Evaluation of Neural and Genetic Algorithms For Synthesizing Parallel Storage Schemes

Mayez Al-Mouhamed * and Hussam Abu-Haimed †

Abstract

Exploiting compile time knowledge to improve memory bandwidth can produce noticeable improvements at run-time [30, 6]. Allocating the data structure [30] to separate memories whenever the data may be accessed in parallel allows improvements in memory access time of 13% to 40%. We are concerned with synthesizing compiler storage schemes for minimizing array access conflicts in parallel memories for a set of compiler predicted *data access patterns*. The access patterns can be easily found for many synchronous dataflow computations like multimedia compression/decompression algorithms, DSP, vision, robotics, etc. A storage scheme is a mapping from array addresses into storages. 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 storage scheme 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 parallel memories, i.e. a scalable parallel memory. 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 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. The solutions presented are also useful for other optimization problems.

Keywords: Heuristics, memory organization, parallel memories, performance evaluation, storage schemes

1 Introduction

There is pressing need for innovative memory architectures and organization [19, 8] to reduce bandwidth unbalancing between processor and main memory. Current memory latency and bandwidth are far from yearly recorded reduction in processor clock time and memory cost. Hierarchical memory systems overcome the latency problem by storing items in the memory level that is consistent with the frequency of their references. Thus the latency becomes

*Computer Engineering Department, College of Computer Science and Engineering, King Fahd University, Dhahran 31261, Saudi Arabia (mayez@ccse.kfupm.edu.sa).

†Department of Computer Science, Stanford University, Stanford, USA. (husam@leland.stanford.edu)

slightly slower than that of the fastest memory in the hierarchy. On the other hand, the modest improvement in memory bandwidth has come not from technology but from improved memory system design. Therefore, research on how to achieve high memory bandwidth focuses on how to map data structures onto parallel memories so as to favor a class of access patterns [8]. The aim of these methods is: 1) improving bandwidth in hierarchical memory systems, and 2) mapping of some data structures onto parallel memories.

With increasing processor speed, memory interleaving was used to provide high bandwidth through simultaneous access of consecutive addresses which fall into distinct memories. A stride memory access is a sequence of addresses $a, a+s, a+2s, \ldots, a+(2^n-1)s$, where a is the origin, s is the stride, and 2^n is the number of parallel memories. Interleaving allows conflict-free access only when the stride associated with successive references is relatively prime to the number of memories. A study of stride distribution from actual programs reveals [31] that 80% of references have stride 1, 10% with stride other than 1, and for $k \geq 1$ a stride $r2^k$ is used in 10×2^{-k}% of the cases. This explains why interleaving causes significant performance impairment due to non-uniform memory access in the case of stride and block (set of neighboring array elements) accesses. A prime number of memories [24] increases the numbers of memories and data patterns that can be accessed without conflict but finding the address is computationally expensive. Increasing the number of memories beyond the degree of interleaving (super-interleaving) [7] partially contributed in reducing memory conflicts in vector processors.

Budnik and Kuck [5] proposed *storage schemes* based on row-rotation for conflict-free access to rows, and columns of arrays. Harper [16, 17] proposed a dynamic storage scheme that optimizes access for one stride and provides increased throughput for other strides compared to low order interleaving. Buffers at memory inputs and outputs were used to reduce the effects of transient degradation in pipelined memories. To avoid run-time overhead in the above schemes, Sohi [32] proposed bit-wise Boolean address transformations for vector processors in order to determine the memory number where a given array element should be stored. The scheme can be efficiently used for power of 2 strides. It is also profitable for non power-of-2 strides. Linear transformations from the processor address to the storage location were studied in [11]. Norton [27] synthesized a bit-wise transformation matrix that allows conflict-free access to a number of power-of-2 strides. Single linear and nonlinear data patterns like diagonal and coils can be accessed without conflicts by using *multiskewing* [9] 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. Compile time reordering of the data sequences proved that compiler effort can be rewarded by noticeable improvements in memory bandwidth at run-time. Compiler optimization that attempts maximizing *temporal* and *spatial* localities and minimizing mapping conflicts produced encouraging results [26, 20]. Specifically, optimized programs for direct mapped caches resulted in lower miss ratio [26] 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 [30] 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 [6] 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. It inserts requests in the code for automatic prefetching and preferred color for each virtual page which customizes the application's page mapping policy at run-time. The SUIF compiler achieves more that 50% improvement over a standard page mapping policy. Improving scalar access in parallel memories is studied in [14]. In this case, compile-time scheduling of a very low number of data transfers demonstrated that a very high percentage of memory access conflicts can be avoided.

The hierarchical memory system is one approach to address the bandwidth mismatch problem between the processor and the memory. For the TERA MTA supercomputer [4] there is no hierarchical memory and the multithreaded execution pipelines can tolerate a 100-cycle clock latency between the logic units and the memory. It is predicted that processor-memory latency will be 10^4-10^5 cycles in the Petaflops HTMT Computer Project [33] even with the use of exotic memory/network technology. The multimedia community [22] severely criticized the hierarchical memory system because it provides only 1-D locality and it is responsible for the unpredictable memory access time due to cache misses, a feature to be avoided in dynamic media processing. There is need for new design concepts to the issue of main-memory organization and predictability of its access time.

This paper proposes a scalable main memory system provided that the data access patterns can be identified by the compiler, a feasible task for many synchronous dataflow computations like multimedia compression/decompression algorithms, DSP, Vision, Robotics, etc. Specifically, 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 compiler predicted storage schemes that minimize overall access time for arbitrary sets of data access patterns. These storage schemes make the hardware transparent to the user and avoid reorganizing the data, but require the address transformation be implemented as part of processor address translation. A general approach for synthesizing compiler storage schemes is proposed. Given a set of parallel memories, finding conflict-free storage schemes for accessing an array by using arbitrary sets of data patterns is NP-complete. Finding the storage scheme that minimizes overall access time of a given set of data patterns is reduceable to weighted graph coloring. Optimizing the storage scheme is investigated by using three different coloring approaches which are: (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 associated to multi-pattern access. Sections 4 summarizes the NP-completeness of the problem. In sections 5 we present synthesis of storage schemes by using: (1) constructive heuristics, (2) neural methods, and (3) genetic algorithms. Evaluation of synthesized storages for each case is carried out in Section 6. Conclusions and future extensions to this work are presented in Section 7.

2 Background

It is desirable to store array data that should be simultaneously accessed into different memories so that parallel access can be achieved. For loop-carried-dependency, the order of access to the array elements is generally constrained by the dependencies which occurs across the

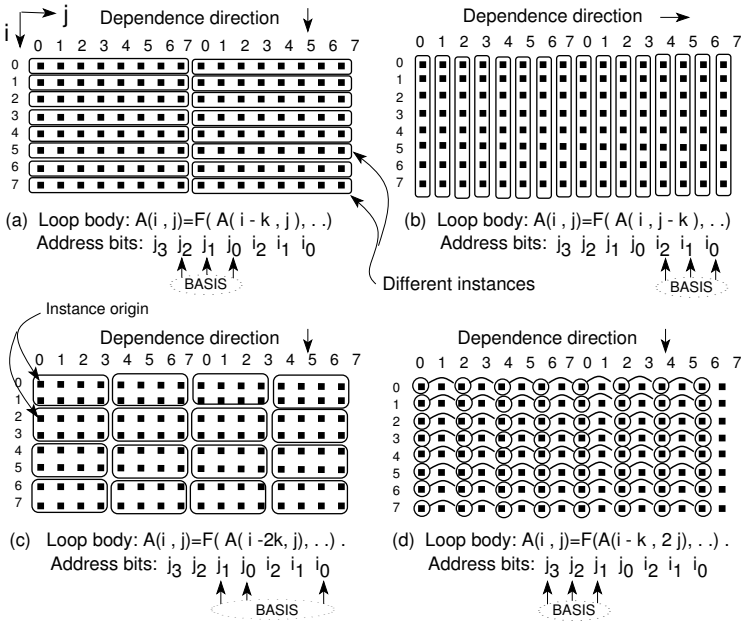


Figure 1: Example of patterns: sub-row pattern (a), sub-column pattern (b), 2×4 block pattern (c), and sub-row with stride-2 pattern (d)

iterations. A *pattern* is defined as a collection of array elements that are related to each other by some neighborhood relationship. Figure 1 shows an 8×16 array that is partitioned into sets of 8-element patterns like the row pattern (a), column pattern (b), 2×4 block pattern (c), and row with stride-2 pattern (d). To promote parallelism, the compiler restructures the loops such that at run time the data fetched from the memory allows execution of parallel threads. Thus the accessed patterns are dictated by the data dependence and compiler restructuring which are generally intended to expose inherent parallelism and to promote data locality. Figure 1 shows some access patterns that are adequate for given loop data dependencies.

The *origin* of a pattern instance is the coordinate of its upper leftmost element as shown in Figure 1. Changing the origin allows access to different instances of the pattern. The array can be seen as a collection of pattern instances. All pattern instances have the same number of elements which is also the number of parallel memories. A perfect storage scheme must allow fetching any instance of a given pattern in parallel which allows achieving linear speed up in accessing parallel memories. For example, a conflict-free storage scheme for the row and the column allows fetching any row or any column in one memory cycle. We are interested in access patterns of arbitrary dimension but having power-of-2 elements.

