'=1010101.

9. Goto Step 2.

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

The crossover operation is particularly important in finding good solutions. It can be proved that over time, crossover tends to produce populations with an exponentially increasing number of good small substrings (and an exponentially decreasing number of bad small substrings) [15, 16]. The hope is that strings containing these many good building blocks will have good overall performance. The next Section describes how cache replacement policy can be improved using this paradigm.

3 Cache Replacement Families

3.1 Types of Families and Applying the Genetic Algorithm

As previously noted, 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. The Introduction described one way of using *more* information, namely the use both LRU and FIFO information in deciding which line to replace. Keeping a *history* of lines accessed or replaced in each set is a way of using *different* information. The *history* could also include the history of hits and misses to each set or to the cache as a whole.

In addition to *LRU-FIFO*, several "combination" families are possible. We investigate the following:

1. **LRU-History:** *Combination of LRU and a hit/miss history of accesses to each set.*
2. **LRU-Count:** *Combination of LRU and a count of hits to each line.*
3. **FIFO-History:** *Combination of FIFO and a hit/miss history of accesses to each set.*
4. **FIFO-Count:** *Combination of FIFO and a count of hits to each line.*

We re-emphasize that the genetic algorithm searches only one family at a time looking for good replacement policies. To find which family has the best replacement policy, requires searching each family in turn. We also emphasize that the searches are done "offline," not dynamically by the cache.

Example: Consider *LRU-History*. Let the cache have $K = 4$-way associativity and let it record whether the 3 previous accesses to the set were hits or misses. (In a real cache this would require a 3-bit shift register in each set.) These specifications imply that each member of this *LRU-History* family has 16 bits:

- The hit/miss history of the 3 previous accesses is recorded in 3 bits, $Hist = B_2B_1B_0$. Let B_0 refer to the most recent access and B_2 to the most ancient.
- The replacement policy must indicate which line to replace on a miss for each of the $2^3 = 8$ possible combinations of hits and misses.
- On a cache miss, 4-way associativity $\Rightarrow$ 2 bits to specify the LRU rank ($0 = MRU$ to $3 = LRU$) of the line to be replaced.
- Since the replacement policy must indicate the LRU rank of the line to replace for each of the 8 combinations of hits and misses, each family member has $8 \times 2 = 16$ bits.

Since each replacement policy in this *LRU-History* family has 16 bits, this family has $2^{16} = 65536$ members, i.e. there are 65536 possible replacement policies. Since the rank of the least recently used line is assumed to be 3, family member, 33333333_4 or 65535, corresponds to LRU replacement: no matter what the history of hits and misses, the least recently used line is replaced.

In a real cache, the replacement policy corresponding to the best-performing family member would either 1) be built into the cache as combinational logic —mapping the 3 bits of history to 2 bits specifying the LRU rank of the line to be replaced, or 2) be loaded into a 16-bit register by the operating system. The cache would view the register as an $8 \times 2-$bit memory, again mapping the 3 bits of history to 2 bits specifying the LRU rank of the line to be replaced.

To find a good replacement policy from this *LRU-History* family, the genetic algorithm first randomly selects a small sample from the 65536 members of the family, just as A, B, C, and D were selected in Section 2. Each of the selected members is used as the replacement policy in a cache trace simulation. The trace represents the workload to which the cache is to be tuned. Those family members which yield the highest hit rates reproduce, crossover, and mutate to produce a new generation from the *LRU-History* family. This process continues for as many generations as desired–generally until improvement is slow or non-existent between generations.

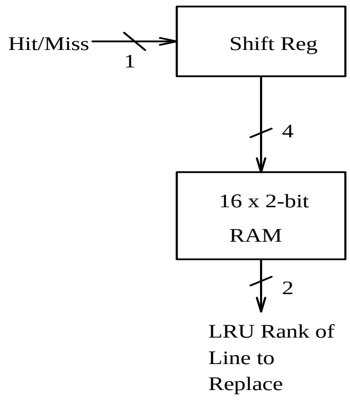


Figure 2: Block diagram of hardware for 4-bit *LRU-History Family* (Standard LRU Hardware not Shown)

To make this more concrete, consider an example of how one family member chooses replacements. Let the family member S under consideration be

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

As already indicated, S can be viewed as an array of 8, 2-bit elements, one for each possible history of hits and misses. Each element specifies the LRU rank of the line to replace for a given history. Further assume that $hit = 1$ and $miss = 0$, and that the current $Hist = 110$, i.e. on the last access there was a miss; the time before that, a hit; and longest ago, a hit. Assume that we now have a miss. Replacement works as follows

- The LRU rank of the line replaced is that specified by the $B_2B_1B_0 = 110_2 = 6th$ element in the string, $10_2 = 2$. In other words the second least recently used line is replaced.
- Since there was a miss, the new $Hist = 100$.
- Assume there is a hit to the set. The new $Hist = 001$.
- Another miss causes the line whose LRU rank is specified by the $001_2 = 1st$ element to be replaced. In this case $11_2 = 3$, so the LRU line is replaced.

