A Cost-Effective Heuristic Storage For Minimizing Access Time of Arbitrary Data Templates

Mayez A. Al-Mouhamed and Steven S. Seiden

ICS-UCI Technical Report 93-30

June 18, 1993

A Cost-Effective Heuristic Storage For Minimizing Access Time of Arbitrary Data Templates

Mayez A. Al-Mouhamed[*] and Steven S. Seiden[†]

ICS-UCI Technical Report 93-30

June 18, 1993

Abstract

The serialization of memory accesses is a major limiting factor in high performance SIMD computers. For these machines, the data templates that are accessed by a program can be perceived by the compiler, and therefore, the design of conflict-free storage schemes may dramatically improve performance.

The problem of finding storage schemes, with minimum hardware requirements, for accessing a set of arbitrary templates is proved to be NP-complete. To design cost-effective storage schemes, we introduce two parameters: the number of 1's in the storage matrix (affecting hardware complexity) and the access frequency of each template. Heuristics are proposed to find storage schemes with minimum hardware (*Perfect Schemes*) but without enforcing a high degree of conflict reduction. Another heuristic is proposed to augment perfect storage schemes by using minimum additional hardware in order to reduce the degree of conflict (*Semi-Perfect Schemes*).

Experimental evaluation is carried out using a Monte Carlo simulation. Performance of the proposed heuristics is compared to solutions obtained using branch-and-bound search. Results show that perfect-schemes may deviate on the average by 20% from the optimum access time in the case of 10 arbitrary templates and 16 memories. However, semi-perfect schemes lead to dramatic reduction of the degree of conflict compared to perfect-schemes. The proposed heuristic storage outperforms row-major interleaving and row-column-diagonals storage. The time complexity of the proposed heuristics is $O(p(t+n)+n^2t)$, where t, 2^p, and n, are the number of templates, the number of processors, and the number of distinct vectors of the template bases, respectively.

Keywords: Heuristics, NP-completeness, parallel memories, performance evaluation, storage schemes

[*]Department of Information and Computer Science, University of California, Irvine, CA 92717. On leave from King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia.

[†]Department of Information and Computer Science, University of California, Irvine, CA 92717.

1 Introduction

Non-uniform memory access is frequently encountered in high-performance computers. The serialization of memory accesses is a major limiting factor to bandwidth balancing between processor and main memory. With increasing processor speed, memory interleaving has been used as a simple storage scheme to map consecutive addresses into different physical memories so that simultaneous accesses can be performed in one memory cycle.

Interleaving allows conflict-free access only when the stride associated with successive references is relatively prime to the number of memories. Increased numbers of patterns can be conflict-free accessed when the number of memory modules is prime [1]. Unfortunately, the address transformation becomes computationally expensive[2].

Budnik and Kuck [3] proposed storage schemes based on row rotation which have been statically implemented on SIMD computers [4, 1] for conflict-free access to arbitrary row, column, and both types of diagonals of arrays. Using linear arddress transformation, multi-stride vector access has been proposed [2] for vector-computers by factoring the stride into two integers, the first is a power of 2 and the other is relatively prime to 2. Conflict-free access is obtained for that stride. The resulting storage outperforms the regular interleaving for arbitrary strides provided that a buffer is used for every memory. Access to interleaved memories in vector processors can also be improved [5] by using non-singular skewing-matrices.

Improving the performance of scalar access to parallel memories has also been addressed [6] for long-instruction-word (LIW) computers. LIWs have multiple functional units that operate in lock-step and fetch data required by parallel operations from multiple memory modules. Conflict-free allocation of data may not always exist, but a high percentage of memory conflicts can be avoided [6] by using a very low degree of duplication.

In applications such as numerical analysis and vision processing, many other data patterns than the rows and columns are frequently accessed. These are applications that require the highest execution speed. Because the knowledge of patterns or templates is within the capability of the compiler, storage schemes that offer higher flexibility have been proposed [7, 8] as dynamic linear transformations from the processor address to the storage location. A necessary and sufficient condition [7] for conflict-free access of combined templates is that the image by the storage scheme of the basis of each template should be linearly independent. As the interconnection network should provide data alignment [9] between processors and memories, other constraints can then be used for finding the storage matrix.

The data templates that are accessed by a program can be mapped by a dynamic storage scheme that minimizes the overall memory conflicts for a set of given templates. Dynamic storage schemes make the hardware transparent to the user and avoid reorganizing the data, but require the address transformation be implemented as part of every processor hardware to increase concurrency.

We are concerned with dynamically reconfigurable storage schemes for SIMD models that minimize the overall access time for an arbitrary set of weighted data templates. We are proposing procedures to automate the process of finding combined storage schemes that minimize the hardware cost of implementing the storage scheme. The derivation of combined storage schemes is based on two newly introduced factors: the number of 1's in the storage matrix and the frequency of access of each template. The first factor controls overall hardware complexity. The second factor establishes a precedence among templates. Given a program that requires access to a set of arbitrary weighted templates, the problem is to find a cost-effective storage scheme that minimizes the degree of conflict, i.e. minimum access time.

This paper is organized as follows. Section 1 presents the notation and assumptions used. In sections 3 and 4, we define the bases associated with a given template and the storage matrix (XOR-scheme), respectively. Sections 5 and 6 introduce perfect and non-perfect XOR-schemes, respectively. Section 7 treats the NP-completeness of the problem. In sections 8 and 9, semi-perfect schemes and heuristics are presented, respectively. Evaluation of heuristics is carried out in Section 10. Conclusions and future extensions to this work are presented in Section 11.

2 Background

Consider an SIMD computer that consists of a number of processing elements and memory units connected by a network. Any processor can access any memory unit through an interconnection network. We assume that there are an equal number P of processors and memory units, and that P is a power of two 2^p. If more than one processor tries to access a location in a given memory unit, during a given instruction cycle, a *conflict* occurs. If i processors all try to access a given memory unit during the same cycle, it takes i cycles for the memory unit to serve them. Since all processors run in lock-step, the entire computation is dramatically slowed. It would be desirable to store the data that should be simultaneously accessed into different memories so that parallel access to all items can be achieved.

Suppose that we know a priori the memory access patterns of a given program. We assume that the data to be accessed is a two dimensional array. A *template* is a pattern of array elements. A particular instantiation of a template is called a *template instance*, and the upper left-most element of a template instance is called the *origin* of the template instance. Template instances are non-overlapping. We would like to find a function that allows us to access all templates in a *conflict free* manner. By this, we mean that for all given templates, for all template instances, all of the elements of any template instance map to distinct memory units. For instance, in a matrix multiply algorithm, we would like to have conflict free access to all rows, and all columns. The set of all rows (columns) is a template, and each row (column) is a template instance. We want to allocate the array among the memory units in such a way that no two elements of the same row are in the same memory unit, and no two elements of the same column are in the same memory unit. Such an allocation is called a *storage scheme* [2, 10, 7].