Consider a parallel memory that consists of $N = 2^n$ modules. Without loss of generality, the memory is considered as a two dimensional array of size $2^d \times 2^d$ such that the element in the ith row and the jth column is denoted by (i, j). We define a vector space $F = Z_2^d$ for row position. Integer i is a vector defined over $F = Z_2^d$ which is represented by $i_{d-1} \ldots i_1 i_0$. Let $B(F) = \{f_{d-1}, \ldots, f_0\}$ be the canonical basis of F. A row i is expressed as $i_{d-1} f_{d-1} \oplus \ldots \oplus i_1 f_1 \oplus i_0 f_0$. We similarly define vector spaces G for column positions with canonical bases $B(G) = \{g_{d-1}, \ldots g_1, g_0\}$. Since we use 2^n memories, we are interested in storage schemes that use n address bits out of 2^{2d} to determine a storage. For this, we define H as a vector space for memory unit numbers and its basis $B(H) = \{h_{n-1}, \ldots h_1, h_0\}$.

Addition and multiplication are modulo 2.

The cartesian product of the vector spaces F and G is another vector space $F \times G$ with basis $B(F \times G) = \{f_{d-1}, \ldots, f_1, f_0, g_{d-1}, \ldots, g_1, g_0\}$. Vector space $F \times G$ is isomorphic to Z_2^{2d}. Any location $(i, j) \in F \times G$ in memory is uniquely associated with a linear combination $i_{d-1}f_{d-1} \oplus \ldots i_0 f_0 \oplus j_{d-1}g_{d-1} \oplus \ldots \oplus j_0 g_0$ of the basis vectors of $B(F \times G)$.

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 $B(T_1) = \{g_2, g_1, g_0\}$. The components of all elements that fall into the same pattern instance cover all possible binary combinations. 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 a given pattern the address bits other than those 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 in Figure 1-(b) allows accessing sub-columns 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

In a system with 2^n memories only n address bits are needed to find the memory in which an array element is stored and other address bits are used to generate the offset. Each location $(i, j) \in F \times G$ has $2d$ components over the basis $B(F \times G)$. Denote by V a subspace of $F \times G$ whose basis $B(V)$ is formed by n vectors out of the $2d$ canonical vectors of $B(F \times G)$, where $n \leq 2d$. Denote by x is the part of (i, j) that is used in finding the memory number where element (i, j) is stored. From now on we consider vector x (also y) as the projection of some (i, j) over V.

We are interested in finding a Boolean matrix M that causes an array to be distributed over the memories for a given set of data patterns. Each array element $a(i, j)$ will be stored into memory module Mx and M will be called *storage scheme*. A necessary and sufficient condition to cause the elements of an array be distributed over the memories so that to allow parallel access to a given data pattern is that the storage scheme M is non-singular [28, 25, 1]. Consider a set $\Gamma = \{T_1, \ldots, T_q\}$ of data patterns so that each pattern consists of 2^n elements. The basis of each pattern T_k is denoted by $B(T_k) = \{t_{k,n-1}, \ldots, t_{k,0}\}$, where $t_{k,n-1}, \ldots, t_{k,0}$ are some canonical vectors chosen from $B(F \times G)$. Each $t_{k,u} \in B(F \times G)$ for $0 \leq u \leq n-1$ and $1 \leq k \leq q$ and $B(T_k) \subset B(F \times G)$. Since each pattern instance has 2^n elements, our objective is to define a storage scheme for the array $a(i, j)$ so that any pattern instance can be accessed in one memory cycle. In other words, 2^n elements of any pattern instance should be distributed over the 2^n memories [1]. Now, we define the basis for a set of data patterns in order to define a combined storage scheme.

Definition 1 *The basis B of a set $T_1, \ldots, T_q$ of data patterns is the set of all distinct canonical vectors of the bases of all patterns $T_1, \ldots, T_q$: $B = \cup_{1 \leq k \leq q} B(T_k) = \{t_{m-1}, \ldots, t_0\}$.*

The number of distinct vectors m in the union of all pattern bases always satisfies $n \leq m \leq 2d$. Consider the previously defined set of patterns $\{T_1, T_2, T_3, T_4\}$ for which $B = \cup_{1 \leq k \leq 4} B(T_k) = \{f_2, f_1, f_0, g_3, g_2, g_1, g_0\}$. The address bits over the basis vectors of a pattern must be extracted and used to select the storages that contains the pattern elements. For this, we formally define these address bits that will be used to select a unique storage for each pattern.

Definition 2 *The projection of vector $(i, j) \in F \times G$ over the basis $B(T_k)$ is defined by vector $x^{b(T_k)} = x_{n-1} t_{k,n-1} \oplus \ldots \oplus x_0 t_{k,0}$ that is formed by the components of (i, j) over the basis $B(T_k)$.*

It is important to note that each data pattern has power of 2 elements and all instances of a given data pattern are non-overlapping. Therefore, the projection of vector (i, j) over $B(T_k)$ gives the location of element (i, j) within pattern T_k and the other components of (i, j) remain constant when accessing an instance of T_k. The constant components specify the origin of the pattern instance. For example instances of T_1 can be numbered by the components of their elements (constant) over $\{f_2, f_1, f_0, g_3\}$ which are the vectors of $B - B(T_1)$. Notice that the offset of each element of a given instance of T_1 is formed by the components over $B - B(T_1)$. Moreover, the projection of all elements of an instance over the canonical basis of its pattern take all possible binary combinations. Similarly, the projection of vector x over B consists of all the address bits that contribute in the selection of the storage elements because these address bits are the only changing bits in the address when a pattern instance is accessed.

Definition 3 *The projection of vector $(i, j) \in F \times G$ over the basis B of is the vector $x^b = x_{m-1} t_{m-1} \oplus \ldots \oplus x_0 t_0$ that is formed by the components of (i, j) over the canonical vectors of B.*

Each vector (i, j) admits a projection x^b over the basis B of all data patterns. For example, vector $(i, j) = f_{d-1} i_{d-1} \oplus \ldots f_0 i_0 \oplus g_{d-1} j_{d-1} \oplus \ldots g_0 j_0$ has a projection $x^b = f_2 i_2 \oplus \ldots \oplus f_0 i_0 \oplus g_3 j_3 \oplus \ldots \oplus g_0 j_0$ over the basis $B = \cup_{1 \leq k \leq 4} B(T_k)$. In the following we define the combined storage matrix M that is used to evaluate the memory module number where array element $a(i, j)$ must be stored.

Definition 4 *The combined storage associated to all data patterns $\{T_k : 0 \leq k \leq q\}$ is a Boolean matrix M of dimension $n \times m$ such that each array element $a(i, j)$ is stored into memory location Mx^b.*

The combined storage matrix M is a collection of columns vectors $M = [C_{m-1}, \ldots, C_0]$, where C_u is an $n \times 1$ column vector. There is one-to-one correspondence between each canonical vector $t_u \in B$ and column C_u of M. The column vector C_u is the image by M of t_u for $0 \leq u \leq n$. As $B(T_k) \subset B$, then each column vector C_u of M is the image by M of some canonical vector $t_{k,u} \in B(T_k)$. For example, the combined storage matrix M for the data patterns T_1, T_2, T_3, and T_4 is a 3×7 matrix which can be arbitrarily selected as:

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

Definition 5 *The restricted matrix M^{T_k} of M to pattern T_k is defined by the m columns of M that are the images by M of all canonical vectors of basis $B(T_k)$.*

The storage matrix M is $m \times n$ and there are m columns in M_{T_k}, then M_{T_k} is an $m \times m$ matrix. For example, the restricted matrices M_{T_1}, M_{T_2}, M_{T_3}, and M_{T_4} are the following:

$$M_{T_1} = \begin{array}{c} \begin{matrix} g_2 & g_1 & g_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \quad M_{T_2} = \begin{array}{c} \begin{matrix} f_2 & f_1 & f_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \quad M_{T_3} = \begin{array}{c} \begin{matrix} g_1 & g_0 & f_0 \end{matrix} \\ \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} \end{array} \quad M_{T_4} = \begin{array}{c} \begin{matrix} g_3 & g_2 & g_1 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

Now we need to present a Lemma on M in order to cause all patterns of T be accessible in parallel.

Lemma 1 *The storage scheme M allows parallel access to a set $T = \{T_1, \ldots, T_q\}$ of data patterns if and only if the restricted matrix M_{T_k} to each pattern $T_k \in T$ is non-singular.*

Proof Consider all the 2^n elements of some pattern $T_k = \{(i, j)\}$ that should all be simultaneously accessed when a given instance of T_k is to be accessed in parallel. The projection of each element (i, j) over the basis $B(T_k)$ is defined by $x^{b(T_k)} = x_{n-1}t_{k,n-1} \oplus \ldots \oplus x_0 t_{k,0}$. When T_k is accessed in parallel, the components $x_{n-1} \ldots x_0$ take all possible combinations (2^n) and all the remaining components of (i, j) over B remain constant. Therefore, the projection of (i, j) over the basis B, which is x^b, of all the patterns can be divided into two groups as follows.