3.2 Hardware

We emphasize that the hardware required to implement these families is low. As already described, the *LRU-History* family requires standard LRU hardware, a short 3 or 4 bit shift register for each set, and either a single 16 or 32-*bit* register or a small amount of combinational logic. If each set has its own replacement policy, the single 16 or 32-*bit* register is replaced with a 16 or 32-*bit* register in every set. In a cache with 4-way associativity and 32-byte lines (and hence 128 bytes per set), the overhead of these registers is only 2% or 4%. A block diagram of this hardware is shown in Figure 2. The other replacement families can be implemented with similarly simple hardware as outlined in [3].

On a cache hit, all that is required in addition to standard LRU bookkeeping is that a 1 be shifted into a shift register. On a miss, the LRU rank of the replacement line must be computed before the line is replaced, however, there should be ample time to do this while the new line is fetched from memory.

4 Results

4.1 Methodology

Results presented in this paper are divided into two parts. In the first part, results are given for a set of six relatively short traces generated from 6 benchmarks using the SPARC simulator, *SAS* [2]. These benchmarks were used with quite small 512-byte instruction and data caches with 32 byte lines and 4-way associativity. This small-scale setup served as a proof-of-concept. The short traces limited what were still quite lengthy simulation times, while the small cache size increased the number of conflict misses, thereby emphasizing the difference between replacement policies. These simulations confirmed the validity of our approach and also allowed us to find which families of replacement policies generally perform best.

In the second part, we took the best performing family of replacement policies and applied it to larger caches and to longer address traces. Specifically we studied 7 traces (*dec0, fora, forf, lisp, macr, pasc,* and *ue02*) generated using ATUM (Address Tracing Using Microcode) [2]. These ATUM traces have been extensively studied and reported on [2, 1]. We also used one address trace (*spice*) from the development of the MIPS architecture.

The measure used by the genetic algorithm to compare different replacement policies is not the overall hit rate, but the *line hit rate* [25]. *Line hit rate* is 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

[2] The 6 benchmarks were a calculator program (*ccal*), the *emacs* editor, a *kalman* filter routine, a Unix shell (*mcsh*), a symbolic polynomial program (*poly*), and *whetstone*.

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.

Line hit rates vary much more widely than do hit rates and ranged from less than 25% to over 95% for these benchmarks and caches. Hit rates for the 512-byte caches were generally between 80% and 90%, while for the 8-kbyte caches they were generally above 95%.

As already mentioned, we often employ a measure termed *LRU-OPT gain*. *LRU-OPT gain* measures how much of the possible gain from LRU to OPT is attained. For example if LRU has a *hit rate* of 60% and OPT has a hit rate of 75%, and a genetic algorithm approach achieves a *hit rate* of 65%, then the *LRU-OPT gain* is $\frac{65-60}{75-60} = 33\%$.

We found that genetic algorithm achieved good performance examining 100 replacement policies per generation. We generally allowed a fixed total of 9 generations, finding that in almost all cases, little or no improvement was obtained in later generations. For more details about the genetic algorithm parameters employed, see [3].

We did not employ any of the common trace reduction techniques, such as those of Puzak [20]. The reason is that much of the additional information that we use, such as histories of hit/miss information would be eliminated by such techniques.

The depth of history kept and the maximum counts are small: histories were 4-deep and counts were 1-bit. These small values provide performance nearly as good as using larger values [3].

4.2 Comparison of Replacement Families

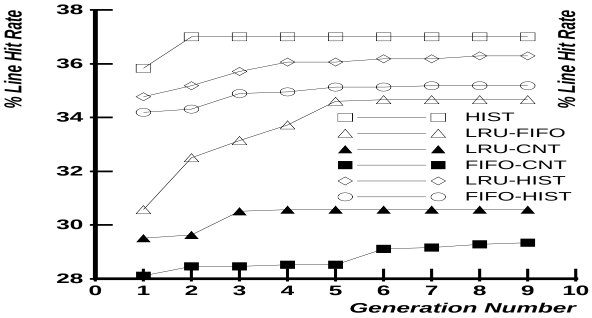
We first attempted to determine if any of the families outlined in Section 3.1 contain replacement policies which are generally superior to LRU. We met with limited success. To find such a policy we used traces from our six small benchmarks to simulate a multitasking workload, switching traces during the simulation approximately every 20,000 addresses.