We make several simplifying assumptions which do not affect the generality of our discussion. First, the memory is a single two dimensional array such that the element in the ath row and the bth column is denoted by (a, b). The upper left-most element is $(0, 0)$. Second, the sizes of the horizontal and vertical dimensions are both 2^d. By convention, the array is always indexed by a variable a in the vertical direction, and a variable b in the horizontal direction.

The binary representation of an integer x is $x_{d-1} \dots x_1 x_0$. In other words, the zeroth bit (least significant) of x is x_0, the first bit is x_1 etc....

A row position a can also be thought of as a vector, over the finite field $\mathbb{Z}_2$, the integers modulo 2. In $\mathbb{Z}_2$, addition corresponds to logical **exclusive or**, and multiplication corresponds to logical **and**. $(a_0, a_1 \dots a_{d-1})$ is the vector representation of a, in terms of the bits of a. We define a vector space $\mathcal{F} = \mathbb{Z}_2^d$ for horizontal position. Let $F = \{f_0, f_1, \dots f_{d-1}\}$ be the canonical basis of $\mathcal{F}$. i.e. $f_0 = (1, 0, 0 \dots 0)$, $f_1 = (0, 1, 0 \dots 0) \dots f_{d-1} = (0, 0, 0 \dots 1)$. Each row has a unique representation as a vector in $\mathcal{F}$. A Row a is expressed as $a_0 f_0 \oplus a_1 f_1 \oplus \cdots \oplus a_{d-1} f_{d-1}$ in terms of F. For example, 11 in binary is 1011, and so the vector representation of row 11 is $(1, 1, 0, 1)$. Expressed as a linear combination of the basis F, row 11 is $f_0 \oplus f_1 \oplus f_3$. We similarly define vector spaces $\mathcal{G}$ for column positions and $\mathcal{H}$ for memory unit numbers, with canonical bases $G = \{g_0, g_1, \dots g_{d-1}\}$ and $H = \{h_0, h_1, \dots h_{p-1}\}$, respectively.

The Cartesian product of the vector spaces $\mathcal{F}$ and $\mathcal{G}$ is a new vector space $\mathcal{V} = \mathcal{F} \times \mathcal{G}$ with basis $\{f_0, f_1, \dots f_{d-1}, g_0, g_1, \dots g_{d-1}\}$. Let $n = 2d$. We denote this combined basis as $V = \{v_0, v_1 \dots v_{n-1}\}$, where $v_0 = f_0$, $v_1 = f_1$, $v_d = g_0$ etc.... This vector space is isomorphic to $\mathbb{Z}_2^n$. Any location (a, b) in memory is uniquely associated with a linear combination of the basis elements $a_0 v_0 \oplus \cdots \oplus a_{d-1} v_{d-1} \oplus b_0 v_d \oplus \cdots \oplus b_{d-1} v_{n-1}$. Adding two vectors in $\mathbb{Z}_2^n$ corresponds to bitwise **exclusive or**. Multiplying a vector by the scalar 1 gives back that vector, while multiplying a vector by the scalar 0 results in the zero vector. We refer to the elements of the basis V using either f's and g's, or v's, depending on which is notationally convenient.

3 Templates

We define a template $\mathcal{T}_i$ by a basis T_i, which is a nonempty subset of V. Notice that there is a definite distinction between the definition of a template $\mathcal{T}_i$, which is a set of sub-arrays, and its basis T_i, which is a set of vectors. We assume all templates bases are of size p. This is best explained by looking at some examples. Let p and d both be 3. Our basis is $V = \{f_0, f_1, f_2, g_0, g_1, g_2\}$, or alternatively $V = \{v_0, v_1 \dots v_5\}$. Consider the template $\mathcal{T}_1$ defined by $T_1 = \{f_0, f_1, f_2\}$. The set of templates instances described are all non-overlapping columns of eight elements, the upper left-most template instance having origin $(0, 0)$. Every element in a template instance is a linear combination $a_0 f_0 \oplus a_1 f_1 \oplus a_2 f_2 \oplus b_0 g_0 \oplus b_1 g_1 \oplus b_2 g_2$, where the b's are constant, and the a's are allowed to vary ($b_0 g_0 \oplus b_1 g_1 \oplus b_2 g_2$ is the template instance's origin). Intuitively, we are letting the three least significant bits of a vary, while the other bits of a, and all bits of b, remain fixed. By allowing different bits to vary, we generate templates of different shapes. Let $\mathcal{T}_2$ have basis $T_2 = \{f_0, f_1, g_1\}$. Then, in $\mathcal{T}_2$ all template instances are four elements tall. Since g_0 is omitted, this template skips a column. Thus, we have two 4×1 sub-arrays, spaced two columns apart. We define $\mathcal{T}_3$ by $T_3 = \{f_1, f_2, g_1\}$. This template is four 1×2 sub-arrays spaced two rows apart. We let $\mathcal{T}_4$ have basis $T_4 = \{f_0, f_1, g_0\}$. It is a 4×2 sub-array. All of these templates are illustrated in Figure 1.

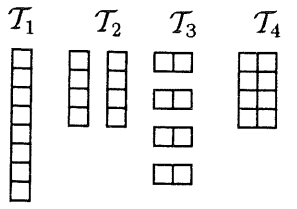


Figure 1: From left to right $\mathcal{T}_1 \dots \mathcal{T}_4$

4 XOR-Schemes

An XOR-scheme is a linear function $\phi : \mathcal{F} \times \mathcal{G} \mapsto \mathcal{H}$. The function ϕ is represented by a $p \times n$ matrix, which we denote Φ. We apply ϕ to a vector X by matrix multiplication:

$$\phi(x_0, x_1 \dots x_{n-1}) = \Phi \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

The ith entry of the jth column of Φ is $\Phi_{i,j}$. (The upper left-most entry is $\Phi_{0,0}$.) We denote the ith column of Φ by $\Phi_{*,i}$, and the ith row of Φ by $\Phi_{i,*}$. The columns of this matrix represent the values, in terms of the basis H, of ϕ on the members of the basis V. I.e. $\Phi_{*,i}$ is the value of $\phi(v_i)$.

