Evaluation of Neural and Genetic Algorithms For Synthesizing Parallel Storage Schemes

Mayez Al-Mouhamed
Computer Engineering Department
CCSE, King Fahd University
Dhahran 31261, Saudi Arabia.
mayez@ccse.kfupm.edu.sa

Hussam Abu-Haimed
Department of Computer Science
Stanford University
Stanford, CA 94309.
husam@leland.stanford.edu

Abstract

Exploiting compile time knowledge to improve memory bandwidth can produce noticeable improvements at run-time [13, 1]. Allocating the data structure [13] to separate memories whenever the data may be accessed in parallel allowed improvements in memory access time of 13% to 40%. We are concerned with dynamic storage schemes for which the compiler can predict some of the *access patterns* of parallelized programs. A storage scheme provides a mapping from array addresses into storages. However, finding a conflict-free storage scheme for a set of data patterns is NP-complete. This problem is reduceable to *weighted graph coloring*. Optimizing the address transformation is investigated by using: (1) *constructive heuristics*, (2) *neural methods*, and (3) *genetic algorithms*. The details of implementation of these different approaches are presented. Using realistic data patterns, simulation shows that memory utilization of 80% or higher can be achieved in the case of 20 data patterns over up to 256 unbuffered parallel memories. The neural approach was relatively very fast in producing reasonably good solutions even in the case of large problem sizes. Convergence of proposed neural algorithm seems to be only slightly dependent on problem size. Genetic algorithms outperformed all others and are recommended for advanced compiler optimization especially for: (1) large problem sizes, and (2) applications which are compiled once and run many times over different data sets.

1 Introduction

There is pressing need for innovative memory architectures and organization [8, 2] to reduce bandwidth unbalancing between processor and main memory. One problem is how to map data structures onto parallel memories so that to favor a class of access patterns [2].

Interleaving causes significant performance impairment due to non-uniform memory access in the case of stride and block accesses. Sohi [14] proposed bit-wise boolean address transformations for vector processors in order to determine the memory number where a given array element should be stored. Buffers at memory inputs and outputs were used to reduce the effects of transient degradation in pipelined memories. Single linear and nonlinear data patterns like diagonal and coils can be accessed without conflicts by using *multiskewing* [3] which uses different linear skewing schemes over different sections of the array.

Improving memory bandwidth in hierarchical memory systems aims at exploiting compile time knowledge to reduce unnecessary data transfer between processor and main memory. Compiler optimization that attempts maximizing *temporal* and *spatial* localities and minimizing mapping conflicts produced encouraging results [12, 9]. Optimized programs for direct mapped caches resulted in lower miss ratio [12] than unoptimized programs operating on a set associative cache of the same size.

To reduce memory conflicts in multiprocessors, compiler directed compaction-based data partitioning [13] was applied to a class of synchronous dataflow computations. The data structure is allocated to separate memories whenever the compaction algorithm finds that data may be accessed in parallel. Partial duplication of data was also used. Improvements in performance ranging from 13% to 40% were obtained. Another technique called *compiler directed page coloring* [1] uses compiler's knowledge of the access pattern of parallel applications to direct run-time virtual memory page mapping. Here the compiler (Stanford SUIF) explicitly attempts predicting the *access patterns* of compiler parallelized applications which gives more that 50% improvement over a standard page mapping policy.

We are concerned with storage schemes for which the compiler can predict some of the array data patterns that are accessed at run time. Our objective is to find *address transformation*, as part of processor address translation, that minimizes overall access time for arbitrary sets of data access patterns. Finding conflict-free storage scheme for accessing an array by using arbitrary sets of data patterns is NP-complete. This problem is reduceable to weighted graph coloring. Optimizing the address transformation is investigated by using: (1) constructive heuristics, (2) neural methods, and (3) genetic algorithms.

This paper is organized as follows. Section 2 presents some background. In section 3, we review the basis for data patterns and storage schemes. In sections 4 we present three compiler methods for synthesizing storage schemes. Evaluation of synthesized storages for each method is carried out in Section 5. Conclusions are presented in Section 6.

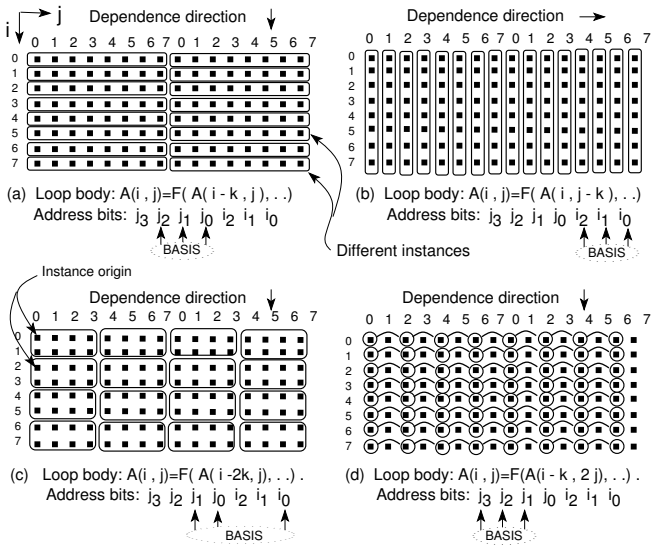


Figure 1: Patterns: (a) rows, (b) columns, (c) 2×4 block, and (c) row with stride 2

2 Background

Figure 1 shows an 8×16 array that is partitioned into a set of 8-elements row pattern (a), column pattern (b), 2×4 block pattern (c), and row with stride-2 pattern (d). The patterns shown are accessed during execution of given loops which have loop-carried dependencies. The accessed patterns are dictated by the data dependence and compiler restructuring.