For an instruction cache the *LRU-History* family contained the best replacement policy—it improved the overall miss rate for the six small benchmarks from 7.8% to 7.6%, an *LRU-OPT gain* of 9%. Every family but *History* contained a replacement policy which bested LRU. For data caches, *LRU-History* and *LRU-Count* fared best and approximately equally poorly, lowering the miss rate by only 0.03% and giving an *LRU-OPT gain* of only 2%. No policy was found in any of the other families that even matched LRU.

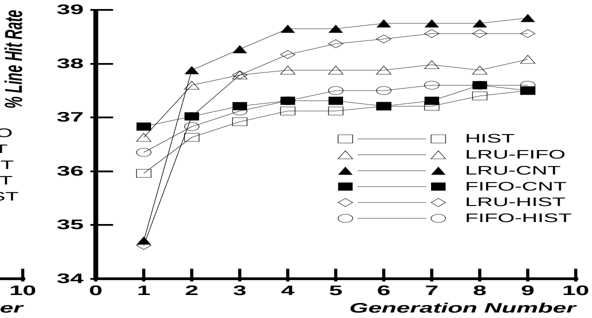
When this new instruction cache replacement policy was applied to the 8 larger ATUM and MIPS benchmarks, a minuscule 0.04% decrease over LRU in mean miss rate resulted. On the other hand the data cache replacement policy scored a 0.2% decrease. It is interesting

	I-Cache	D-Cache
LRU-FIFO	14	8
LRU-Count	7	10
LRU-Hist	18	9
History	8	2
FIFO-Count	2	-20
FIFO-Hist	14	-31
Mean	10	-4

Table 1: Mean *LRU-OPT Gains* for 6 Small Benchmarks



(a) *Whetstone I-Cache*



(b) *Whetstone D-Cache*

Figure 3: Improvement in Best Policy by Generation

to note, that the replacement policies found for the instruction and data caches were quite different, with the data cache policy closer to LRU, particularly when the previous access to the set was a hit.

Since these results were less than stellar, we were curious as to the cause. One possibility is that the genetic algorithm did not do a good job in finding the best policy in a family. While this may be partially to blame in some cases, we do not think that it is the primary cause for reasons outlined in Section 5.1. The other major possibility is that the families we propose contain no members that significantly improve upon LRU. This appears to be the case for general purpose workloads, due to LRU's strength across a wide variety of applications.

However, since the families we propose each contain many policies, they have an advantage over LRU: different family members can be used for different applications. As outlined in Section 3 all that is required is that the operating system write the desired replacement policy into a cache register. (The family of replacement policies is still hardwired into the cache.)

As can be seen in Table 1, when *LRU-History* is individually customized to the six small benchmarks, the mean *LRU-OPT gain* for instruction caches doubles from that achieved with

	INSTRUCTION			DATA		
	LRU	*LRU-Hist*	*LHBS*	LRU	*LRU-Hist*	*LHBS*
ccal	11.2	11.0	10.8	10.4	10.4	10.1
emacs	5.0	4.9	4.7	5.7	5.6	5.5
kalman	12.6	12.1	11.8	14.2	14.0	13.5
mcsh	7.7	7.2	7.1	5.9	5.8	5.8
poly	9.3	9.0	8.8	14.5	14.4	14.2
whetstone	8.5	7.8	7.5	17.4	16.8	16.4
MEAN	9.0	8.7	8.5	11.4	11.2	10.9

Table 2: Miss Rates for six small benchmarks and 3 replacement policies

a single common replacement policy to 18%. For data caches it goes from 2% to 9%.