We can also consider ϕ as an ordered set of p functions, $\{\phi_0, \phi_1 \dots \phi_{p-1}\}$, mapping from $\mathcal{F} \times \mathcal{G}$ to $\mathbb{Z}_2$, where $\phi(X) = \phi_0(X)h_0 \oplus \phi_1(X)h_1 \oplus \cdots \oplus \phi_{p-1}(X)h_{p-1}$. The matrix of ϕ_i is $\Phi_{i,*}$.

Then ϕ allows conflict free access to $\mathcal{T}_i$, if and only if ϕ maps each linear combination of T_i to a unique element of $\mathcal{H}$. Since ϕ is linear, all translations of these linear combinations are also conflict free. In other words, if for one template instance X in $\mathcal{T}_i$, ϕ restricted to X is one to one, then for all template instances X in $\mathcal{T}_i$, ϕ restricted to X is one to one.

5 Perfect XOR-Schemes

We say that an XOR-scheme is *perfect* if and only if all columns $\Phi_{*,i}$ contain at most one non-zero entry. In other words, in the expression $\phi(X) = \phi_0(X)h_0 \oplus \phi_1(X)h_1 \oplus \cdots \oplus \phi_{p-1}(X)h_{p-1}$, any particular x_i is used at most once. Finding a perfect XOR-scheme is desirable, because such a scheme requires minimum hardware to be implemented.

We give an example using the templates of Section 3. The bases of these templates are shown in Table 2. (An 'x' indicates a vector's inclusion in a template basis.) We have $n = 6$ and $p = 3$. We would like to find a perfect XOR-scheme for this set of templates. But first, let us consider the subset of templates $\{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3\}$. Notice that f_1 appears in all templates bases of this subset. Therefore, let $\phi_0(a_0, a_1, a_2, b_0, b_1, b_2) = a_1$. Further, notice that g_0 appears only in T_3 and f_0 appears only in T_1 and T_2. Therefore, let $\phi_1(a_0, a_1, a_2, b_0, b_1, b_2) = b_0 \oplus a_0$. Finally, notice that g_1 appears only in T_2 and f_2 appears only in T_1 and T_3. We let $\phi_2(a_0, a_1, a_2, b_0, b_1, b_2) = b_1 \oplus a_2$. Let us generalize this procedure.

	f_0	f_1	f_2	g_0	g_1	g_2
T_1	x	x	x			
T_2	x	x			x	
T_3		x	x	x		
T_4	x	x		x		

Figure 2: Templates bases $T_1 \ldots T_4$

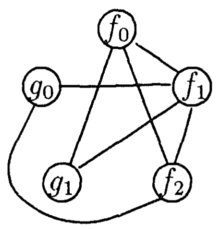


Figure 3: Conflict graph for T_1, T_2, T_3

Theorem 5.1 *Let Φ be the matrix of a perfect XOR-scheme. If $\Phi_{k,i} = 1$ and $\Phi_{k,j} = 1$ where $i \neq j$, and v_i and v_j are both in T_ℓ, then $\mathcal{T}_\ell$ cannot not be accessed conflict free.*

Proof This is easily seen by a counting argument. ϕ_k can only take on two values. x_i and x_j can each take on two values, for a total of four. Further, because each column of Φ contains at most one non-zero, no other ϕ_k varies with x_i or x_j. Therefore, given a template instance, we are mapping two elements of it to each memory unit in its image. ■

Let $T : V \mapsto 2^T$ be a function mapping from a given basis vector to the set of templates bases which contain that vector. More precisely, $T_i \in T(v)$ if and only if $v \in T_i$. We allow $\Phi_{k,i} = 1$ and $\Phi_{k,j} = 1$ only if $T(v_i)$ and $T(v_j)$ are disjoint. To represent this relationship, we create a graph (V, E), called the *conflict graph* of the template set. The vertices of the graph are the vectors of the basis V. An edge (v_i, v_j) is in E if and only if $i \neq j$ and $T(v_i) \cap T(v_j) \neq \varnothing$. The graph for T_1, T_2, and T_3 is illustrated in Figure 3.

Theorem 5.2 *For a set of templates $\mathcal{T}$, a conflict free XOR-scheme for P memory units exists, if and only if the conflict graph of the templates is p-colorable.*

Proof Suppose the conflict graph is p-colorable. Then let $\Phi_{i,j} = 1$ if and only if vertex v_j has color i. ϕ is perfect and conflict free, since for all i, if $\Phi_{i,j} = 1$ and $\Phi_{i,k} = 1$, then v_j and v_k cannot be in the same template. Conversely, given a conflict free XOR-scheme ϕ, color the conflict graph of its templates by giving a vertex v_i the color j if and only if $\Phi_{j,i} = 1$. No vertex is assigned more than one color, because each column of Φ contains at most one non-zero entry. No two adjacent vertices are assigned the same color, because this would imply that the XOR-scheme is not conflict free. ■

Indeed, the graph in Figure 3 is 3-colorable. The matrix of the perfect XOR-scheme is:

$$\begin{array}{c} \\ \Phi = \end{array} \begin{array}{c} \begin{array}{cccccc} f_0 & f_1 & f_2 & g_0 & g_1 & g_2 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

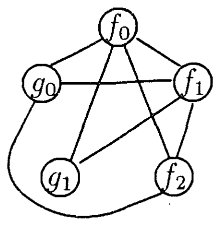


Figure 4: Conflict graph for $T_1 \ldots T_4$

However, consider what happens when T_4 is added. The graph of $T_1 \ldots T_4$ is shown in Figure 4. This graph is not 3-colorable, because g_0, f_0, f_1 and f_2 form a clique, and thus no perfect XOR-scheme exists.

We now prove:

Theorem 5.3 *Finding a perfect XOR-scheme for a set of templates and arbitrary p is NP-complete.*

Proof It is simply shown that finding a perfect XOR-scheme is in NP. We non-deterministically choose were to place the one in each column, and verify that the resulting matrix maps each template conflict free. Graph coloring is NP-complete [11]. Suppose we are given an arbitrary graph (V, E). We construct a set of templates as follows. For each vertex v_i in V of degree zero, create a template containing only the vector v_i. All other vertices are covered by some edge. For each edge (v_i, v_j), create a template basis containing only v_i and v_j. (V, E) is the conflict graph of the set of templates thus constructed. We have already shown that the conflict graph of a set of templates is p-colorable if and only if there is a perfect XOR-scheme for P memory units. If we have n vertices in the graph the number of templates is $O(n^2)$, thus finding a graph coloring is polynomially reducible to finding a perfect XOR-scheme. ∎

Note that two-coloring is polynomial, and thus finding a perfect XOR-scheme for 4 memory units is tractable.