- The projection of (i, j) over the basis $B(T_k)$ of the currently accessed pattern T_k. These components take all the 2^n combinations when considering all accessed elements of T_k.
- The projection of (i, j) over the remaining canonical vectors of B $(B - B(T_k))$ which are constant when T_k is accessed in parallel.

Hence the product Mx^b can be decomposed into two terms: 1) the projection of x over $B(T_k)$ which is $x^B(T_k)$, and 2) the projection of x over the remaining canonical vectors of B, which we denote by x^r. Mx^b can then be written as:

$$Mx^b = M_r.x^r \oplus M_{T_k}.x^{b(T_k)} \quad (2)$$

The product $M_r.x^r$ yields a constant vector because x^r is the same for all the elements of T_k that are accessed in parallel. Therefore, pattern T_k can be accessed in parallel if and only if the restricted matrix M_{T_k} is non-singular. ∎

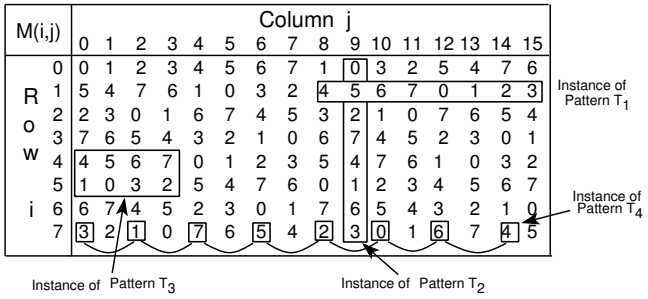


Figure 2: Mapping of array element (i, j) to memory $M(i, j)$ for 8 memories

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 $B(T_3) = \{f_0, g_1, g_0\}$. The origin can be arbitrarily chosen. Let x^b be the projection of (i, j) over $B = \cup_{1 \leq k \leq 4} B(T_k)$ 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 when accessing pattern T_3. 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:

$$M.x^b = \begin{array}{c} \begin{matrix} f_2 & f_1 & g_3 & g_2 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} . \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{array}{c} \begin{matrix} f_0 & g_1 & g_0 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \end{array} . \begin{pmatrix} i_0 \\ j_1 \\ j_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \oplus M_{T_3} . \begin{pmatrix} i_1 \\ j_1 \\ j_0 \end{pmatrix} \quad (3)$$

The parallel access to instances of T_3 only requires that M_{T_3} be non-singular. Summing a constant vector 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.

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 being the projection of (i, j) over B. The mapping is obtained from Equation 1. Here all four patterns can be accessed without conflicts because all corresponding pattern matrices are non-singular. The elements of each instance of any pattern are uniformly distributed over all memories with some skew. Four frames are shown in Figure 2, each corresponds to one pattern instance and each frame contains eight distinct memory numbers. The mapping from the array address space into memories is completely defined by the map shown in Figure 2 which identifies the array elements that fall within each memory. Hence we need additional address bits to specify the offset address where to find each array element within a memory.

In this example the address space is formed by 7 bits which are the components over B. One may decide to order the element of each memory according to a specific data pattern

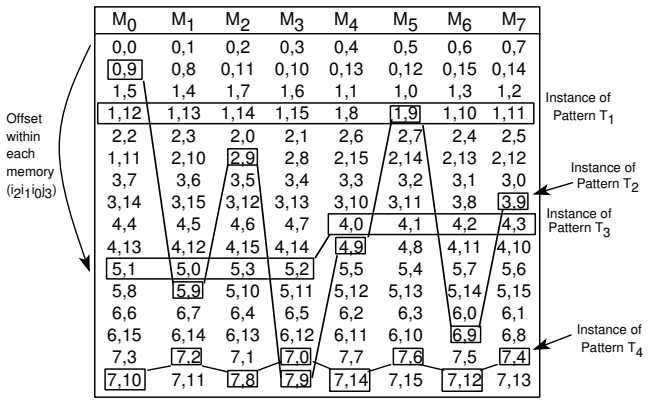


Figure 3: Storage of elements (i, j) into memories and offset

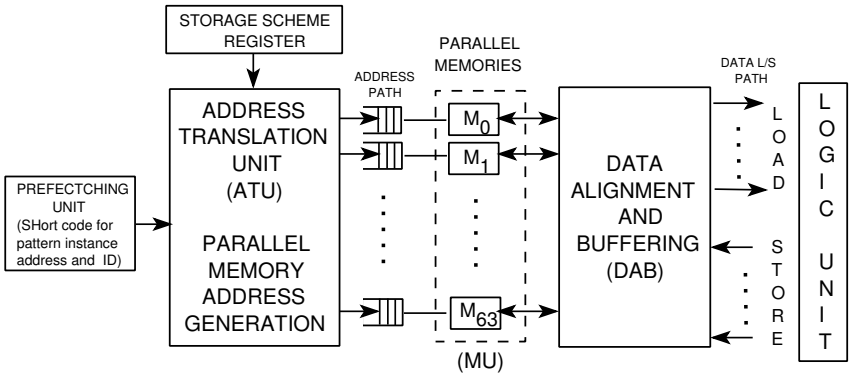


Figure 4: Model of the parallel memory system

distribution, such as T_1. If T_1 is optimally implemented (M_{T_1} has full rank) by the storage matrix M, then all array elements that belong to any instance of T_1 will be accessed with the same offset from all memories. The reason is that the offset does not include any of the bits that change (take all possible combinations) during an access to an arbitrary instance of T_1. In this case, T_1 is called *reference pattern for offset*. The *offset* is formed by all address bits except those that are in the basis of T_1. Therefore, the offset is (i_2, i_1, i_0, j_3) because $B(T_1) = \{j_2, j_1, j_0\}$. Note that other choices are also possible. It can be easily shown that all array elements mapped to the same memory fall into distinct offsets if and only if the sub-matrix of *reference pattern* 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).

The performance of a storage scheme M depends on the rank of each of its sub-matrices that are restricted to each pattern. The rank of the restricted matrix ($rank(M_T)$) to pattern T gives the number of clocks needed to access each instance of T. If M_T has full rank ($rank(M_T) = n$), then only one cycle is required for each access to any pattern instance. On the other hand, if $rank(M_T) = k < n$, then 2^{n-k} cycles will be required for accessing each pattern instance.

The parallel memory model is shown on Figure 4. The prefetching unit generates requests

for accessing pattern instances using a pattern identifier and pattern instance offset. A few registers are used to hold the storage schemes for the currently accessed arrays. The address translation unit (ATU) uses the storage scheme and pattern instance offset in generating the parallel memory requests. Each request consists of a vector of memory numbers (M), R/W, and offsets. In general, accessing a pattern instance takes 2^{n-k} memory cycles if the rank of the corresponding pattern submatrix is $k \leq n$, i.e. some memories contain up to 2^{n-k} elements of each instance of the currently accessed pattern. The ATU buffers the requests that are destined to each memory. Head-of-line requests are submitted to the memory unit (MU), an access occurs, and the output data is buffered to assemble the whole data of the accessed pattern instance. Next the data elements are alignment and delivered to the logic unit. The parallel memory organization is intended to provide fine-grain data parallelism at the level of the main memory system.

4 NP-completeness

Suppose we are given a vector space Z_2^m, a 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).

Lemma 2 *Linear independence satisfiability is NP-complete.*

Proof 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 in which each basis vector is a vertex and an edge between two vertices is created if the corresponding basis vectors are in at least one pattern basis. Here 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. We would like to know if this graph is 3-colorable. 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. ∎

We clarify the idea of the conflict graph with an example. Let $B = \{t_0 \ldots t_5\}$ and T be as defined in Figure 5-a. The conflict graph of T is shown in Figure 5-b. This graph is 3-colorable; one coloring is shown. Thus we can make a satisfying assignment to all t_x.

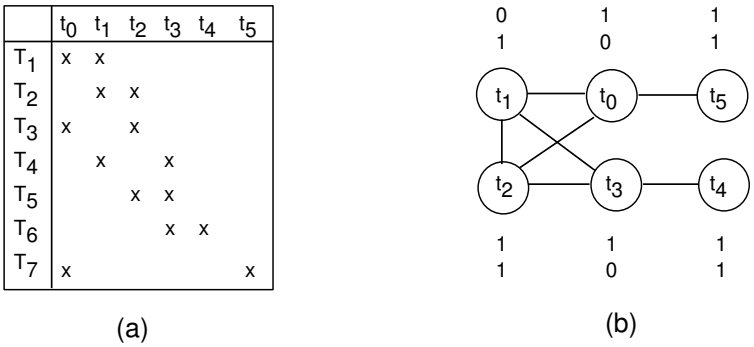


Figure 5: Conflict graph for the set of patterns

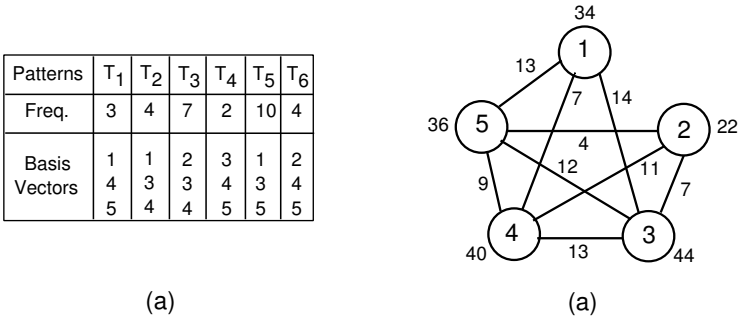


Figure 6: Set of pattern bases and their conflict graph

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.