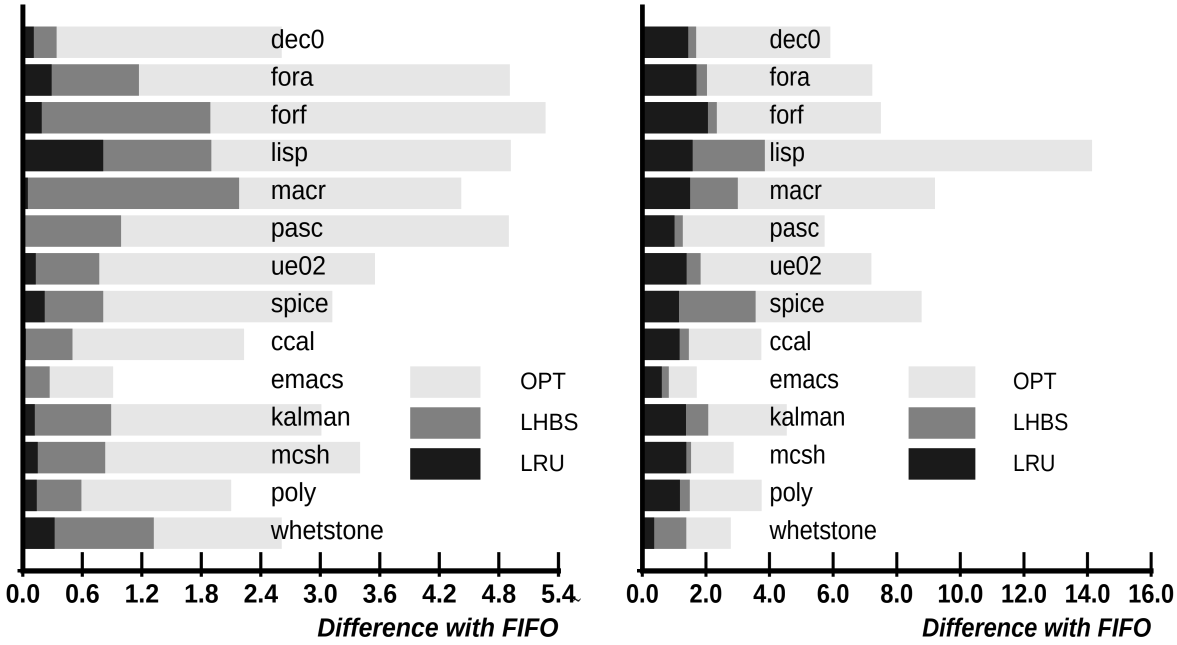
Improvement was not as great for all families. Performance was also not consistent, varying markedly, not only between families, but also within single families on different benchmarks and between instruction and data caches. Table 1 and Figure 3 give some indication of this, and that as was the case in the search for a single new replacement policy, *LRU-History* is the best overall performer. Not only does it have the highest mean *LRU-OPT gain* for instruction caches and nearly the highest for data caches, but it is a consistently high for all benchmarks and uniformly superior to LRU.

It is clear that substantially more improvement is obtained on instruction caches than data caches. There are at least two reasons for this:

1. The *line hit rate* in data caches is substantially higher than in instruction caches—62.4% on average versus 33.2% for LRU. Hence existing algorithms already perform well in data caches.
2. The other 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. LRU's *line hit rate* is 88.8% that of OPT on average for a data cache, while the ratio is 66.6% for an instruction cache.

The gains from customizing replacement policy to individual benchmarks are significant. However, if the replacement policy is optimized to individual benchmarks, each cache set can be customized with its own replacement policy, instead of having a common policy for the entire cache. We term the best *LRU-History/By-Set* policy found for a given benchmark, *LHBS*.

We found that use of *LHBS* significantly improved performance over a common replacement policy for the entire cache. Table 2 shows the performance improvement for each of the



(a) Instruction Cache (b) Data Cache

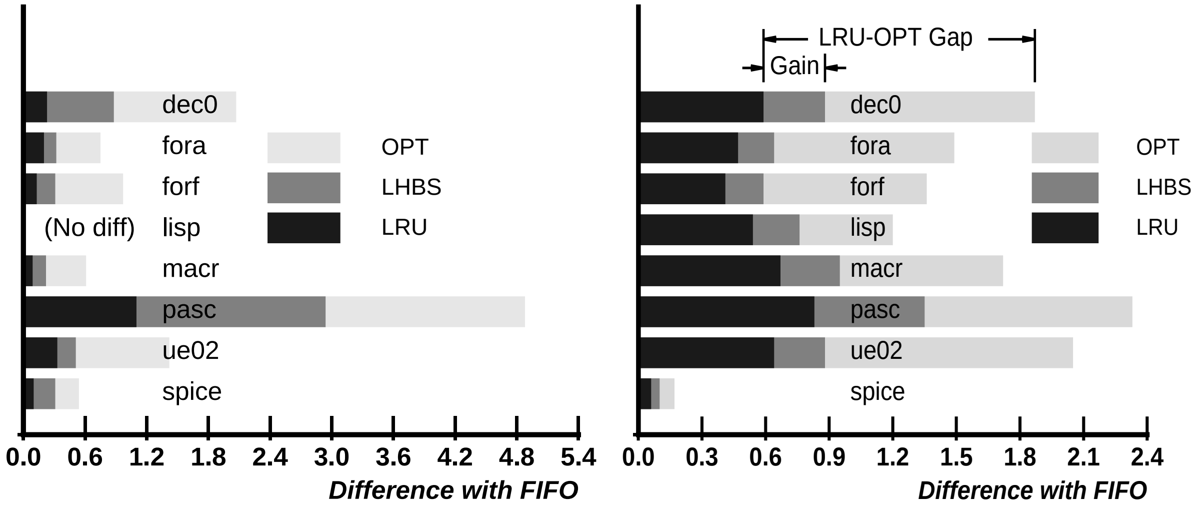
Figure 4: Differences in Miss Rate with FIFO for 512-byte Caches