6 Non-Perfect XOR-schemes

Suppose we do not restrict ourselves to perfect XOR-schemes. Does the set of templates in Figure 2 have an XOR-scheme? By inspection, we find the matrix of one possible XOR-scheme is:

$$\Phi = \begin{array}{c} \begin{array}{cccccc} f_0 & f_1 & f_2 & g_0 & g_1 & g_2 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Why does this function allow conflict free access? Consider ϕ restricted to the template $\mathcal{T}_1$. The matrix of this restricted function is:

$$\begin{array}{c}
\begin{array}{ccc} f_0 & f_1 & f_2 \end{array} \\
\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}
\end{array}$$

Notice that this matrix has rank 3; its columns are linearly independent. The dimension of the image of ϕ restricted to T_1 is three, and therefore, conflict free access is assured. We denote ϕ restricted to T_i by $\phi(T_i)$. The matrix of $\phi(T_i)$ is $\Phi(T_i) = (\Phi_{*,j})_{v_j \in T_i}$. The matrices of ϕ restricted to $T_2 \ldots T_4$ are, respectively:

$$\Phi(T_2) = \begin{array}{c}
\begin{array}{ccc} f_0 & f_1 & g_1 \end{array} \\
\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}
\end{array}$$

$$\Phi(T_3) = \begin{array}{c}
\begin{array}{ccc} f_1 & f_2 & g_0 \end{array} \\
\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}
\end{array}$$

$$\Phi(T_4) = \begin{array}{c}
\begin{array}{ccc} f_0 & f_1 & g_0 \end{array} \\
\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}
\end{array}$$

all of which have rank 3. In general, we need to find a function ϕ with matrix Φ, such that $\Phi(T_i)$ has rank $|T_i|$, for all templates $\mathcal{T}_i$. How can such XOR-schemes be found? First consider a more general problem.

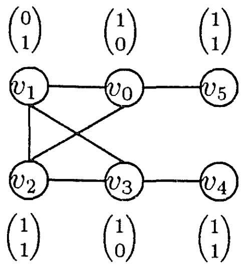
7 A General Problem

Suppose we are given:

- A vector space $\mathcal{Z} = \mathbb{Z}_2^p$.
- A set of variables $V = \{v_0, v_1, \ldots v_{n-1}\}$.
- A set $T = \{T_1, T_2, \ldots T_t\}$, where each T_i is some member of 2^V. Each variable must appear in some T_i.

The problem is to assign each variable a value in $\mathcal{Z}$, such that for all i, the vectors assigned to the variables in T_i are linearly independent. We call this problem *linear independence satisfaction* (LIS).

Theorem 7.1 *LIS is NP-complete.*



	v_0	v_1	v_2	v_3	v_4	v_5
T_1	x	x				
T_2		x	x			
T_3	x		x			
T_4		x		x		
T_5			x	x		
T_6				x	x	
T_7	x					x

Figure 5: $T_1 \ldots T_7$ and their independence graph

Proof Consider the case where $p = 2$ and $|T_i| = 2$ for all T_i in T. We call this problem 2-LIS. The vectors in $\mathbb{Z}_2^2$ 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 $\mathcal{Z}^* = \{z_1, z_2, z_3\}$. Any two distinct members of $\mathcal{Z}^*$ are linearly independent. Therefore, for each $T_i = \{v_j, v_k\}$ we must assign to v_j and v_k distinct members of $\mathcal{Z}^*$. We call (V, T) the *independence graph* of T. Each vertex in the graph is a variable v_i, and there is an edge between v_i and v_j if and only if $\{v_i, v_j\}$ is in T. We can solve 2-LIS if and only if the independence 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 (v_i, v_j), we create a $T_k = \{v_i, v_j\}$. We use the algorithm for 2-LIS to assign each v_i a value in $\mathcal{Z}^*$. 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 independence graph with an example. Let $V = \{v_0 \ldots v_4\}$ and T be as defined in Figure 5. The independence graph of T is also illustrated in this figure. This graph is 3-colorable; one coloring is shown. Thus we can make a satisfying assignment to all v_i.

Our proof that LIS is NP-complete does not depend on the fact that $\mathcal{Z}$ is over $\mathbb{Z}_2$. LIS is NP-complete over any finite field.

It is easily seen that LIS is equivalent to finding a non-perfect XOR-scheme. Note that finding a general XOR-scheme for 4 memory units is NP-complete, where finding a perfect XOR-scheme for 4 memory units is polynomial. Also note that we can build independence graphs only for $p = 2$. We need to find a model of XOR-schemes for $p > 2$, from which good heuristics can be derived.

8 Semi-Perfect XOR-Schemes

We investigate a class of XOR-schemes which are somewhere between perfect and general XOR-schemes. We call this class of XOR-schemes *semi-perfect.* We say that an XOR-scheme ϕ, represented by a matrix Φ, is semi-perfect if and only if for all templates $\mathcal{T}_i$ in $\mathcal{T}$, the matrix of ϕ restricted to $\mathcal{T}_i$ contains at most one column with two non-zero entries, and the rest of the columns have one or zero non-zero entries.

Theorem 8.1 *Let Φ be a matrix over $\mathbb{Z}_2$ with at most one non-zero entry in each column. Let Φ' be defined by:*

$$\Phi'_{i,j} = \begin{cases} 1 & i = c \text{ and } j = k \\ \Phi_{i,j} & \text{otherwise} \end{cases}$$

for some fixed c and k. Then $rank(\Phi') \geq rank(\Phi)$.

Proof If $\Phi_{c,k} = 1$ then $\Phi' = \Phi$, and therefore $\text{rank}(\Phi') = \text{rank}(\Phi)$. Otherwise, if some other column $\Phi'_{*,x}$ has a non-zero in row c, then add $\Phi'_{*,x}$ to $\Phi'_{*,k}$ giving a new matrix Φ'', which has the same rank as Φ'. Since this other column must have at most one non-zero entry, the only change to Φ'' will be that $\Phi''_{c,k} = 0$. Now $\Phi'' = \Phi$, and therefore $\text{rank}(\Phi'') = \text{rank}(\Phi)$. If no other column $\Phi_{*,x}$ has a non-zero in row c, then the rank of Φ' is one greater than that of Φ. ∎

Given a perfect XOR-scheme for a set of templates (in which some templates are not accessed conflict free), we can use the preceding theorem to create a semi-perfect XOR-scheme with a decreased number of conflicts, by selectively adding ones to its matrix. In fact, the example given in Section 6 is a semi-perfect XOR-scheme. We call this process of selectively adding ones *augmenting.*

Given a perfect XOR-scheme which is not conflict free, we want to know if it can be augmented to a conflict free semi-perfect scheme. Unfortunately, answering this is NP-complete. Each column $\Phi_{*,i}$ will either be augmented or not. For each template $\mathcal{T}_j$ at most one column of $\Phi(\mathcal{T}_j)$ may be augmented. Given these restrictions, we wish to find a subset of columns to augment such that all $\Phi(\mathcal{T}_j)$ are augmented. For $p = 3$ this problem is exactly ONE-IN-THREE SAT, which is NP-complete [12].