5 Synthesis of heuristic storage schemes

To synthesize storage schemes we present three different approaches to *weighted graph coloring* which are: (1) *constructive heuristics*, (2) *neural methods*, and (3) *genetic algorithms*. We start by defining the weights and objective function.

The access frequency $f(T_k)$ of pattern T_k is the number of times a pattern is accessed during the running of the program. We extend the weight function to the edges and vertices of the conflict graph, where each vertex corresponds to a basis vector. We define the weight of an edge:

$$\omega(t, t') = \sum_{t,t' \in B(T_k)} f(T_k)$$

Each vector is associated a weight that is the sum of edge weights which link this vector to all the others:

$$\omega(t) = \sum_{t' \in B} \omega(t, t')$$

These definitions should be intuitive. The weight of an edge is proportional to the number of extra CPU cycles that will be spent if the vertices of that edge are identically colored by

assuming that all other edge constraints are met.

Consider the set of patterns $\{T_1, \ldots, T_5\}$ that are defined using their basis vectors B(T) which are denoted here by $1, \ldots, 5$ and their access frequency $\omega(T)$ as shown in Figure 6-a.

The conflict graph for the above set of patterns is shown on Figure 6-b with the weights $\omega(t, t')$ and $\omega(t)$. The union of the vectors of all the pattern bases is $B = \{1, 2, 3, 4, 5\}$.

The performance of a storage scheme M depends on the rank of each of its sub-matrices as explained in Section 3. The rank of the restricted matrix ($rank(M_T)$) to pattern T gives the number of clocks needed to access each instance of T. In general, if $rank(M_T) = k \leq n$, then $2^{n-k} f(T)$ cycles will be required for $f(T)$ accesses to distinct instances of M_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 $T_1, \ldots, T_q$. If each pattern T is accessed $f(T)$ times, then $C(M)$ is:

$$C(M) = \sum_{i=1}^{q} f(T_i) 2^{n - rank(M_{T_i})} \tag{4}$$

The function $F(M) = \sum_{i=1}^{q} f(T_i)$ represents a lower bound on the number of cycles to access all the combined patterns defined for a given storage M. Accessing an instance of a pattern requires on the average $C(M)/F(M)$ cycles which are needed to access 2^n array elements. When accessing $F(M)$ pattern instances there are $F(M)2^n$ busy memory time slots, each corresponds to one accessed array element. However, scheme M can access these $F(M)2^n$ array elements by using $C(M)2^n$ time slots. There is no guarantee that $F(M)2^n$ array elements can be accessed by an optimum storage scheme through only $F(M)$ cycles. We always have $F(M) \leq C(M)$. The performance function is $U(M) = F(M)/C(M)$ which is a lower bound on the *Parallel Memory Utilization* for the storage scheme M which requires on the average $C(M)/F(M)$ cycles to access each pattern instance.

5.1 Constructive approaches

In the following we use *weighted graph coloring* for allocating values (colors) to the basis vectors. See [14, 3, 34] for more details on coloring heuristics of *undirected conflict graphs* (UCGs). Here, the basis vectors are associated an UCG in which a node u represents a basis vector and assigned the weight $\omega(u)$. An edge (u, v) is given the weight $\omega(u, v)$ that is the extra number of memory cycles over the optimum that will be needed if u and v are given the same color. The number of nodes in the graph is m which is also the number of basis vectors in the union of all pattern bases. The number of colors is n which is also the number of vectors each pattern basis.

The degree of dependence of the pattern bases is arbitrary which indicates that the conflict graph may be formed by a collection of non-connected sub-graphs. In the following we present two constructive coloring heuristics for synthesizing storage schemes.

5.1.1 Weighted coloring with node splitting

A first coloring heuristic called *weighted coloring with node splitting* (WCNS) operates on weighted conflict graphs and performs 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

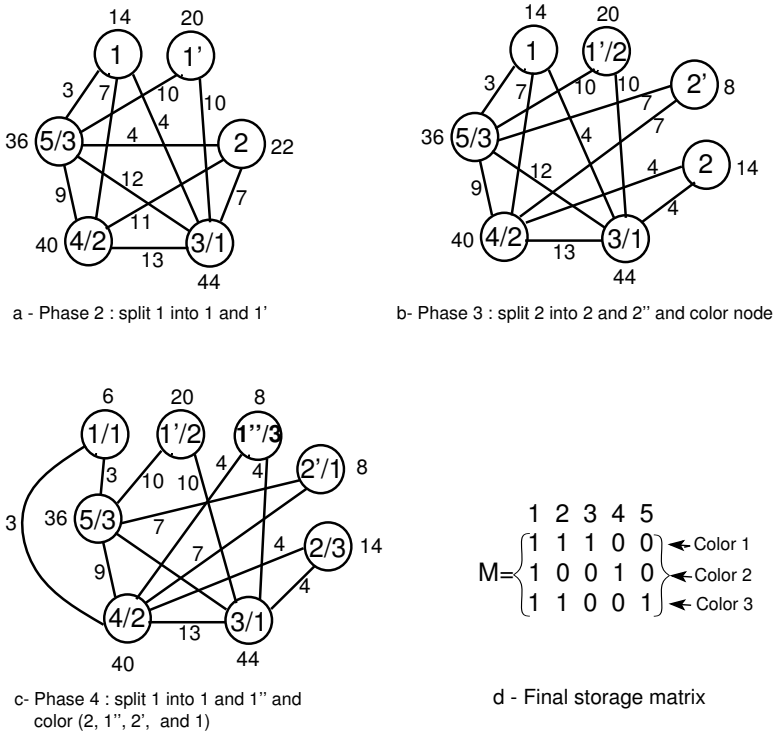


Figure 7: WCNS phases (a,b, and c) and resulting storage matrix

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 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 on 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.

Figure 7 shows the coloring by WCNS of the set of patterns displayed in Figure 6. In Phase 1, the heuristic colors nodes 3, 4, and 5 in order of decreasing weights. In phase 2, it splits node 1 into 1 and $1'$ because the neighbors of 1 have all available colors (Figure 7-a). In phase 3, it splits node 2 into 2 and $2'$ for the same reason and color $1'$ (Figure 7-b). Finally, in phase 4 (Figure 7-c) the heuristic splits 1 again into 1 and $1''$ which makes all the remaining nodes colorable. The resulting storage matrix is shown in Figure 7-d.

We analyse the time complexity of WCNS. In the worst case we have a fully connected graph with m nodes and each has $m-1$ edges, where m is the number of distinct vectors in the basis of all patterns. An upper bound on the number of splits of a node with $m-1$ edges is $m-n$, where n is number of colors. Each split requires updating of the graph with a cost of $O(m)$. Therefore, the time complexity of WCNS is $O(m^3 - m^2 n)$.

5.1.2 A clustering-based heuristic

The clustering (CA) heuristic considers each pattern basis and uniformly distributes its vectors over a set of clusters, where each cluster is associated to a color. The distribution is uniform because distinct pattern basis vectors are assigned to distinct clusters. To minimize the conflicts, a vector is mapped to the cluster whose conflict with the vector is the least among all clusters. The conflict between a vector and a cluster depends on the vectors which are already assigned to the cluster. In the following we present this heuristic in more detail.

Initially, for each color j ($0 \leq j \leq n-1$) we create an empty cluster C_j. The pattern bases are sorted in the decreasing order of their weights and the pattern basis B_T with the highest weight is taken first.

Let's $B_T = \{v_0, \ldots, v_{n-1}\}$ be the current pattern basis. The algorithm evaluates the conflict array $conf(i,j)$ for each $v_i \in B_T$ and each cluster C_j, where $conf(.,.)$ is an $n \times n$ array. The value of the array conflict $conf(i,j)$ is the sum of conflicts between vector v_i and all the vectors $\{e_{j,0}, e_{j,1}, \ldots\}$ that are members of cluster C_j:

$$conf(i,j) = \sum_{e_{j,k} \in C_j} \omega(v_i, e_{j,k})$$

Since the clusters are initially empty, we set $\omega(v_i, \emptyset) = 0$ and $\omega(v_i, v_i) = 0$. Now, the basis vectors $\{v_0, \ldots, v_{n-1}\}$ are taken in the decreasing order of their weights and the vector v_i that has the highest weight is taken first. Vector v_i is copied into cluster C_j if $conf(i,j)$ is the least among all the n clusters. Next, cluster C_j is locked to ensure that no other vector from the same pattern basis will be copied into C_j. Each cluster receives at most one vector of each pattern basis. The above process is repeated for all the vectors of the current pattern basis which leads to distribute its vectors over the clusters. The algorithm terminates when all the pattern bases have been visited.

As a result of this heuristic each cluster contains the basis vectors that have the least degree of conflict. The heuristic is useful to achieve two objectives: 1) minimizing the intra-cluster conflicts which is likely to cluster independent vectors, and 2) reducing duplication of copies of the same vector in distinct clusters to increase linear independence.