Consider the parallel access to 8 memories for the array elements $a(i, j)$ shown in Figure 1-(a) and notice that components (j_2, j_1, j_0) take all possible combinations for the 8 array elements of each pattern instance. In Figure 1-(a), the accessed pattern (T_1) consists of a sub-row of 8 successive elements of array A. T_1 is associated a basis $B(T_1) = \{g_2, g_1, g_0\}$, where g_2, g_1, and g_0 are canonical vectors. Note that (j_2, j_1, j_0) are the components of (i, j) over the basis $B(T_1)$, i.e. projection of (i, j) over $B(T_1)$. Note that when accessing any instance of the same pattern the address bits (of its elements) other than those defined over its basis are constant. These bits are used as the *pattern origin*. For example, (j_3, i_2, i_1, i_0) represent the pattern origin of T_1 and used to select one given instance. By changing the origin we can access different pattern instances.

Pattern T_2 shown on Figure 1-(b) allows accessing subcolumns of 8 successive elements. The basis of T_2 is $B(T_2) = \{f_2, f_1, f_0\}$. Finally, patterns T_3 and T_4 shown in Figures 1-(c) and (d) have basis of $B(T_3) = \{g_1, g_0, f_0\}$ and $B(T_4) = \{g_3, g_2, g_1\}$, respectively.

Finding a storage scheme that allows conflict-free access to one of the above patterns does not pose any problem. During pattern access, one may take the components of accessed elements over the pattern basis as storage numbers. We are interested in finding efficient storage schemes that allow minimum access time for an *arbitrary set of data patterns*.

3 Analysis of storage schemes

Consider a storage matrix M for the data patterns T_1, T_2, T_3, and T_4 that is formed by a 3×7 matrix which can be selected as:

$$M.x^b = \begin{array}{c} \begin{matrix} f_2 & f_1 & f_0 & g_3 & g_2 & g_1 & g_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array} . \begin{pmatrix} i_2 \\ i_1 \\ i_0 \\ j_3 \\ j_2 \\ j_1 \\ j_0 \end{pmatrix} \quad (1)$$

where addition and multiplication are modulo 2. The pattern sub-matrix M^{T_k} is formed by the m columns of M that are the images by M of all canonical vectors of basis $B(T_k)$. For example M_{T_1} and M_{T_2} are:

$$M_{T_1} = \begin{array}{c} \begin{matrix} g_2 & g_1 & g_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \quad M_{T_2} = \begin{array}{c} \begin{matrix} f_2 & f_1 & f_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

For this consider the parallel access to some instance of pattern T_3 for which the origin is $(i_2, i_1, 0,\ j_3, j_2, 0, 0)$ and $B(T_3) = \{f_0, g_1, g_0\}$. Let x^b be the projection of (i, j) over $B = \cup_{1 \le k \le 4} B(T_k)$ so that the accessed elements of T_3 are defined by $\{x^b\} = \{(i_2, i_1, i_0, j_3, j_2, j_1, j_0)\} = (0, 0, -, 1, 1, -, -)$, where i_0, j_1, j_0 take all possible combinations of bits. In this case, the element $a(i, j)$ is stored into memory:

$$M.x^b = \begin{array}{c} \begin{matrix} f_2 & f_1 & f_0 & g_3 & g_2 & g_1 & g_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array} . \begin{pmatrix} 0 \\ 0 \\ i_0 \\ 1 \\ 1 \\ j_1 \\ j_0 \end{pmatrix}$$

The product Mx^b can be decomposed into the following sum:

$$M.x^b = \begin{array}{c} \begin{matrix} f_2 & f_1 & g_3 & g_2 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} . \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{array}{c} \begin{matrix} f_0 & g_1 & g_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \end{array} . \begin{pmatrix} i_0 \\ j_1 \\ j_0 \end{pmatrix} =$$

$$\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \oplus M_{T_3} . \begin{pmatrix} i_1 \\ j_1 \\ j_0 \end{pmatrix} \quad (2)$$

The parallel access to instances of T_3 only requires that M_{T_3} be non-singular. Summing a constant to $M_{T_3} x^{b(T_3)}$ changes the naming of the storages but maintain one-to-one mapping between the elements of the accessed pattern and the memories. It can be easily proved that M allows parallel access to patterns $T = \{T_1, \ldots, T_q\}$ if and only if each submatrix M_{T_k} is non-singular.