six small benchmarks. The mean *LRU-OPT gain* jumps from 18% to 27% for instruction caches and from 9% to 20% for data caches. In individual cases the improvement provided up to a 1% absolute decrease in miss rate over LRU. Since for these benchmarks, the largest *LRU-OPT gap* was only a little over 3%, such improvement is quite respectable.

4.3 Performance on Larger Systems

After establishing that our replacement policies perform significantly better than LRU when customized to individual benchmarks, we applied it to larger ATUM and MIPS benchmarks, and measured performance on both 512-byte and 8-kbyte caches. This larger setup offered generally larger *LRU-OPT gaps* (up to 12.6%) and more chance for improvement. 8-kbyte instruction and data caches are also more typical of those used onchip in current architectures.

Results for this larger and more realistic setup are summarized in Figures 4 and 5. These Figures depict several statistics of interest. The long light bar depicts the gap in miss rate between FIFO and OPT. The shortest dark bar depicts the gap in miss rate between FIFO and LRU. Finally the medium gray bar gives the gap between FIFO and *LHBS*. As illustrated for *dec0* in Figure 5(b), the distance from the *right* end the LRU bar to the right end of the OPT bar depicts the *LRU-OPT gap*, while the fraction of this *LRU-OPT gap* covered by the *LHBS* bar is the *LRU-OPT gain* of *LHBS*. For *dec0* the *LRU-OPT gap* is 1.3%, while the *LRU-OPT gain* is 23%, i.e. 23% of the 1.3% gap is bridged.



(a) Instruction Cache (b) Data Cache

Figure 5: Differences in Miss Rate with FIFO for 8-kbyte Caches

As the Figures indicate, *LHBS* is always superior to LRU. The difference is especially marked for the 512-byte instruction cache, where the improvement of *LHBS* over FIFO average almost six times the improvement of LRU over FIFO. For an 8-kbyte instruction cache, *LHBS* still averages better than a two-fold improvement. For both 512-byte and 8-kbyte data caches, the improvement of *LHBS* over FIFO is about 1.5 times that of LRU. Clearly for an instruction cache and quite possibly for a data cache, if it is worthwhile to use LRU over FIFO, it is worthwhile to use *LHBS* over LRU.

Figures 4 and 5 also indicate the wide variation in how well traditional policies fare compared to OPT. For these 8 large benchmarks, the *LRU-OPT gap* ranges by an order of magnitude from 0.11%[3] to 12.6%. Yet over this entire range, *LHBS* is able to improve performance yet further, dropping the 0.11% by 0.04% to 0.07% and the 12.6% by 2.3% to 10.3%. These correspond to 36% and 18% *LRU-OPT gains* respectively. The mean *LRU-OPT gain* for the 8 large benchmarks and an 8-kbyte instruction cache is over 27% while the mean for an 8-kbyte data cache is 26%.

These gains are actually better than the 24% and 12% *LRU-OPT gains* recorded for 512-byte caches. A likely reason for the stronger improvement in the larger cache is that an 8-kbyte cache has 64 sets, while a 512-byte cache has only 4. Since under *LHBS* each set specifies its own replacement policy, an 8-kbyte cache has much wider latitude in dealing with unusual behavior in individual sets. This latitude is exploited to yield the larger *LRU-OPT gains*.

Predicting which benchmarks will show the most improvement from use of *LHBS* is generally difficult. There was no apparent correlation between *line hit rate* and 1) *LRU-OPT gain*,

[3]An 8-kbyte instruction cache running the LISP benchmark actually had no difference between LRU and OPT.

2) absolute percent improvement in miss rate, or 3) relative percent improvement in miss rate. As might be expected, there was a weak negative correlation between *hit rate* and absolute improvement in miss rate, but no apparent correlation between hit rate and the other two measures of improvement.