9 Heuristic Approaches

Since no good algorithm is known for any NP-complete problem, we are justified in proposing heuristics for finding XOR-schemes. It is reasonable to assume that if the memory access patterns of a program are known, the access frequency to the given patterns will also be

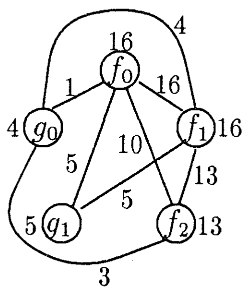


Figure 6: Weighted conflict graph

known. For instance, given the templates introduced in Section 3, suppose that $\mathcal{T}_1$ is accessed 10 times, $\mathcal{T}_2$ is accessed 5 times, and $\mathcal{T}_3$ and $\mathcal{T}_4$ are accessed 3 and 1 times, respectively. We assign each template $\mathcal{T}_i$ a weight $\omega(\mathcal{T}_i)$ based on this information. So we have $\omega(\mathcal{T}_1) = 10$, $\omega(\mathcal{T}_2) = 5$, $\omega(\mathcal{T}_3) = 3$ and $\omega(\mathcal{T}_4) = 1$.

First, we consider two heuristics for finding perfect XOR-schemes. We will heuristically attempt to p-color the conflict graph of our template set. To do this, we extend the weight function to the edges and vertices of the graph. We define the weight of an edge:

$$\omega(v_i, v_j) = \sum_{v_i, v_j \in T_k} \omega(\mathcal{T}_k)$$

The weight of a vertex is:

$$\omega(v_i) = \max_{v_i \in T_j} \omega(\mathcal{T}_j)$$

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 (assuming that all other edge constraints are met). The weighted graph for $\mathcal{T}$ is shown in Figure 6.

The heuristic coloring of graphs has been studied extensively. Graph coloring heuristics are used for solving problems such as scheduling [13], and register allocation [14]. One of the simplest heuristics is the *greedy coloring algorithm* [15]. Although there are no performance guarantees associated with this method, it has the advantage of simplicity, and it seems to work well in practice. The heuristics we present are modifications of the basic greedy method for weighted graphs.

A color number is an integer in the range $[0 \ldots p-1]$. A vertex number is an integer in the range $[0 \ldots n-1]$. A template number is an integer in the range $[0 \ldots t-1]$. We will use the following data structures:

- An array of color numbers `color[v]` indexed by vertex numbers.
- A boolean array `colored[v]` indexed by vertex numbers. Initially all elements are false.
- An adjacency list for each vertex `adj[v]`, implemented by a doubly linked list.
- An incidence matrix `edge`. If `v` is adjacent to `w` then `edge[v][w]` points to the entry of `v` in `w`'s adjacency list. Otherwise `edge[v][w]` is `nil`. This allows us to delete an edge from the graph in $O(1)$ time.

```
color_vertex (v)
{ m := infinity;
  for i := 0 to p-1 do
    if loss[v][i] < m then
      { c := i;
        m := loss[v][i]; }
color[v] := c;
colored[v] := true;
forall w in adj[v] do
      { delete_edge(w,v);
        loss[w][c] := loss[w][c] +
                           weight(w,v); } }
```

Figure 7: Subroutine used to color a vertex

- An array of weight values `loss[v][c]`, indexed by vertex numbers and colors. All values are initially zero.
- A max-heap `heap` of vertices, ordered by weight. Operations on `heap` are: `build_heap()` which initializes the heap to contain all vertices, `delete_max_heap()` which removes the maximal vertex from the heap, `insert_heap(v,w)` which inserts a vertex into the heap, and `empty_heap()` which returns true if the heap is empty, and false otherwise.

In both of the heuristics presented, we make use of the coloring subroutine `color_vertex` shown in Figure 7.

This routine colors a vertex with the color which has the lowest 'loss', and updates the loss values of the vertex's neighbors. It picks the best color locally. It deletes edges from the graph incident to the vertex being colored, as they will need no further consideration.

The first heuristic is called *Highest-Weighted-Conflict-First* (HWCF) and works as follows. Put all vertices in a priority queue ordered by weight. Pick the maximally weighted vertex. Color it. Repeat until all vertices are expended. This is illustrated in Figure 8.

We analyze the time complexity of HWCF. Let m be the number of edges in the conflict graph of the template set. The size of the input is $O(tp)$. We require $O(n^2t)$ time to build the conflict graph. The subroutine `color_vertex` will be called exactly n times. The body of the first loop will therefore be executed $O(np)$ times. The body of the second loop might be executed as many as n times on each call to `color_vertex`. However, overall it will be executed m times. So the amortized complexity of `color_vertex` is $O(np+m)$. Build-heap takes $O(n)$ time. The main loop of HWCF will be executed n times, and each `delete_max_heap` operation takes $O(\log n)$ time. HWCF runs in $O(p(t+n)+n^2t)$ time, where t, 2^p, and n, are the number of templates, the number of processors, and the number of distinct vectors of the template bases, respectively.

For the second heuristic, we require the data structures used by HWCF, and the following additional ones:

```
HWCF ()
{ build_heap();
  while not empty_heap() do
     { v := delete_max_heap();
       color_vertex(v);   } }
```

Figure 8: Heuristic HWCF

- A max-heap `heap2` of vertices, ordered by weight. Operations on `heap2` are: `build_heap2()` which initializes the heap to contain all vertices, `delete_max_heap2()` which removes the maximal vertex from the heap, `delete_heap2(v)` which deletes a vertex from the heap, and `empty_heap2()` which returns true if the heap is empty, and false otherwise.
- A boolean array `candidate[v]`, indexed by vertex numbers. All entries are initially false.
- An array `h2v[i]` of pointers to `heap2` entries indexed by vertex numbers. `h2v[i]` always points to the entry of vertex `i` in `heap2`. (Operations on `heap2` keep `h2v` up to date.) This allows us to perform `delete_heap2(v)` in $O(\log n)$ time.