Figure 2 shows the mapping of each array address (i, j) into the memory module number $M(i, j)$ where array element $a(i, j)$ is stored, i.e. $M(i, j) = Mx$ and x is being the projection of (i, j) over B. Here all four patterns can be accessed without conflicts because all corresponding pattern matrices are non-singular. Since M_{T_1} has full rank, the offset can be taken as (i_2, i_1, i_0, j_3) because $B(T_1) = \{j_2, j_1, j_0\}$. It can be easily shown that all array elements which are

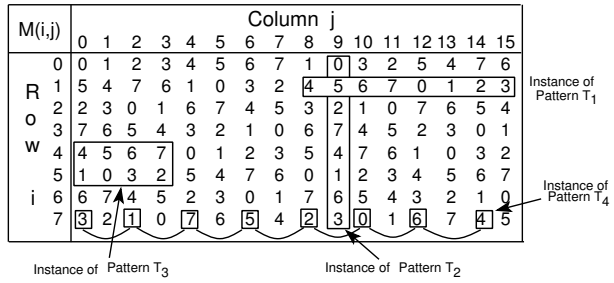


Figure 2: Mapping of array element (i, j) to memory $M(i, j)$ for 8 memories

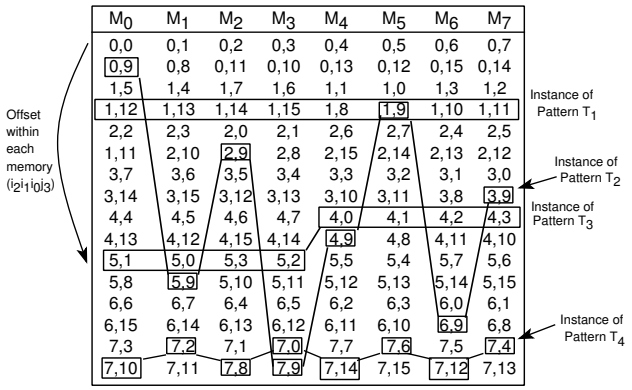


Figure 3: Storage of elements (i, j) into memories and offset

mapped to the same memory fall into distinct offsets if and only if the sub-matrix M_{T_1} is non-singular. Figure 3 shows the storage of array elements (i, j) into the eight memories. Each array element $a(i, j)$ is stored into memory $M(i, j) = Mx$ at offset (i_2, i_1, i_0, j_3).

Now we study the NP-completeness. Suppose we are given a vector space Z_2^m, set of variables $B = \{t_{m-1}, \ldots, t_0\}$, and a set $\Gamma = \{T_1, T_2, \ldots, T_q\}$ such that $B(T_k)$ is any set of n vectors of B. Each vector $t_u \in B$ must appear in some $B(T_k)$. The problem is to assign each $t_u \in B$ a vector in Z_2^m, such that for all T_k, the vectors assigned to all t_u in T_k are linearly independent. We call this problem *linear independence satisfiability* (LIS). Consider the case where $m = 2$ and $|B(T_k)| = 2$ for all $T_k \in T$. We call this problem 2-LIS. The vectors in Z_2^2 are:

$$z_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad z_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} z_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad z_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Note that z_0 cannot be assigned to any variable. Let $Z^* = \{z_1, z_2, z_3\}$. Any two distinct members of Z^* are linearly independent. Therefore, for each $B(T_k) = \{t_x, t_y\}$ we must assign to t_x and t_y distinct members of Z^*. We call (B, Γ) the *conflict graph* of T. Each vertex in the graph is a variable t_x, and there is an edge between t_x and t_y if and only if $\{t_x, t_y\}$ is in some T_k. We can solve 2-LIS if and only if the conflict graph is 3-colorable.

2-LIS is obviously in NP. To prove that 2-LIS is NP-complete, consider an arbitrary undirected graph. For all the vertices of degree greater than zero, we create a variable. For each edge (t_x, t_y), we create a $T_k = \{t_x, t_y\}$. We use the algorithm for 2-LIS to assign each t_x a value in Z^*. We use this assignment to color the non-zero degree vertices of the conflict graph. We then color the degree-zero vertices with some fixed color. This clearly show that LIS is NP-complete.

Note that finding a general storage scheme for 4 memory units is NP-complete, Also note that we can build conflict graphs only for $m = 2$. We need to find a model of storage schemes for $m > 2$, from which good heuristics can be derived.

4 Synthesis of heuristic storage schemes

The access frequency $f(T_k)$ of pattern T_k is the number of times a pattern is accessed. Given bases vectors t and t', the weight of edge (t, t') is $\omega(t, t') = \sum_{t,t' \in B(T_k)} f(T_k)$. The weight of t is $\omega(t) = \sum_{t' \in B} \omega(t, t')$.

The number of access cycles $C(M)$ for a combined storage scheme M is the sum of the access cycles of all of its q patterns $T_1, \ldots, T_q$. The least number of cycles to access all the patterns is $F(M) = \sum_{i=1}^{q} f(T_i)$. If each pattern T is accessed $f(T)$ times, then we need $C(M) = \sum_{i=1}^{q} f(T_i) 2^{n - rank(M_{T_i})}$ cycles if scheme M is used. The performance function is $U(M) = F(M)/C(M)$ that is a lower bound on parallel *memory utilization* for the storage scheme M.