Each cluster c_i should map to one row of the storage matrix M. Since each column j of the above matrix corresponds to one vector e of the union of all the pattern bases because such a column is the image by M of e ($j = Me$). The storage matrix M is formed by examining each cluster so that the ith row of M is formed by 1 in each column j such that $t_j \in c_i$, otherwise the row is completed with zeros.

As an example consider the previous set of 6 patterns shown in Figure 6. There is one step for each pattern as shown in Figure 8. The clusters are initialized with the pattern basis that is the most frequently accessed which is T'_5 and the clusters become $c_0 = \{1\}$, $c_1 = \{3\}$, and $c_2 = \{5\}$. Next, in each step a pattern is considered, the corresponding conflict table is evaluated, and the basis vectors are clustered. In Figure 8 we circled the cost of retained mappings of vectors to clusters.

The final clusters are $\{1,4\}$, $\{3,5\}$ and $\{1,2,5\}$ which shows that 1 and 5 are duplicated in the solution which is equivalent to node splitting. Vectors of each cluster will receive the cluster color. As vector may appear in more than one cluster, the final coloring of such vector is the sum (exclusive-or) of all its cluster colors. Clusters $\{1,4\}$, $\{3,5\}$ and $\{1,2,5\}$ are assigned the coloring $(1,0,0)$, $(0,1,0)$, $(0,0,1)$, respectively. Since vector 1 is in the first

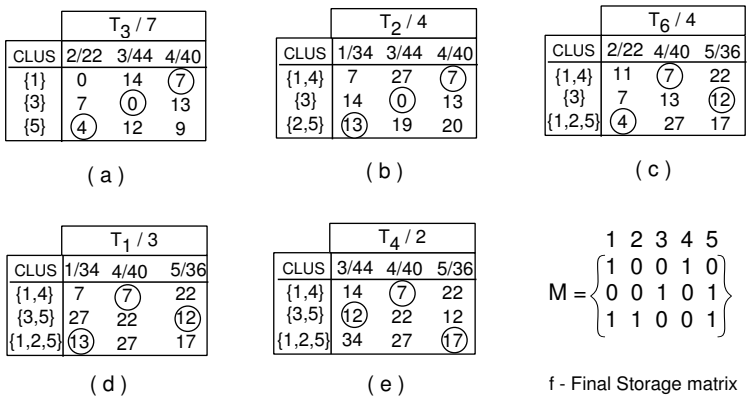


Figure 8: Clustering phases (a, b, c, d, and e) and resulting storage matrix

and last clusters which causes vector 1 to be colored as $(1, 0, 1)$. Similarly, vector 5 is colored $(0, 1, 1)$. Figure 8-f shows the corresponding storage matrix M which is non-singular for all the patterns.

We analyse the time complexity. Building the weight matrix for q patterns takes $O(m^2nq)$ and that of the conflict table is $O(n^2mq)$. Mapping qn vectors to clusters takes $O(n^2q)$. Therefore, the time complexity of clustering is $O(m^2nq + n^2mq)$.

5.2 A neural approach

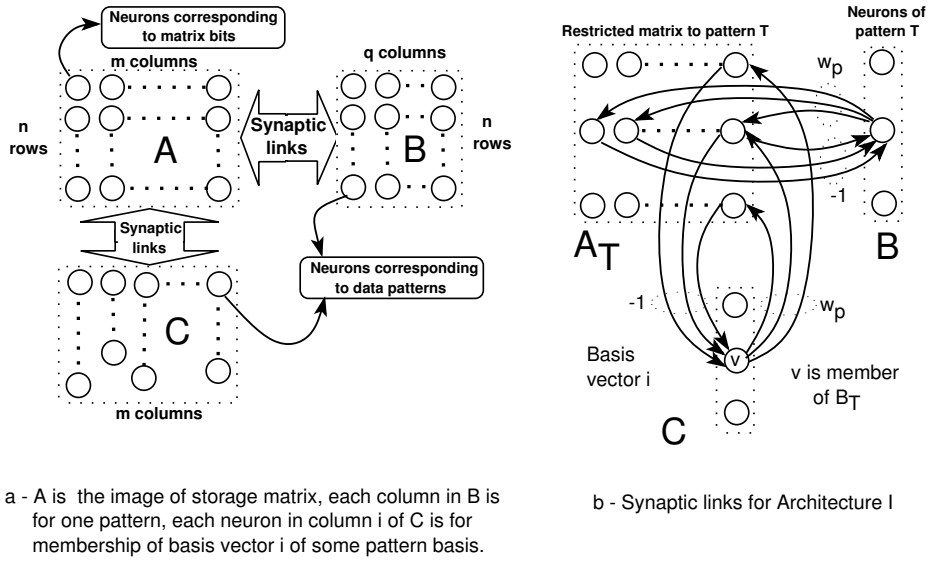
A neuron consists of a cell body and two tree-like links called *dendrites* and *axon* [23]. A neuron receives signals from other neurons through its dendrites. Axon is a long link that terminates into strands. To deliver the signal from one neuron to another, strands meet with dendrites. The junction is called *synapse* which can be *excitatory* or *inhibitory* with respect to the output of connected neuron.

An artificial *Neural Network* (NN) is a collection of interconnected neurons; each has a number of input synapses, a body, and output synapse. A weight is assigned to each input synapse. The neuron body performs a summation operation of the weighted input synapses, filter the sum using a non-linear function (F), and outputs the result. Mainly, the non-linear function is a unit step function with threshold (θ). Formally, 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 n_i is $v_i = F(\sum_{j=1}^{N} w_{j,i}v_j - \theta_i)$, where N is the number of neurons.

The NN architecture most commonly used in combinatorial optimization problems [23] is the *feedback network* which is used with *Hopfield* network model [18]. In the above model, the feedback causes the network to iteratively produce transient solutions before reaching a stable state.

Hopfield showed that a network energy function ($E = -\frac{1}{2}\sum_i \sum_j w_{i,j}v_iv_j$) will be in a local optima when the network converges [18, 21]. 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.

Designing a NN which ensures the restricted matrices are all non-singular is very complex. We use more flexible conditions but 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



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 9: Neural network NN-1

values to neighboring components of distinct vectors. The idea is to promote generation of vectors that are linearly independent. In the following we present two implementations of the NN which are called *neural network 1* (NN-1) and *neural network 2* (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 in figure 9-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. 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.

All synaptic connections are inter-block as shown in Figure 9-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 is the pattern access frequency $w(T)$. 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

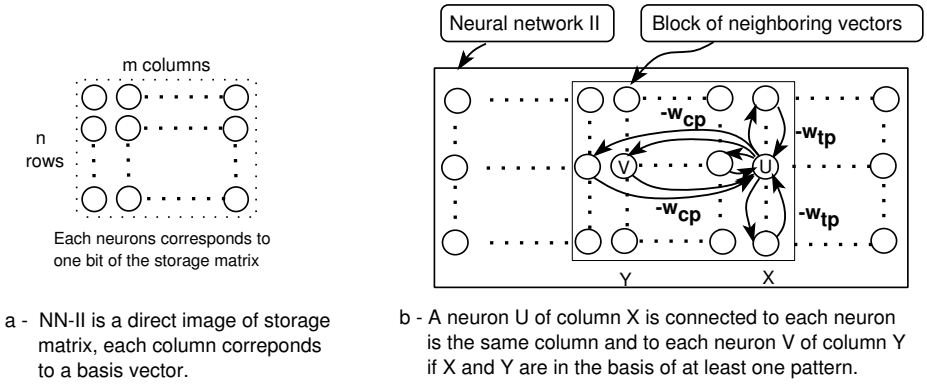


Figure 10: Neural network NN-2

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. Term N^2 is the cost of the loop and term kN is the cost of searching next neuron. 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 10-a shows NN-2. In NN-1 similar action was generated, but with external forces. In NN-2 there is only one block which corresponds 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 are available. Having

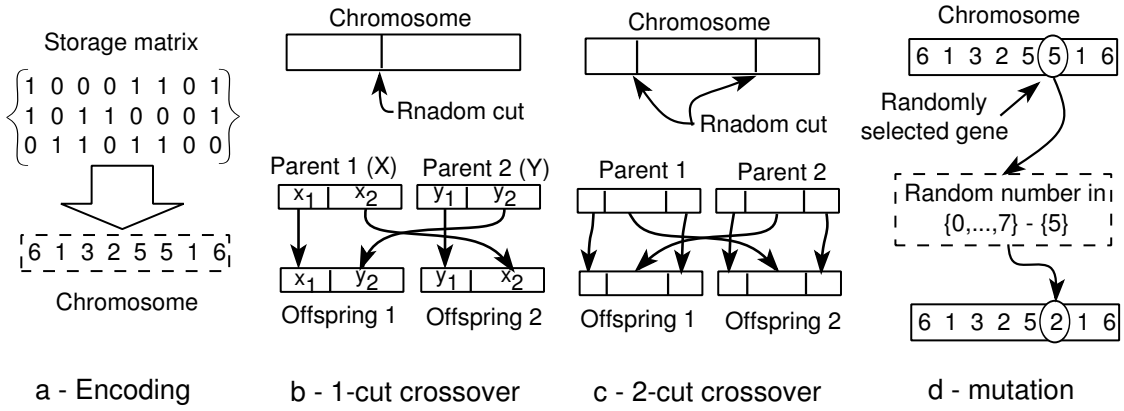


Figure 11: GA: encoding of solution (a), 1-cut crossover (b), and 2-cut crossover (c)

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 10-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.