The second heuristic, called *Most-Immediate-Conflict-First* (MICF), works as follows. We are best equipped to decide the color of a vertex when some of its neighbors have already been colored. The un-colored neighbors of vertices in the colored set are coloring candidates. Whenever a vertex is colored, add the vertices adjacent to it to a priority queue (which is empty before any vertices are colored). We repeatedly remove the maximal vertex from the priority queue and color it, until the current component is completely colored. Within each component of the graph (the graph may not be connected) we begin by coloring the vertex with maximal vertex weight.

We keep a priority queue `heap` of vertices which are coloring candidates. We also must have some method of locating the maximal vertex in each component. This is facilitated through the use of `heap2`, which contains all vertices which have not been colored.

Initially, all vertices are in `heap2`. The algorithm begins by selecting the vertex with maximal weight from `heap2` and coloring it. Its neighbors are then inserted into `heap`, and deleted from `heap2`. We then repeatedly delete the maximal vertex in `heap`, color it, remove its adjacent vertices from `heap2` and add them to `heap`. We keep track of which vertices have been inserted into `heap` using the `candidate` array. We use this information to insure that no vertex will be inserted into `heap` twice. This proceeds until there are no edges in `heap`. We have completely colored the current component. We proceed to the next component by deleting the maximal vertex remaining in `heap2`. The code for MICF is illustrated in Figure 9.

We analyze the time complexity of MICF. Again, the size of the input is $O(tp)$, and it takes $O(n^2t)$ time to construct the graph. The analysis of `color_vertex` is the same as for HWCF, $O(np+m)$ time is used. Each vertex is inserted into `heap` exactly once. We therefore have n calls to insert, and `delete_edge`. `delete_max_heap` is also called n times. `insert_heap` and

```
MICF ()
{ build_heap2();
  heap := empty_heap;
  while not empty_heap2() do
    { v := delete_max_heap2();
      done := false;
      while not done do
      { forall w in adj[v] do
        if not candidate[w] then
            { insert_heap(w);
              candidate[w] := true;   }
        color_vertex(v);
        delete_heap2(v);
        if empty_heap() then
          done := true;
        else
          v := delete_max_heap();} } } }
```

Figure 9: Heuristic MICF

`delete_max_heap` take at most $O(\log n)$ time, while `delete_edge` takes $O(1)$ time. The sum of the number of calls to `delete_max_heap2` and `delete_heap2` is n, and each call requires $O(\log n)$ time. The total is $O(n \log n)$. The total complexity is $O(p(t+n)+n^2t)$, the same as that of HWCF.

We now introduce a heuristic SP for augmenting a perfect XOR-scheme, which has conflicts, to a semi-perfect scheme, with fewer conflicts. We assume that a perfect XOR-scheme is given. We require the following data structures:

- An array `T[i]` indexed by template numbers. Each `T[i]` is a list of the basis vectors in T_i. `T[i][j]` is the index of `j`th vector in the `i`th template basis.
- A max-heap `heap3` of templates, ordered by weight. Operations on `heap3` are: `build_heap3()` which initializes the heap to contain all templates, `delete_max_heap3()` which removes the maximal template from the heap, and `empty_heap3()` which returns true if the heap is empty, and false otherwise.
- A boolean array `used_set[i]` indexed by colors.
- An array `temp[i]` indexed by vertex numbers. `temp[i]` is a list of the templates whose basis contains the basis vector corresponding to v_i.
- An array `card[i]` indexed by vertex numbers. `card[i]` is the number of template bases in which the basis vector corresponding to v_i appears. (`card[i]` is the size of `temp[i]`.)
- A boolean array `blocked[i]` indexed by vertex numbers. All entries are initially false.

```
augment (t,v)
{ for i := 0 to p-1
      used_set[i] := false;
for i := 0 to p-1
  {  c := color[T[t][i]];
      used_set[c] := true;   }
for i := 0 to p-1
  if not used_set[p] then
      { Phi[i][v] := 1;
        break;  }
forall w in blocked_by[v]
      blocked[w] := true;   }
```

Figure 10: Subroutine augment used by SP

- An array `Phi[i][j]` which is initialized to be the matrix of the perfect XOR-scheme.
- The `color[v]` array of the perfect XOR-scheme. Both HWCF and MICF produce this array as a side-effect.
- An array `blocked_by[v]`. `blocked_by[v]` is a list of all vertices `w` that will become blocked if `v` is augmented. (All basis vectors represented by `v` and `w` are both in some template.)

The heuristic works as follows. If a one is added to a column $\Phi_{*,i}$, then a one cannot be added to any column $\Phi_{*,j}$, where $\Phi_{*,i}$ and $\Phi_{*,j}$ are both in some $\Phi(\mathcal{T}_k)$. Or alternatively, v_i and v_j are both in T_k. Whenever a one is added to a column $\Phi_{*,i}$, mark all $\mathcal{T}_k$ such that v_i is in T_k as *blocked.*

Initially, no columns are blocked. We consider templates in order of their weight (maximal first). If $\mathcal{T}_i$ is conflict free, go to the next template. Otherwise, two columns of $\Phi(\mathcal{T}_i)$ are necessarily identical. Adding a non-zero to one of the columns will increase the rank of $\Phi(\mathcal{T}_i)$, provided that the row to which the non-zero is added contains only zeros, and that the column is not blocked. In the case that one of the columns is blocked, we augment the other one. If neither is blocked, we may choose to augment either. We choose the one which is contained in the least number of templates. Let the column so chosen be $\Phi_{*,j}$. We examine the rows of $\Phi(\mathcal{T}_i)$. If some row k contains only zeros, we set $\Phi_{k,j} = 1$, and add all the vectors (columns) which appear in some template with v_j to the blocked set. We then proceed to the next template.

SP uses the subroutine augment which is shown in Figure 10. The code for SP is illustrated in Figure 11.

SP begins by initializing `heap3` to contain all templates. SP then selects the maximal template `t` from `heap3`. If the template is blocked, we skip directly to the next template. Otherwise, every pair of vectors `T[t][i]` and `T[t][j]` in the template basis is checked in

```
SP ()
{build_heap3();
 while not empty_heap3() do
   {t := delete_max_heap3();
      done := false;
      for i := 0 to p-1 do
      { for j := i+1 to p do
         if color[T[t][i]] = color[T[t][j]] and
           (not blocked[T[t][i]] or not blocked[T[t][j]]) then
         { if blocked[T[t][i]] then
              augment(t,T[t][j]);
           else if blocked[T[t][j]] then
              augment(t,T[t][i]);
           else if card[T[t][i]] > card[T[t][j]] then
              augment(t,T[t][j]);
           else
              augment(t,T[t][i]);
           done := true;   }
        if done then
         break;  } } } }
```

Figure 11: Heuristic SP

the inner two loops. Each pair corresponds to an edge in the conflict graph of the template set. Each vector corresponds to a vertex. If some pair with identically colored vertices is found, augment the column corresponding to the vector which appears in fewer template bases. Once this is done, the inner two loops are broken out of by setting `done` to true. This process continues until `heap3` is empty. We maintain a set `blocked` of templates which are blocked.