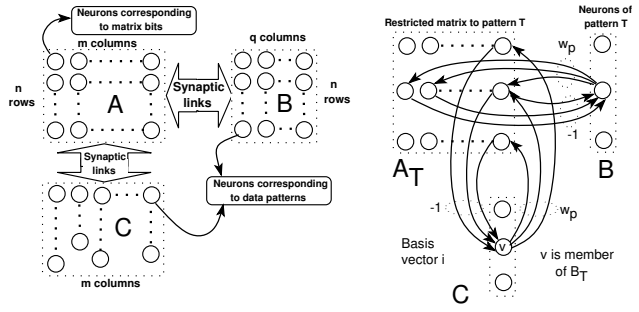
We present the details of three compiler optimization methods for finding the storage scheme which are: (1) *constructive coloring*, (2) *neural methods*, and (3) *genetic algorithms*.

4.1 Weighted coloring with node splitting

Weighted coloring with node splitting (WCNS) operates on weighted conflict graphs and perform node splitting when it fails in coloring a node. WCNS repeats until all the nodes are colored while always choosing an uncolored node v with the highest weight. Node v is colored with the smallest available color that is not used by its neighbors. In the case, all the available colors have been assigned to the neighbors of the current node v, then v is split into two nodes v' and v''. The splitting operation must divide the pattern bases that contain v into two groups which nearly have equal weights. Whenever a node is split, WCNS re-evaluate the weights for all uncolored neighboring nodes and restart again.

A node u that is present in only in one pattern basis has necessarily $n - 1$ neighbors. Such a node u can always be colored without splitting. A node v that is split is necessarily present in more than one pattern basis because it has at least n neighbors that are all assigned the n colors. Splitting node v into v' and v'' means that some of the pattern bases that contain v will be represented by v' and the other bases will be represented by v''. Node splitting has the effect of reducing the degree of conflicts with other vectors at the cost of duplicating the encoding of vectors (1s) in the storage matrix.

It can be easily shown that the time complexity of WCNS is $O(m^3 - m^2 n)$, where m is the number of nodes, n is the



a - A is the image of storage matrix, each column in B is for one pattern, each neuron in column i of C is for membership of basis vector i of some pattern basis.

b - Synaptic links for Architecture I

Figure 4: Neural network NN-1

number of colors, and 2^n is the number of memories.

4.2 A neural approach

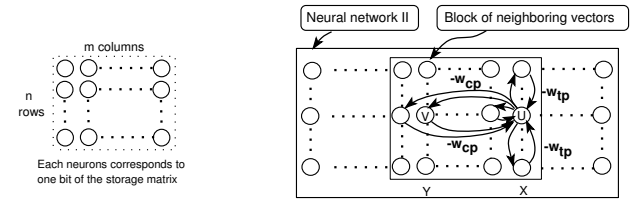
An artificial *Neural network* (NN) [11] is a collection of interconnected neurons each has a number of input synapses, a body, and output synapse. Let v_i be the output of neuron n_i and $w_{j,i}$ be the weight of synapse $s_{j,i}$ that is connecting the output of neuron n_j to input of n_i. The output of neuron n_i is a non-linear unit step function $v_i = F(\sum_{j=1}^{N} w_{j,i}v_j - \theta_i)$, where N is the number of neurons and θ is a threshold.

The NN architecture most commonly used in combinatorial optimization problems [11] is the *feedback network* (FN) [7]. The feedback causes the network to iteratively produce transient solutions before reaching a stable state.

In FN the energy function ($E = -\frac{1}{2}\sum_i \sum_j w_{i,j}v_i v_j$) will be in a local optima when the network converges [7, 10]. Therefore, the designer has to: 1) define how a network output should be decoded into a solution, and 2) map the objective function into the energy function. This mainly consists of setting the synaptic weights.

Finding correlated non-singular matrices is complex. We use more flexible conditions that are likely to produce non-singular matrices which are: 1) all columns and non-zero, 2) all rows are non-zero, and 3) promotion of dissimilar assignment of values to neighboring components of distinct vectors. Two NNs are presented to promote generation of vectors that are linearly independent which are denoted by (NN-1) and (NN-2).

In NN-1, the net consists of three blocks of neurons which are $A(n, m)$, $B(n, q)$, and $C(., m)$ as shown on Figure 4-a. Every neuron in block A represents one bit in the storage matrix which means that A will be the solution. There are 2^n memories and m distinct vectors in the union of all pattern bases. All neurons in A have a threshold of zero. Each column of neurons of B represents one pattern. Basis vector e_k is associated column k of C which has one neuron for each pattern basis (at most q) to which e_k belongs to. In other words, columns of C have different sizes and the maximum size of C is $q \times m$. All neurons in B and C have a threshold of -0.5.



a - NN-II is a direct image of storage matrix, each column correponds to a basis vector.

b - A neuron U of column X is connected to each neuron is the same column and to each neuron V of column Y if X and Y are in the basis of at least one pattern.