5.3 A genetic approach

Genetic algorithms (GAs) [10] are directed random search strategies used in solving optimization problems. Fit individuals survive and produce next generation while weak individuals extinct. Hence, good genes of fit individuals survive and genetic evolution takes place. GA operates on sample solutions from diverse areas of the search space and then gradually intensifies the search in promising areas. A GA is based on (1) random search to explore different areas of the solution space, and (2) directing and narrowing the search through the use of probabilistic selection of intermediate solutions. GAs use the crossover operator as the main search and mutation as secondary operator.

Genetic evolution [13] 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.

A GA requires encoding of the solution as chromosomes to allow application of genetic operators like crossover and mutation. The steps of one iteration of GA are: (1) random selection of an initial set of solutions and evaluation of their fitness, (2) probabilisetic selection of a subset of solutions, each with a probability equal to its fitness, and (3) production of new offsprings through application of crossover and mutation operators over initial solutions. In mutation, a randomly selected gene of the offsprings chromosome is inverted. This process continues in a loop until some conditions are met. GAs employ the following operators: 1) the

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

Table 1: Typical and suitable control parameters

encoding scheme, 2) the fitness function, 3) the initial population, 4) the selection mechanism, 5) the crossover operator, 6) the mutation, and 7) control and termination conditions.

To be successful an *encoding scheme* must be clearly manifested in the solution genes [12]. A solution or chromosome is encoded as a string of integers or genes. It is highly preferable if we can attribute the fitness of a given solution to a subset of its genes with some high probability. Here, a solution is an $n \times m$ Boolean storage matrix M with 2^n and 2^m being the number of memories and number of distinct basis vectors for all patterns, respectively. 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 $2^n - 1$. This is shown in Figure 11-a. A restricted sub-matrix M_T of M 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 where T represents one of the patterns having relatively high access frequencies. Moreover, the high fitness of M_T is due to the value of its rank which means that a large number of its vectors are linearly independent. 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. Recall the performance function of storage scheme M that is the *memory utilization* defined by $U(M) = F(M)/C(M)$, where ratio $C(M)/F(M)$ is the average number of cycles needed to access each pattern instance. The fitness function is the parallel memory utilization $U(M)$ which increases with increasing goodness.

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. The *initial population* was randomly generated and each gene was selected as a random integer from 0 to $2^n - 1$. Since the initial population is influenced by the random number generator, we generated large enough initial population to cover all possible genes and enough chromosome combinations of the genes.

The *selection* is based on *survival for the fittest* which consists of keeping good genes for latter recombination by using the crossover operator. Having large number of offsprings in the new population which are taken from a good solution is likely to increase the chances of getting the right combination of an optimum solution in one of these offsprings. Therefore, the number of offsprings taken from a parent solution is proportional to its fitness. For this we used a *roulette wheel selection* method in which every solution is allocated a pie slice proportional to its normalized fitness. The roulette is spin, rotate for a random angle, and stops at one solution slice. 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. Selecting large number of offsprings increases the chance of taking the substring most responsible for the high fitness of a solution. We used two methods for crossover which we call *1-cut* and *2-cut*.

In *1-cut crossover* a random cut point is selected between 0 and $m-1$, where m is the length of each chromosome. This divides (see Figure 11-b) two chromosomes $X = (X_1, X_2)$ and $Y = (Y_1, Y_2)$, where X_1 and Y_1 have same number of genes. Crossover generates two offsprings $O_1 = (X_1, Y_2)$ and $O_2 = (Y_1, X_2)$. Each of O_1 and O_2 has m genes. In *2-cut crossover* two random cut points are selected as shown in Figure 11-c. Notice that crossover is applied here with probability (p_c) which means that two parents can be included in the new generation without being modified.

Since the crossover operator does not create new chromosomes but only recombines existing chromosomes, the *mutation* operator creates new genes to avoid that unique gene be lost for ever. Generally, the mutation of a randomly selected gene consists of complementing its binary and creating an inverted gene. To add more diversity in the search process, we used another mutation operator that 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 11-d for $n = 3$. This mutation operator was applied with probability (p_n). It gave better results than the inversion.

Table 1 shows the typical and used (suitable) control parameters in our implementation. The population size should be set so that enough randomness and diversity in chromosomes and genes is present. For large n there is large number of possible genes that should exist in the population. Also large values of q cause large variation in fitness of different combination of genes which requires large population to cover these combinations. This explains why the initial population size P_{size} was experimentally set to $n + q + 10$ for the class of studied problems. 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.

6 Evaluation

The evaluation is based on: (1) generating realistic sets of data patterns, (2) synthesizing storage matrices by using each of the proposed methods, (3) evaluating the parallel memory utilization U (see Section 5) for each method by simulating the parallel memory depicted on Figure 4, and (4) comparing to other results. The latencies of the simulated parallel memory are: (1) 2 clocks for the ATU, (2) 20 clocks for the MU, and (3) 1 clock for the DAB.

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 discrete normal distribution over a set of $3n$ basis vectors. This promotes generation of neighboring basis vectors which causes the generated patterns to be among rows, columns, rectangular blocks of different shapes, and power-of-2 strides in each direction. For example, the 12 patterns shown on Figure 12 for 16 memories. These are used in bitonic sorting (patterns (a)-(e)), LU and matrix (patterns (a) and (l)), and vision and multimedia neighborhood operators (patterns (f)-(l)) like the *linear-shift invariant* and *weighted median*

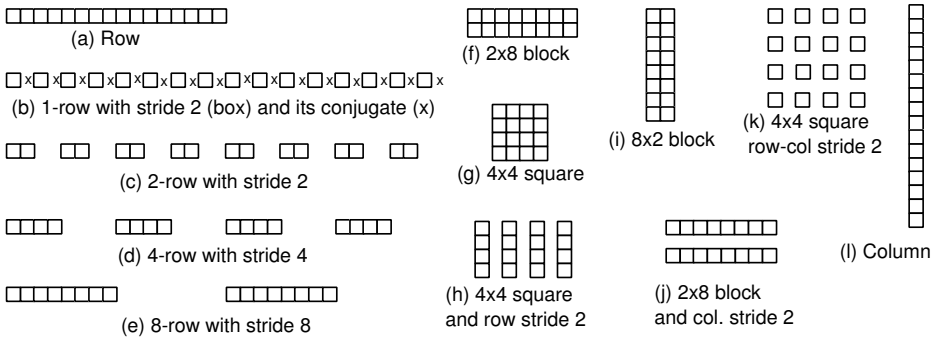


Figure 12: Some access patterns used in sorting, LU, matrix, and multimedia

[15, 29, 22]. For example the bitonic sorting operators [29] require partial sorting of: (1) groups of 2 adjacent elements (pattern (b) and its conjugate denoted by x), (2) groups of 4 adjacent elements (pattern (c) and its conjugate), (3) groups of 8 adjacent elements (pattern (d) and its conjugate), etc. Figure 12 assumes 16 memories and 16 logic units.

The pattern shown in Figure 12-(b) is used for load/store of the data in sorting groups of 2 adjacent elements. Each group consists of a left element (marked by a square frame) and a right element (marked by a neighboring x). All left elements can then be accessed in parallel in one cycle and all right elements are accessed in the next cycle. Thus two cycles are needed to load all the logic units, each with two neighboring elements. After sorting the logic units store the data by using the same access pattern. Bitonic sorting may efficiently use the access patterns shown on Figure 12 (a)-(e). The generated linear access patterns largely cover the case of arrays referenced in loop-carried dependencies of many scientific programs as well as mainstream vision and multimedia parallel algorithms.

The use of correlated basis vectors for each pattern introduces correlation among the patterns which makes the problem of finding conflict-free storage harder and exposes the synthesis methods to realistic situations.

Figures 13, 14, 15, 16, 17, and 18 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.

6.1 Evaluation of constructive heuristics

Figures 13 and 14 show the memory utilization U for WCNS and CA versus the number of patterns. Near optimum results were 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.

WCNS outperforms CA in all cases of power-of-2 access patterns. 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. Neighboring vectors which appear at least in one pattern basis are not given the same color by WCNS and one of these vectors will be split if no color is available. 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.

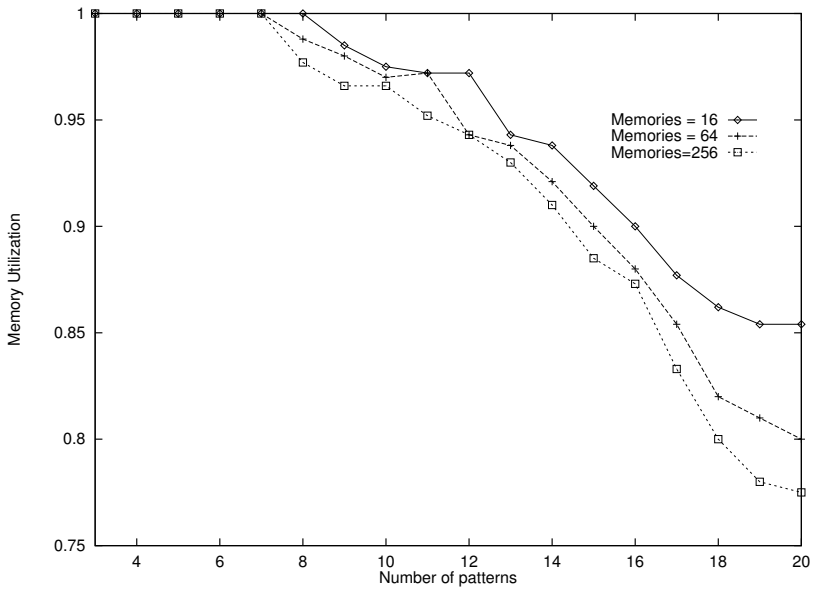


Figure 13: Utilization of the parallel memory system using weighted coloring with node splitting

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).