The `augment` subroutine works as follows. We initialize `used_set` to be empty. The `i`th basis vector of the `t`th template has color `c`. This corresponds to the `c`th row of the matrix Φ containing a non-zero. When the second loop is completed, `used_set[c]` will be false exactly when the `c`th row of Φ is all zeros. We may augment any of these rows. In the third loop the first such row is found, and augmented. We then update `blocked` in the fourth loop.

We analyze the time complexity of SP. Let α be the number of calls to `augment`. The first three loops of `augment` all run in $O(p)$ time. The body of the third loop may be executed as many as n times in the worst case. The overall complexity of all calls to augment is $O(\alpha pn)$. The main loop of SP will be executed t times. Each call to `delete_max_heap3` takes at most $O(\log t)$ time. The body of the innermost loop will be executed at most $O(p^2 t \log t)$ times. The overall complexity of SP is $O(p^2 t \log t + \alpha pn)$. Note that it is not possible that $\alpha = t$. If this were the case then all template bases would be disjoint, and either HWCF or MICF would find a conflict-free coloring.

10 Performance Evaluation

We experimentally evaluate the performance of the proposed schemes and compare them to other known schemes.

Let $\omega(\mathcal{T}_i)$ be the number of accesses to $\mathcal{T}_i$. A lower bound on the number of accesses can be defined as:

$$A_{\min} = \sum \omega(\mathcal{T}_i)$$

Unfortunately, there is no guarantee that the lower bound is achievable. Therefore, the optimum access time (A_{opt}) of a perfect scheme, for a given set of templates each with a given frequency, is found by using a branch-and-bound algorithm.

We evaluate the performance of a perfect XOR-scheme by comparing the number of accesses required with the optimal perfect XOR-scheme. Given a storage scheme s which on the average requires A_s accesses, the average percent deviation of scheme s from A_{opt} is evaluated as $P_s = (A_s/A_{\text{opt}} - 1) \cdot 100$ which is the percentage of extra memory accesses beyond the optimal that s requires. The number of memory accesses for a perfect XOR-scheme is:

$$A_s = \sum_{\mathcal{T}_i \in \mathcal{T}} \omega(\mathcal{T}_i) 2^{(p-\textbf{rank}(\Phi(\mathcal{T}_i))}$$

We tested heuristics HWCF and MICF using a Monte Carlo simulation. Template sets were generated randomly. Given n, p, and t, we generated randomly t unique templates consisting of p unique vectors, selected randomly from a basis of n vectors. Each template was given a random weight between 1 and 10^5, inclusive. Heuristics HWCF and MICF were run on each template set. The results of this simulation are displayed in Table 1.

	$p=3$		$p=4$		$p=5$		$p=6$	
t	P_{HWCF}	P_{MICF}	P_{HWCF}	P_{MICF}	P_{HWCF}	P_{MICF}	P_{HWCF}	P_{MICF}
3	2.1	0.2	6.3	1.5	11.8	3.7	20.9	10.3
4	5.0	0.9	11.0	4.3	18.0	11.2		
5	8.3	1.8	15.8	9.2	24.2	18.0		
6	10.0	3.2	20.0	11.9	28.7	22.5		
7	12.3	4.9	21.9	15.5				
8	14.0	6.7	23.6	18.4				
9	15.4	8.6	24.0	20.4				
10	16.4	10.2	25.0	21.0				
11	16.7	11.4	24.4	22.0				
12	17.4	12.5	23.9	21.8				

Table 1: Monte Carlo simulation comparing HWCF and MICF

	$p=3$		$p=4$		$p=5$		$p=6$	
t	S_{HWCF}	S_{MICF}	S_{HWCF}	S_{MICF}	S_{HWCF}	S_{MICF}	S_{HWCF}	S_{MICF}
3	0.0	0.0	0.2	0.0	0.5	0.1	3.2	0.8
4	0.1	0.0	0.6	0.1	2.4	0.8		
5	0.2	0.1	1.2	0.5	4.7	2.5		
6	0.3	0.1	2.7	1.3	8.4	5.8		
7	0.6	0.1	4.1	2.2				
8	1.3	0.4	6.0	4.5				
9	1.6	0.6	7.6	5.7				
10	2.4	1.3	9.8	7.6				
11	2.8	1.6	10.8	9.4				
12	3.9	2.4	12.2	11.1				

Table 2: HWCF and MICF augmented by SP

Average P_{HWCF} and P_{MICF} are shown for $3 \leq t \leq 12$ and $3 \leq p \leq 6$. One thousand cases were run for each instance of p and t. The speed of the branch-and-bound algorithm prohibited us from completing the table. The number of distinct vectors of all the generated template bases was fixed at $n = 17$ (17 is prime). Note that in general HWCF was outperformed by MICF. Both heuristics degrade in a smooth fashion with increasing numbers of templates and increasing template size. However, for a dozen templates and 16 memory modules ($p = 4$), both heuristics deviate on the average by more than 20% from the optimum solution.

The semi-perfect heuristic SP has been applied to each of the perfect schemes that are found using HWCF, MICF, and branch-and-bound, respectively. The corresponding semi-perfect schemes are denoted by S_{HWCF}, S_{MICF}, and S_{BB}, respectively. Table 2 shows the average percent deviation of S_{HWCF} and S_{MICF} with respect to S_{BB}.

The semi-perfect heuristic strongly reduces the degree of conflict and decreases by nearly 50% the deviation that is obtained for perfect-schemes. This significant improvement is achieved at the cost of incorporating the least number of additional 1's in the storage matrix.