Figure 5: Neural network NN-2

All synaptic connections are inter-block as shown in Figure 4-b. The output of one neuron of B, which corresponds to a pattern T, has connections to only row neurons of A that corresponds to the restricted matrix M_T. The weight of each of these connections $w(T)$. The output of one neuron of C, corresponding to a pattern T, and all neurons of A in the same column have $w(T)$ as weight.

All connections departing from A have weight of -1. Each neuron in A is connected to all neurons of C in the same column. Also each neuron in A, corresponding to a basis vector e_j, is connected to all neurons of C corresponding to any pattern whose basis contains e_j. The connections between A and C guarantee that *all columns of A are non-zero.* At the ith iteration, the output $v_i(t+1)$ of a neuron of C is 1 as long as its inputs are all zeros (threshold is -0.5):

$$v_i(t+1) = \begin{cases} 1 & \text{if } S_i > \theta_i \\ v_i(t) & \text{if } S_i = \theta_i \\ 0 & \text{if } S_i < \theta_i \end{cases}$$

where, $S_i = \sum_{j=1}^{N} w_{j,i}v_j$. This causes one of the neurons of A in the same column to output a 1, which in turn forces the outputs of all neurons in the same column to output zeros. Similarly, connections between blocks A and B guarantee that *all rows of A are non-zero.*

When the output of a neuron of A is 1, the chances of another neuron getting 1 in same row of the restricted block (matrix) are reduced. At least one of the neurons in that row of B will go off and hence the input sum of these vectors gets reduced. This guarantees that neighboring components of vectors of A get *dissimilar assignments* of values.

The update approach consists of selecting the neuron n_i whose $|S_i - \theta_i|$ is the largest. The neuron output is updated if needed (different) in which case we must propagate the updated values wherever necessary prior to restarting the next iteration. If an update is not required (same value), then the neurons are visited in decreasing order of $|S_i - \theta_i|$ until an update is found or the algorithm terminates.

The time complexity of NN-1 is $O(N^2+kN)$, where $N = nm+q(n+m)$ is the number of neurons and k is the number of iterations. From our experience k is very close to $m + n$.

NN-2 is based on the idea of *force directed optimization* (FDO) with the aim of producing dissimilar assignments of vectors in sub-matrices associated to patterns. Figure 5-a shows NN-2. In NN-1 similar action was generated, but with external forces. In NN-2 there is only one block which cor-

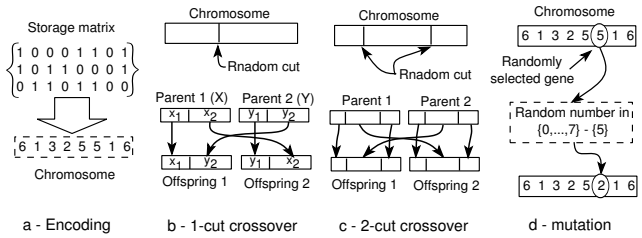


Figure 6: GA: encoding of solution (a), 1-cut crossover (b), and 2-cut crossover (c)

responds to A in NN-1. In the FDO technique, the neurons corresponding to each basis vector directly enforce neurons of neighboring vectors to have distinct assignments. Two basis vectors are neighbors if they belong to the basis of at least one data pattern.

The outputs of a neuron in the ith row of kth column are connected to all neurons in kth column and to all neurons of the ith row which correspond to neighboring vectors. This means that row connections are intra-block with respect to each data pattern. The weight of a row synapse that is linking neurons $n_{i,j}$ to $n_{i,k}$ is proportional to the sum of access frequencies of all the patterns to which both basis vectors e_j and e_k belong to. The weight is negative to cause an inhibitory action on the receivers. The setting of row synapses is meant to prevent neurons in the same row of having similar values.

The weight of a column synapse is proportional to the access frequency of all patterns (w_{tp}) the generating neuron (also basis vector) belongs to. Here also the weight is negative. Column synapses prevent assigning ones to more than one row in a column. Using coloring heuristics, the least weighted vector is generally split when no color is available. Having two 1s or more for a basis column vector is equivalent to splitting that vector. Since the synaptic links have $-w_{tp}$ as weights, highly weighted neurons in a column are unlikely to have more than one 1. This means that the corresponding basis vector is unlikely to be split. The synapses of NN-2 are shown on Figure 5-b.

The thresholds are set in the neurons of NN-2 to $-w_{tp}$ to give priority of output update to neurons in highly weighted vectors. Hence at the beginning when all neuron outputs are zero, $S_i = w_{tpi}$ and the highest weighted vector (highest w_{tpi}) will be updated first which corresponds to updating the coloring of the highest weighted node first.

4.3 A genetic approach

Genetic evolution [4, 6] is based on: (1) *selection* of the fittest gene, and (2) *reproduction or crossover* that consists of recombining segments of parents' chromosomes to generate offsprings' chromosomes. The fitness of new generation is expected to improve because only fit individuals participate in the reproduction. Hence the fitness of a given solution should be connected to the objective function.