However, WCNS increases the graph size and execution time after each split. CA has the least execution time because of its directed approach which is also responsible for its relatively low memory utilization. Memory utilization of CA is comparable to the *semiperfect storage* [2] but WCNS improves memory utilization by up to 17% compared to the above two schemes. The semiperfect scheme is a two step simple coloring which was proposed to show potential performance improvement of pattern-oriented storage schemes compared to that of interleaved memories and row-column static storage. The above Figures show that memory utilization higher than 80% can be achieved by WCNS in the case of parallel memories with an arbitrary but given set of up to 20 data patterns.

We also studied the case of integer strides. Sohi [32] 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. We also used CA and WCNS to synthesize storage matrices for the same problem. The average number of cycles we obtained for CA and WCNS were 2.561 and 2.57 cycles, respectively. Sohi's scheme has better memory utilization.

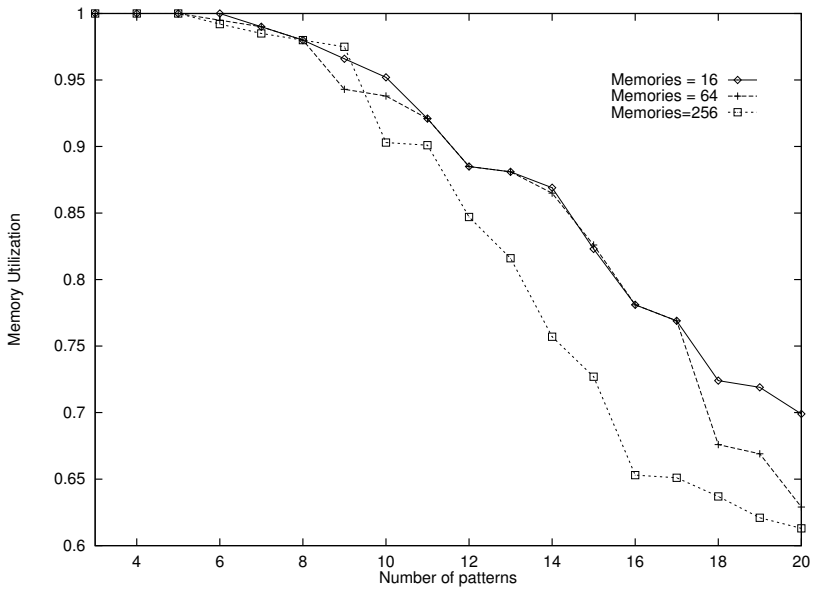


Figure 14: Utilization of the parallel memory system using clustering

6.2 Evaluation of neural methods

The neural approach could not find storage schemes with competitive memory utilization as shown in Figures 15 and 16. 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.

The memory utilization obtained by using NN-1 can be made more acceptable especially in the number of patterns is large enough to tighten the convergence. The clear advantage of NN-1 is its fast convergence. Though NN was not successful in finding competitive solutions compared to the other studied approaches, it is still attractive in the sense that it is a totally different approach, it has fast convergence, and can be designed by using only logic operators.

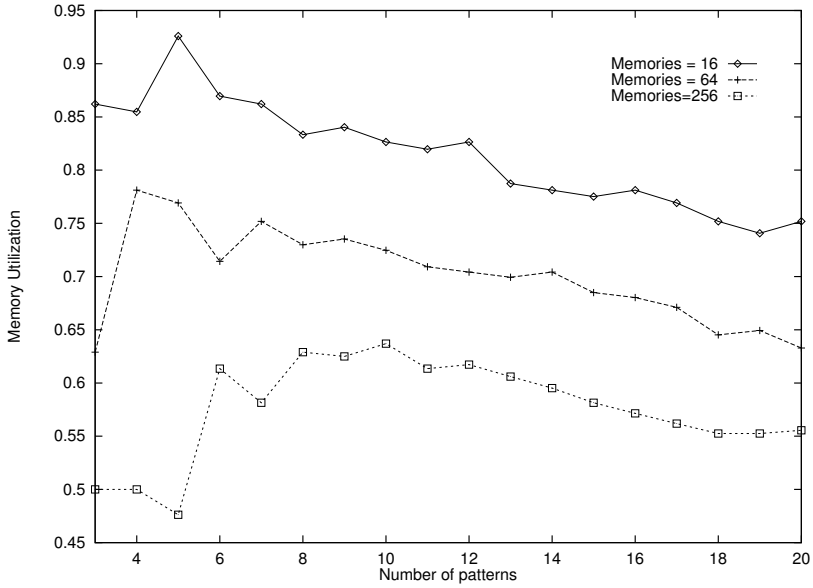


Figure 15: Utilization of the parallel memory system using neural network 1

6.3 Evaluation of the genetic approaches

The memory utilization for the storage schemes that are synthesized by using GAs are shown in Figures 17 and 18. GA 2-Cut slightly outperforms GA 1-Cut. GA 2-Cut was expected to do better because it adds more disruption and variety to population's chromosomes which is an important requirement [10] for small populations. Note that for all studied algorithms (WCNS, CA, NN, and GA) the memory utilization of synthetic pattern access was more sensitive to increase in the number of patterns than increase in number of parallel memories. As shown in the previous plots WCNS performs slightly better than GAs for relatively small number of patterns and nearly the same as GAs for the other case. For large number of pattern and memories, GA 2-Cut gave higher average 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*, 2.561 from *CA*, 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 when accessing strides between 1 and 64 with random origin. Sohi showed that due to buffering at input and output of parallel memories the efficiency of his scheme is very close to 1. 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 static schemes which are intended to be used many times.

Both *GA 1-Cut* and *GA 2-Cut* have similar execution time (see Figure 19) which is nearly 4 times that of NN-2 and 50 times that of CA, WCNS, or NN-1.

The GA gave the highest memory utilization in both cases of pattern access and stride access for relatively large number of patterns and memories. Due to its relatively high execution time, GA may be adequate as an advanced compiler optimization option for synthesizing efficient storage schemes. This is particularly useful for programs that are compiled once and run many times over different data sets.

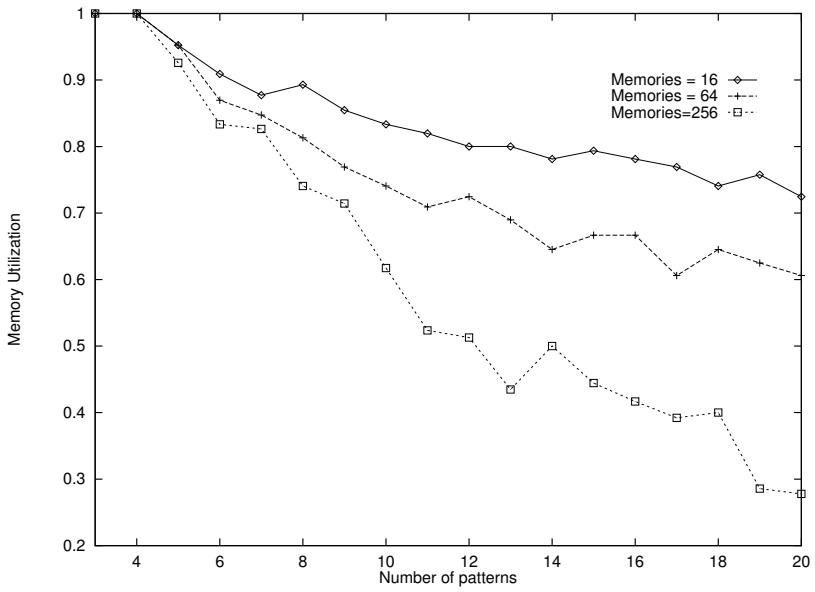


Figure 16: Utilization of the parallel memory system using neural network 2

7 Conclusion

This paper proposes a scalable memory system provided that the data access patterns can be identified by the compiler. Static access patterns can be easily found for many synchronous dataflow computations like multimedia compression/decompression algorithms (motion estimation DCT, FFT, Entropy scan, and interpolation) as well as many other applications in DSP, Vision, Robotics, etc. The aim of this work is to exploit compile time knowledge of parallelized programs to improve the bandwidth of parallel memory systems at run-time. Finding a conflict-free storage scheme for a set of data patterns is NP-complete. This problem is reduceable to *weighted graph coloring*. We investigated three methods of allocating array data to memories to reduce parallel memory access time of a set of *access patterns* that can be predicted at compile time. We used three different coloring approaches based on: (1) *constructive heuristics*, (2) *neural methods*, and (3) *genetic algorithms*. Simulation shows that parallel memory utilization higher than 80% can be achieved for arbitrary sets of up to 20 data patterns. Constructive coloring heuristics are generally preferable during program development. Because of their execution time, genetic algorithms are recommended for advanced compiler optimization for synthesizing the most efficient storage schemes especially for large problem sizes. Programs that are compiled once and run many times over different data sets benefit the most from highly optimized storages. The convergence time of proposed neural algorithm seems to be only slightly dependent on problem size. 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 operators which may greatly simplify its architecture.