5 Robustness and Comparative Performance

5.1 Robustness of Genetic Algorithm Approach

We have shown that within the replacement policy families which we have examined, there exist policies which perform significantly better than LRU. However, we have not yet established that the policies that we have found within these families are the best that exist there. To show that the genetic algorithm approach does indeed perform well in finding good policies within our replacement families, we made four checks.

The first check was the performance of the genetic algorithm with the LRU-combination families (*LRU-History*, *LRU-Count*, and *LRU-FIFO*). Each of these families includes LRU as a replacement policy. Hence it is always possible to do at least as well as LRU. Out of 116 separate instances in which these combination families were searched by the genetic algorithm, in only 4 cases did the genetic algorithm fail to find a policy at least as good as LRU.

The second check was to select a random population from each family equal in size to the number of family members examined by the genetic algorithm. If the genetic algorithm is effective, it should find better replacement policies than the random selection. 84 combinations of benchmark and replacement family were examined. In every case the genetic algorithm found a family member with performance equal to or better than the random selection. In more than 95% of the cases, the genetic algorithm found a family member with superior performance.

The third check was to plot the distribution of generations at which the best replacement policy was found. If genetic algorithms 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, 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.

The fourth and final check was to execute an exhaustive search for the best replacement policy in four limited cases. Since the replacement policy families studied elsewhere in this paper all have at least 4 billion members, exhaustive search of them was precluded. Instead we

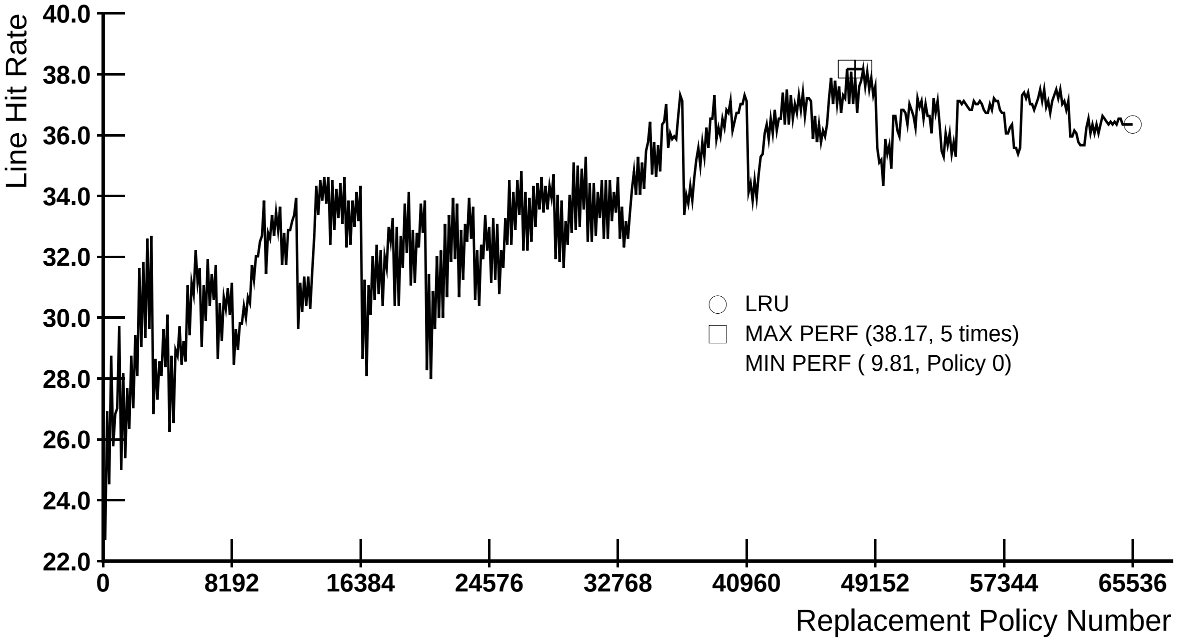


Figure 6: Exhaustive Search of 3-bit LRU-History Family, 512-byte Data Cache, Whetstone Benchmark

examined an *LRU-History* family, but with only 3-bits of history maintained. This family has 65536 replacement policies. We performed a cache trace simulation for each of these replacement policies for two small benchmarks and for 512-byte instruction and data caches.

In 3 of the 4 cases, the genetic algorithm found an optimal replacement policy. (In each of these 3 cases there were multiple optima, making the task slightly easier.) The multiplicity of optima and the number of family members examined by the genetic algorithm to find them are listed in Table 3. The Table indicates that the performance of the genetic algorithm was much superior to chance. In the fourth case the genetic algorithm found one of two replacement policies whose *line hit rate* was only 0.01% below the global optimum.