The *encoding scheme* [5] takes a solution or chromosome and encodes it as a string of integers or genes. Here, a solution is an $n \times m$ Boolean storage matrix M. Each column of M is encoded by its integer. In this way a solution M is a chromosome of m integers each falls in the range between 0 to $2^n - 1$ (Figure 6-a). M_T is a subset of n integers out of m ($m \geq n$). The high *fitness* of solution M can easily be attributed to the high fitness of some sub-matrices M_T. Therefore, the fitness of the solution M can be attributed to the fitness of some sub-matrices M_T and the fitness of a subset of its column vectors which are its genes. The fitness of a solution should measure its goodness. The fitness function is the parallel memory utilization $U(M)$ which increases with increasing goodness.

	Crossover probability P_c	Mutation probability P_n	Termination condition $MaxGen$	Population size P_{size}
Typ.	0.5 to 1.0	0.01 to 0.09	prob. size	prob. size
Sui.	0.65	0.05	n+q	n+q+10

Table 1: Typical and suitable control parameters

The *initial population* must generally satisfy: (1) all possible genes are in the population chromosomes, and (2) the chromosomes are of various and diverse combination of genes. A large enough *initial population* was randomly generated and each gene was selected as a random integer from 0 to $2^n - 1$.

The *selection* is based on *survival for the fittest* which consists of keeping good genes for latter recombination by using the crossover operator. We used a *roulette wheel selection* method in which every solution is allocated a pie slice proportional to its normalized fitness. Clearly, this method guarantees that selecting a solution for reproduction is proportional to its fitness.

The *crossover* operator interchanges randomly selected substrings of parents' chromosomes to form chromosomes of offsprings. We used two methods for crossover which we call *1-cut* and *2-cut*. In *1-cut crossover* a random cut point is selected as shown in Figure 6-b. In *2-cut crossover* two random cut points are selected as shown in Figure 6-c. Notice that crossover is applied here with probability (p_c). The mutation operator consists of replacing the selected gene with a random gene corresponding to an integer between 0 and $2^n - 1$. This is shown in Figure 6-d for $n = 3$. This mutation operator was applied with probability (p_n).

Table 1 shows the typical and used (suitable) control parameters in our implementation. The initial population size P_{size} was experimentally set to $n + q + 10$. It was observed that the termination condition largely depends on the problem size. We experimentally set an upper bound on the number of iterations $MaxGen$ of our GA. The GA terminates if $MaxGen = m + q$ iterations completed, or an optimum solution is found, or the objective function did not improve by at least 5% in the past 4 consecutive iterations.

5 Evaluation

The evaluation is based on: 1) generating realistic sets of data patterns, 2) synthesizing storage matrices by using each of the proposed methods, and 3) evaluating the parallel memory utilization U for each method.