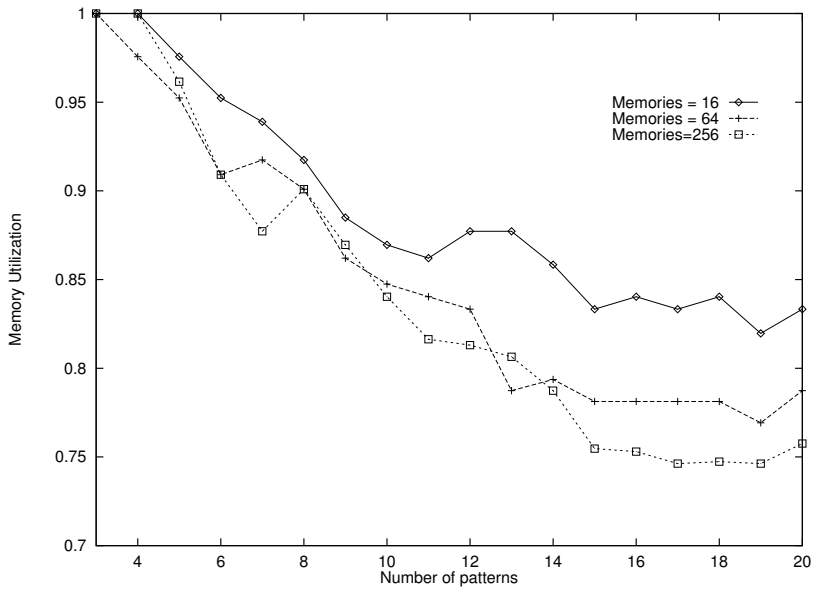


Figure 17: Utilization of the parallel memory system using genetic algorithm 1-Cut

8 Acknowledgment

Thanks to the College of Computer Science and Engineering, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for its computing facilities and its support for conference attendence.

References

[1] M. Al-Mouhamed and S. Seiden. Minimization of memory and network contention for accessing arbitrary data patterns in SIMD systems. *IEEE Trans. on Computers*, 45:6:757–762, Jun 1996.

[2] M. Al-Mouhamed and S. Seiden. A heuristic storage for minimizing access time of arbitrary data patterns. *IEEE Trans. on Parallel and Distrituted Systems*, Vol 8, No. 4:441–447, Apr 1997.

[3] A. Blum. New approximation algorithms for graph-coloring. *Journal of the ACM*, Vol 41, No 3:470–516, 1994.

[4] P. Briggs and J. Feo. The Tera Programming Workshop. *Inter. Conference on Parallel Architectures and Compilation Techniques,, Paris, France*, October 1998.

[5] P. Budnik and D. Kuck. The organization and use of parallel memories. *IEEE Transactions on Computers*, C-20(12):1566–1569, Dec 1971.

[6] 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.

[7] T. Cheung and J.E. Smith. A simulation study of the CRAY X-MP memory system. *IEEE Trans. on Computers*, C-35, No 7:613–622, Jul 1986.

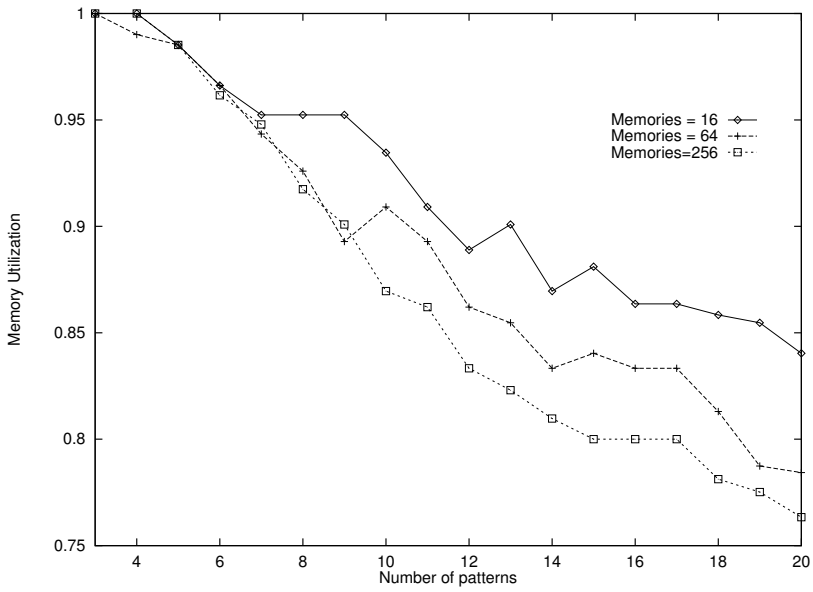


Figure 18: Utilization of the parallel memory system using genetic algorithm 2-Cut

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

[9] 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.

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

[11] J. M. Jalby W. Frailong and J. Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *Proceedings of the International Conference on Parallel Processing*, pages 276–283, 1985.

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

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

[14] R. Gupta and M. L. Soffa. Compile-time techniques for improving scalar access performance in parallel memories. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):138–148, Apr 1991.

[15] R. M. Haralick and L. G. Shapiro. Computer and robot vision. *Addison Wesley*, 1992.

[16] D. T. Harper III. Block, multistride vector, and FFT accesses in parallel memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):43–51, Jan 1991.

[17] D. T. Harper III. Increased memory performance during vector accesses through the use of linear address transformations. *IEEE Transactions on Computers*, 41(2):227–230, Feb 1992.

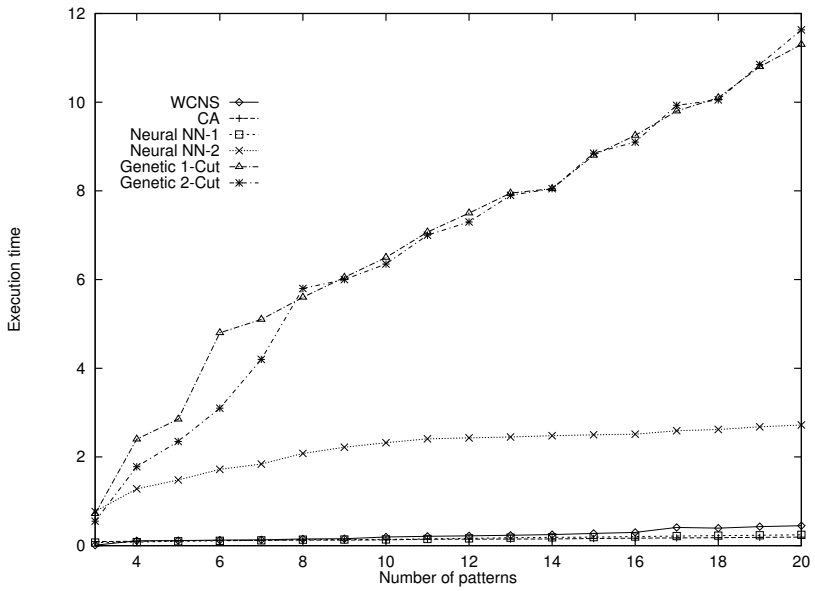


Figure 19: Execution time of WCNS, CA, NN-1, NN-2, GA 1-Cut, and GA 2-Cut for 256 memories

[18] 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.

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

[20] 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.

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

[22] C. E. Kozyrakis and D. A. Patterson. A new direction for computer architecture research. *IEEE Computer*, Nov. 1998.

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

[24] D. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, Dec 1975.

[25] K.Y. Lee. On the rearrangeability of a (2log$N$-1) stage permutation network. *IEEE Trans. on Computers*, Vol 34, May 1985.

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

[27] A. Norton and E. Melton. A class of boolean linear transformations for conflict-free power-of-two stride access. In *Proceedings of the International Conference on Parallel Processing*, pages 247–254, 1987.

[28] M. C. Pease. The indirect binary-cube microprocessor array. *IEEE Trans. on Computers*, C-26, No 5:458–473, May 1977.

[29] M. Quinn. Designing efficient algorithms for parallel computers. *McGraw-Hill Inter., Second Edition*, 1988.

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

[31] A. Seznec and J. Lenfant. Interleaved parallel schemes. *IEEE Trans. on Parallel and Distrituted Systems*, Vol 5, No. 12:1329–1334, Dec 1994.

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

[33] T. Sterling. A hybrid technology multithreaded computer architecture for petaflops computing. *MS 159-79, J.P.L., California Institute of Technology*, January 1997.

[34] X. Zhou, S.-I. Nakano, and T. Nishizeki. Edge-coloring partial k-trees. *IEEE Trans. on Parallel and Distributed Systems*, Vol 21, No 3:598–617, Nov 1996.