Figure 6 depicts the search space for the Whetstone benchmark and a 512-byte data cache. Note that family member 65535 at the far right of the graph is LRU replacement. Clearly many members of the family fare better than LRU. As Figure 6 indicates, the search space has large numbers of local optima, many of which are almost as good as the global optima. The roughness and irregularity of the search space testify to the robustness of the genetic algorithm approach. Figure 6 illustrates one other essential point. The range of performance of the different policies ranges from a *line hit rate* of 9.8% to 38.2%. This wide spread is typical and in searching for good policies, the genetic algorithm also uncovers many that are terrible.

5.2 Comparison of our Results to Other Work

Benchmark	Cache Type	Multiplicity of Optima	Number of Members Examined
Emacs	Ins	28	200
Whetstone	Ins	12	300
Whetstone	Data	5	500
Emacs	Data	1†	900

†Found one of two points with *line hit rate* 0.01% less than the optimum.

Table 3: Number of Family Members examined by Genetic Algorithm to find optimum

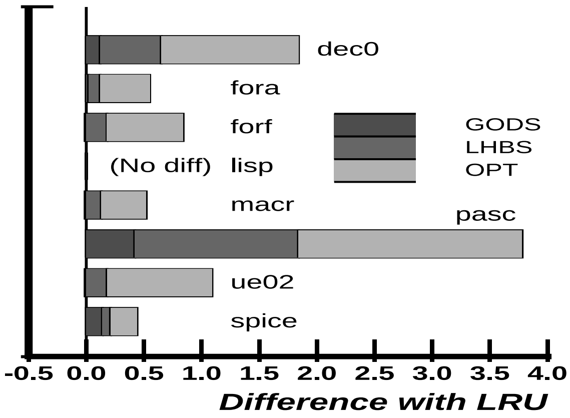


Figure 7: Comparison of GODS, *LHBS*, and OPT for 8-kbyte Instruction Cache

We conclude our results with a comparison of LHBS to two other recent innovations that claim to best LRU. The first innovation is Shadow Cache [19], a hybrid replacement policy somewhat similar to those described here, particularly *LRU-History*. In a Shadow Cache a larger number of tags is kept 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*][26]."

The other innovation is a class of replacement policies developed by Deville and Gobert [11]. The best performing of these and the one against which we compare is GODS. GODS is a simplification of LRU that has that added bonus of generally outperforming it.

As can be seen in Figures 7 to 10, *LHBS* substantially outperforms both Shadow Cache and GODS. *LHBS* has a mean *LRU-OPT gain* of 27% for an 8-kbyte instruction cache versus 6.1% for GODS and 1.2% for Shadow Cache. The story is similar for an 8-kbyte data cache: a

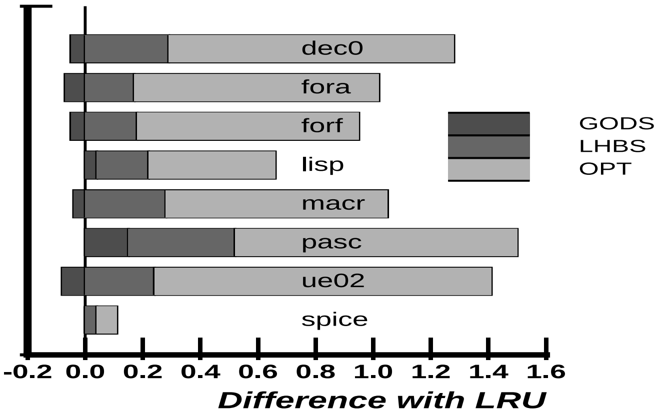


Figure 8: Comparison of GODS, *LHBS*, and OPT for 8-kbyte Data Cache

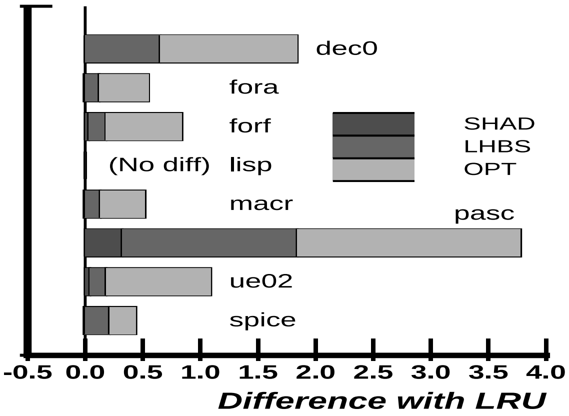


Figure 9: Comparison of Shadow Cache, *LHBS*, and OPT for 8-kbyte Instruction Cache