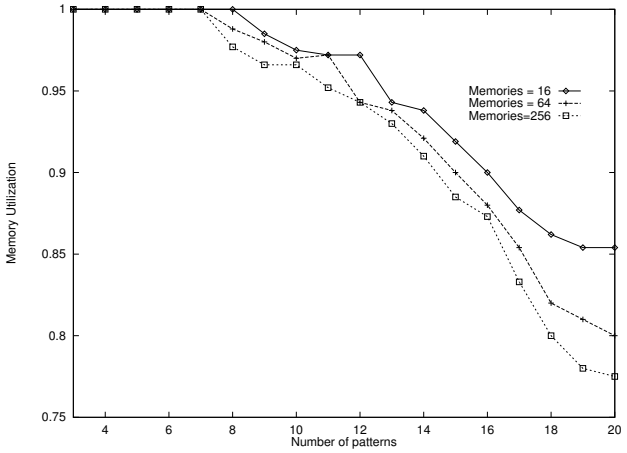


Figure 7: Utilization of parallel memories using WCNS

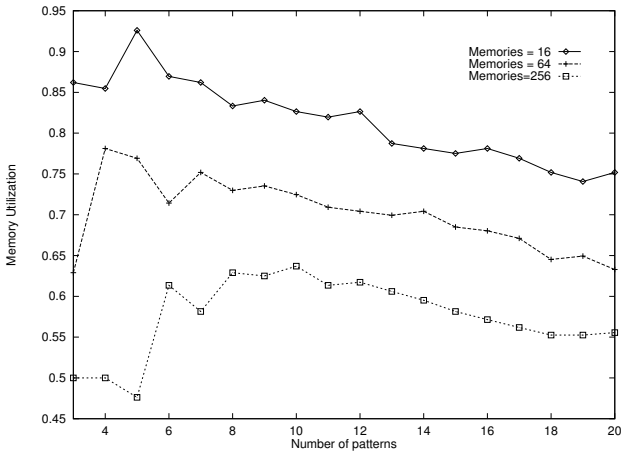


Figure 8: Utilization of parallel memories using NN-1

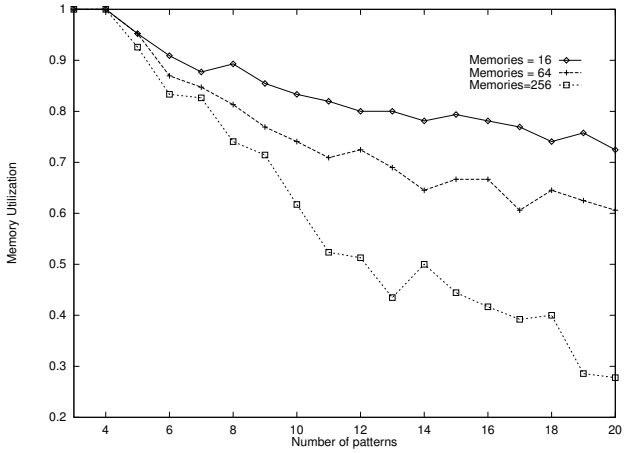


Figure 9: Utilization of parallel memories using NN-2

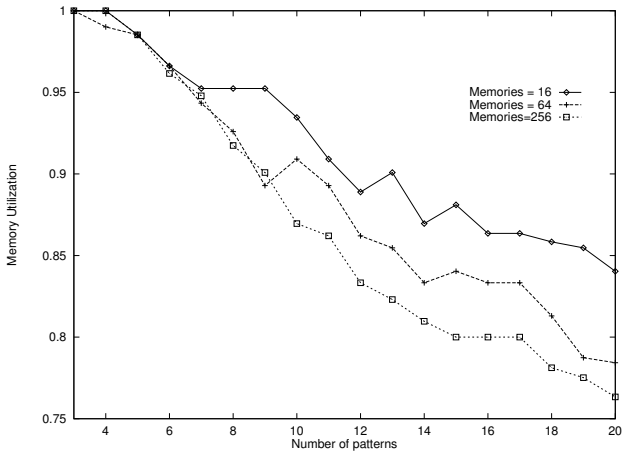


Figure 10: Utilization of parallel memories using GA 2-Cut

A problem instance is represented by tuple (n, q) which corresponds to arrays that are accessed by q data patterns in a system of 2^n memories. We study the case of 16, 64, and 256 memories and for each case we vary the number of patterns q between 3 and 20. For each instance of (n, q) we generate 50 sets of access patterns. To generate realistic access patterns we use a correlated selection function for the basis vectors of each data pattern which employs a normal distribution of basis vectors over a set of size $3n$. This promotes generation of neighboring vectors which has the effect of generating combination of patterns like rows, columns, rectangular blocks of different shapes, and power-of-2 strides. This largely covers the case of arrays referenced in loop-carried dependencies of many scientific programs.

Figures 7, 8, 9, and 10 show the parallel memory utilization for different instances of the number of patterns and number of memories. In the next subsections we evaluate the proposed synthesis methods in the case of: (1) power-of-2 access patterns, and (2) integer stride access.

5.1 Evaluation of constructive heuristics

Near optimum memory utilization U (Figure 7) was obtained when the number of data patterns was below 8 regardless of the number of memories. U smoothly decreases with increasing the number of data patterns which seems to dominate compared to variation in the number of memories. It may seem surprising, however, in our problem splitting a node increases the cost only if it causes two or more vectors in the storage matrix be linearly dependent. Splitting a node may not increase the cost but at the contrary it is likely to cause the resulting vector be linearly independent with respect to its neighbors. Assume vector w is member of two pattern bases which are ($u_1 = (1, 0, 0)$ and $u_2 = (0, 1, 0)$) and ($v_1 = (0, 1, 0)$ and $v_2 = (0, 0, 1)$). WCNS splits w and assigns it $(0, 0, 1)$ and $(1, 0, 0)$, respectively. Thus the overall assignment of w is $(1, 0, 1)$ which is linearly independent with respect to (u_1, u_2) and (v_1, v_2).

The above Figures show that memory utilization higher than 80% can be achieved by WCNS in the case of unbuffered parallel memories with an arbitrary but given set of up to 20 data patterns.

We also studied the case of integer strides. Sohi [14] proposes a storage scheme that consists of a manually synthesized 3×12 Boolean matrix which is intended to improve performance of stride access in the case of 8 memories. The use of buffers at input and output of memories smooths out the transient behavior of memories and gives near optimum memory utilization. For the case of integer stride access, we evaluated the average number of cycles for Sohi's scheme (8 memories) which is 2.43 cycles when accessing strides ranging from 1 to 64 with random starting address. The number of cycles is an indicator of the degree of parallel memory conflicts. The average number of cycles we obtained for WCNS was 2.57 cycles. Sohi's scheme has better memory utilization.

5.2 Evaluation of neural methods

The neural approach could not find storage schemes with competitive memory utilization as shown in Figures 8 and 9. NN-1 was not stable in the case of small number of patterns which is due to poor feedback and lack of data constraints (few patterns) flowing from blocks B and C to A. In this situation, NN-1 became loose and randomly behaved. The utilization generated from NN-2 was smooth from the beginning which means that the coupling among the neurons in NN-2 was sufficient to direct the network to more refined solutions. Comparing the obtained memory utilization, NN-2 outperformed NN-1 for relatively small number of patterns but the opposite was happening for relatively large number of patterns. The crossover is nearly for 10 patterns.