Table 3 shows the average percent increase in the number of 1's, i.e. additional hardware,

	$p = 3$		$p = 4$		$p = 5$		$p = 6$	
t	S_{HWCF}	S_{MICF}	S_{HWCF}	S_{MICF}	S_{HWCF}	S_{MICF}	S_{HWCF}	S_{MICF}
3	0.1	0.0	0.3	0.1	0.5	0.2	0.7	0.4
4	0.3	0.1	0.6	0.3	0.9	0.6		
5	0.6	0.2	1.0	0.7	1.3	1.1		
6	0.9	0.4	1.4	1.0	1.6	1.5		
7	1.2	0.6	1.7	1.4				
8	1.5	0.9	2.0	1.7				
9	1.8	1.3	2.2	2.1				
10	2.1	1.5	2.3	2.2				
11	2.4	1.9	2.5	2.4				
12	2.6	2.2	2.5	2.5				

Table 3: Average augmentation for HWCF and MICF

	0–4	5–9	10–14	15–19	20–24	25–29	30–34
HWCF & SP	76.2	5.2	3.3	1.9	0.9	0.5	0.2
MICF & SP	78.7	3.5	1.9	0.9	0.4	0.2	0.1

Table 4: Percentages of test cases within performance ranges

	$p = 3$		$p = 4$		$p = 5$		$p = 6$	
t	*INT*	*FPS*	*INT*	*FPS*	*INT*	*FPS*	*INT*	*FPS*
3	500	323	837	431	1259	479	1764	484
4	502	326	845	433	1276	476		
5	503	327	843	431	1273	478		
6	502	330	844	431	1276	481		
7	502	329	841	428				
8	503	330	839	428				
9	502	329	840	431				
10	501	328	840	432				
11	501	328	840	431				
12	501	330	843	434				

Table 5: Average percentage deviation of interleaving (INT) and fixed-pattern (FPS) schemes

needed to implement the semi-perfect storage matrices of HWCF and MICF, respectively. For all the studied cases, the cost of upgrading a perfect scheme to semi-perfect exceeds the cost of the perfect scheme by less than 5%.

Analysis of the distribution for the deviations of S_{HWCF} and S_{MICF} indicates that nearly 76% and 78% of the population is within the 4% deviation boundary, respectively. The detailed analysis is shown in Table 4. It gives the percentages of test cases (for the entire study) which fell within the given performance ranges.

Comparison of the proposed scheme is carried out with respect to traditional interleaving and to a fixed-pattern scheme similar to the one proposed in [9]. Here, row-major storage is used as representative of regular interleaving. A fixed-pattern scheme is selected that allows conflict-free access to rows, columns, and both diagonals. Table 5 shows the average access time deviation of accessing randomly generated data templates using row-major interleaving (INT) and the fixed-pattern storage (FPS), respectively. As for the case of HWCF and MICF, deviations of INT and FPS are measured with respect to reference S_{BB}. The row-major scheme is inadequate for arbitrary patterns as it requires nearly five times the reference access time (S_{BB}) in case of 10 templates and 16 processors. The fixed-pattern scheme is superior to the row-major scheme, but it is outperformed by semi-perfect schemes as shown on Table 2.

The proposed heuristics performed well for template bases of small size. However, for larger template sets or template sets with larger bases, a more sophisticated method with reasonable time complexity is not apparent.

11 Comparison to other approaches

By considering a given set of templates, Frailong, Jalby, and Lenfant [7], investigated the design of conflict-free storage schemes by showing how a storage matrix can be defined in terms of the template bases. They give the necessary and sufficient condition for conflict-free access of parallel memories for one template but no method is presented for finding the XOR-scheme in case of composite templates. Their method to design a XOR-scheme involved backtracking, and thus could potentially require exponential search time.

For vector processors and stride access, one approach [2, 10] is based on finding an XOR-matrix that optimizes the access for one stride. Other strides can be accessed with less conflict than that using traditional interleaving provided that buffers are added to each memory.

The proposed approach differ from the above methods because our scheme finds the XOR-scheme for arbitrary combined templates for which the global access time is minimized. We used a weighted conflict graph in which the nodes represent the basis vectors of the templates and the edges represent the amount of access time penalty that should be paid if the image of two nodes, connected by that edge, were not independent by the XOR-matrix. Evaluation of this approach has experimentally proved to be effective in reducing the amount of conflicts while using reasonable implementation cost. The contribution of our work are: 1) a non-redundant XOR-matrix for arbitrary combined templates, 2) use of conflict graph to represent the optimization problem, and 3) an efficient heuristic for minimizing the access time.

12 Conclusions

The performance of SIMD computers can be dramatically affected by the serialization of memory accesses. Perceiving data templates and their access frequency is within compiler capability. The use of this knowledge by a cost-effective heuristic storage scheme has been proposed in an attempt for minimizing access time.

We considered schemes with minimum hardware requirements (Perfect schemes), and proved that finding a storage matrix that minimizes the degree of conflict is an NP-complete problem. Thus, a heuristic approach has been proposed for finding approximate solutions under strict minimum hardware requirements. In this case the number of 1's in the storage matrix is equal to the number of distinct vectors in template bases.

Evaluation of perfect schemes was carried out using a Monte Carlo simulation and by comparing heuristic achievement to the optimum solution. The deviation from optimum of the access time for perfect schemes smoothly increases with the number of templates and their size. Unacceptable deviations have been observed for a dozen templates and 16 processors.

Given a perfect storage scheme, a heuristic has been proposed to further reduce the degree of conflict by incorporating minimum additional hardware, thus creating a semi-perfect scheme. Evaluation shows that semi-perfect schemes strongly reduce the degree of conflict of perfect schemes, with small additional hardware cost.

The proposed dynamic storage schemes are intended to be part of the processor segment translation table. They convert real addresses into module and offset addresses. By applying the proposed approach at the compiler level, significant speedup is expected compared to the traditional memory interleaving technique and other static schemes. Since the allocation scheme is invisible to the programmer, reduced algorithm complexity and reduced design time are immediate developments of this research. One possible future extension to this work is to incorporate network requirements within the proposed heuristics, for various types of networks.

13 Acknowledgments

Thanks to Daniel Hirschberg, George Leuker, and Michael Dillencourt for listening critically to the various proofs.

References

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

[2] 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.

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

[4] 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.

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

[6] 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.

[7] J. M. Frailong, W. Jalby, 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.

[8] D. T. Harper III and J. Jump. Vector access performance in parallel memories using a skewed storage scheme. *IEEE Transactions on Computers*, C-36(12):1440–1449, Dec 1987.

[9] K. Batcher. The multidimensional access memory in STARAN. *IEEE Transactions on Computers*, C-26:174–177, Feb 1977.

[10] 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.

[11] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[12] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 216–226, 1978.

[13] J. McHugh. *Algorithmic Graph Theory.* Prentice-Hall, 1990.

[14] G. J. Chaitan. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(2):201–207, 1982.

[15] B. Bollobàs. *Graph Theory: An Introductory Course.* Springer-Verlag, 1979.