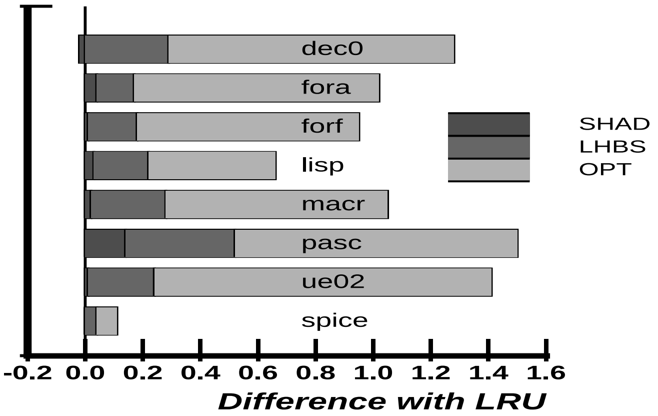


Figure 10: Comparison of Shadow Cache, *LHBS*, and OPT for 8-kbyte Data Cache

26% *LRU-OPT gain* for *LHBS* versus -1.2% for GODS and 2.5% for Shadow Cache.

6 Conclusion

This paper has described the use of genetic algorithms to find replacement policies which improve upon the performance of traditional cache replacement policies. Six families of replacement policies were proposed and investigated, with *LRU-History* the all-around best performer. In particular our approach exhibited the following strengths:

- Significant improvement was obtained over LRU. *LHBS* showed *LRU-OPT gains* of up to 49% and averaged over 25%.
- Hardware to implement *LHBS* is small, simple, and fast.
- Significant improvement was made over two other recent proposed alternatives to LRU: GODS and Shadow Cache.
- Use of genetic algorithms to search families for good replacement policies was successful.

Despite its success, the genetic algorithm/full cache simulation approach to finding the *LHBS* policy for a particular benchmark is quite time consuming. Finding still faster means of generating good *LHBS* policies is an interesting area for future research.

References

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating systems and multiprogramming. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 119–127, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 14(2), June 1986.

[3] Erik R. Altman. Genetic algorithms and cache replacement policy. Master's thesis, McGill University, Montréal, Québec, September 1991.

[4] 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.

[5] J. D. Bagley. *The Behavior of Adaptive Systems Which Employ Genetic and Correlation Algorithms.* PhD thesis, University of Michigan, Ann Arbor, 1967.

[6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[7] L. Davis. Job shop scheduling with genetic algorithms. In *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, pages 136–140, 1985.

[8] L. Davis and D. Smith. Adaptive design for layout synthesis. Technical report, Texas Instruments, Dallas, Texas, 1985.

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

[10] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968. Also in 26(1):43-48, January 1983.

[11] Yannick Deville and Jean Gobert. A class of replacement policies for medium and high-associativity structures. *Computer Architecture News*, 20(1):55–64, March 1992.

[12] D. M. Etter, M. J. Hicks, and K. H. Cho. Recursive adaptive filter design using an adaptive genetic algorithm. In *Proceedings of the 1982 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 635–638, Paris, France, May 3–5, 1982.

[13] J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[14] 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.

[15] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[16] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[17] W. F. King III. Analysis of demand paging algorithms. In *Information Processing 71 — Proceedings of IFIP Congress 71*, volume 1, pages 485–490, Ljubliana, Yugoslavia, August 23–28, 1971.

[18] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 255–264, Barcelona, Spain, September 3–6, 1991.

[19] J. Pomerene, T. R. Puzak, R. Rechtschaffen, and F. Sparacio. Prefetching mechanism for a high-speed buffer store, 1984. Patent Pending.

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

[21] R. S. Rosenberg. *Simulation of Genetic Populations with Biochemical Properties.* PhD thesis, University of Michigan, Ann Arbor, 1967.

[22] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2):143–220, June 1991.

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

[24] James E. Smith and James R. Goodman. A study of instruction cache organizations and replacement policies. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 132–137, Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News,* 11(3), June 1983.

[25] James E. Smith and James R. Goodman. Instruction cache replacement policies and organizations. *IEEE Transactions on Computers*, 34(3):234–241, March 1985.

[26] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[27] 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, Cambridge, Massachusetts, July 28–31, 1987.

[28] Kishor S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, 26(10):938–947, October 1977.