The performance of NN-1 and its execution time were not very sensitive to increase in the number of patterns which suggests that NN-1 could be a good approach for obtaining *fast solutions* in the case of large problems. NN-2 requires much more time than NN-1 because of its larger number of connections that must be considered in the update procedure. Moreover, NN-1 update procedure could be greatly simplified and accelerated by using *logical operators* instead of *arithmetic operators*. This can be done by Oring the outputs of neurons in one column and feeding the result to the corresponding neurons in block C. Similar approach can be used in the horizontal direction.

5.3 Evaluation of the genetic approaches

Figure 10 memory utilization for GA 2-Cut which slightly outperforms GA 1-Cut (not shown). GA 2-Cut was expected to do better because it adds more disruption and variety to population's chromosomes which is an important requirement [4] for small populations. Note that for all methods U was more sensitive to the number of patterns than to the number of memories. For large number of patterns and memories, GA 2-Cut gave the best memory utilization with smaller variance compared to all other studied methods.

For the case of stride access, the lowest average number of cycles we obtained was 2.273 cycles which was generated by using *GA 2-Cut* and the others were 2.342 from *GA 1-Cut*, 2.57 from *WCNS*, and 2.67 from *NN-1* and *NN-2*. Sohi's 3×12 Boolean matrix which is manually synthesized requires on the average 2.43 cycles. GAs may generate solutions that outperform manually optimized schemes even for small problems. This indicates that GAs can be very useful for synthesizing storage schemes for large problem instances especially in the case of programs that are compiled once and run many times over different data sets.

Both *GA 1-Cut* and *GA 2-Cut* have similar execution time which is nearly 4 times that of NN-2 and 50 times that of WCNS and NN-1.

6 Conclusion

The aim of this work is to exploit compile time knowledge (access patterns) of parallelized programs to improve bandwidth of parallel memory systems at run-time. Finding a conflict-free storage for a set of data patterns is NP-complete. This problem is reduceable to *weighted graph coloring*. The aim is to find a method for the design of efficient storage scheme that can be implemented as part of processor address translation. We investigated three methods of allocating array data to memories which are: (1) *constructive heuristic*, (2) *neural methods*, and (3) *genetic algorithms*. Simulation shows that memory utilization higher than 80% for unbuffered parallel memories can be achieved for arbitrary sets of up to 20 data patterns. Using the above array organization with buffering [14] at input and output of parallel memories may produce near optimum memory utilization. Constructive coloring heuristics are generally preferable during program development. Because of their execution time, genetic algorithms are recommended for advanced compiler optimization for synthesizing efficient storage schemes for programs that are compiled once and run many times over different data sets. One neural approach was relatively very fast in producing a reasonably good solution especially in the case of large problem sizes where genetic algorithms require excessive running time. Speeding up the convergence of the neural network can be further accelerated by using logical operators instead of arithmetic.

References

[1] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam. Compiler directed page coloring for multiprocessors. *Inter. Conf. on ASPLOS*, pages 244–255, 1996.

[2] H. G. Cragon. Memory systems and pipelined processors. *Jones and Bartlett Pub.*, 1996.

[3] A. Deb. Multiskewing-A novel technique for optimal parallel memory access. *IEEE Trans. on Parallel and Distributed Systems*, Vol 7, No 6:595–604, Jun 1996.

[4] J.L. Filho, P.C. Treleaven, and C. Alippi. Genetic-algorithm programming environments. *IEEE Computer*, 27, No 6:27–43, Jun 1994.

[5] D.E. Goldberg. Genetic algorithms in search, optimization and machine learning. *Addition-Wesley, Reading, Mass.*, 1989.

[6] J.J. Grefenstette. Genesis: a system for using genetic search procedures. *in Proc. of Conf. Intelligent Systems and Machines*, pages 161–165, 1984.

[7] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *in Proceedings of National Academy of Sciences, USA 79*, pages 2,554–2,558, 1982.

[8] K. Hwang and F. A. Briggs. Computer architecture and parallel processing. *McGraw-Hill Pub.*, 1987.

[9] Chang. P.P. Hwu, W.W. Achieving very high cache performance with an optimized compiler. *Proc. of the 16th Ann. Inter. Symp. on Computer Architecture*, pages 242–251, 1989.

[10] A.K. Jain, J. Mao, and K.M. Mohiuddin. Artificial neural networks: a tutorial. *IEEE Computer*, 29, No 8:31–44, Mar 1996.

[11] S.Y. Kung. Digital neural networks. *Prentice Hall*, 1993.

[12] S. McFarling. Program optimization for instruction caches. *Third Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, 1989.

[13] M. Saghir, P. Chow, and C. Lee. Exploiting dual data-memory banks in digital signal processors. *Inter. Conf. on ASPLOS*, pages 234–243, 1996.

[14] G. S. Sohi. High-bandwidth interleaved memories for vector processors–A simulation study. *IEEE Transactions on Computers*, 42(1):34–44, Jan 1993.