Organization of Parallel Memories

by

Husam Saad Abu-Haimed

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

June, 1997

ORGANIZATION OF PARALLEL MEMORIES

BY

HUSAM SAAD ABU-HAIMED

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In

COMPUTER ENGINEERING

JUNE 1997

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by

HUSAM SAAD ABU-HAIMED

under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee

[illegible]

Dr. Mayez Al – Mouhamed (Chairman)

[illegible]

Dr. Khalid M. Al – Tawil (Co – Chairman)

[illegible]

Dr. Khalid Elleithy (Member)

[illegible]

Dr. Muslim Bozyigit (Member)

[illegible]

Dr. Khalid M. Al – Tawil
(Department Chairman)

[illegible]

Dr. Abdallah M. Al – Shehri
(Dean, College of Graduate Studies)

9-8-97
Date

كلية الدراسات العليا
COLLEGE OF GRADUATE STUDIES

Dedicated

To my mother

who gave me every thing in excess

and asked for nothing but my success.

To my father.

To my brothers and sisters.

Acknowledgement

First and foremost, all praise to Allah who gave me all the help, guidance, and courage to finish my work. May Allah help me to convey all what I learned for the benefit and goodness of Islam and humanity.

Acknowledgment is due to King Fahd University of Petroleum and Minerals. Dhahran, Saudi Arabia, for providing me with all the support needed to conduct this research.

I am so thankful to my thesis advisor, Dr. Mayez Al-Mouhamed, for his continuous help and guidance throughout the course of this thesis and all of my research. As my thesis advisor as well as my academic advisor, he put every possible effort to help me achieve my goals. Working with him is an experience that I can never forget. I am also thankful to my thesis committee members Dr. Khalid Al-Tawil, Dr. Khaled Elleithy, and Dr. Muslim Bozyigit for their real cooperation and helpful and constructive criticism.

All my thanks and appreciation to my mother, in the first place, and then to my family for their cooperation, understanding, and patience throughout my academic work. My Deep thanks go to my dear friend Mr. Tariq Ibraheem for his great help and support since the beginning of my work and up to the last moment. Thanks also to my friends Ala Al-Fuqaha, Ahmed Al-Shargawi. Talal Al-Kharroobi, and Mohamed Kaleem for their help. Last but not least, I am thankful to my friends (*in alphabetical order*) Abdallah Al-

Anzi, Abdallah Al-Ghumaiz, Ahmed Al-Kubayir, Ahmed Al-Sawi, Hatim Khan, Khalid Al-Blwi, Luai Aqil, Mouhamed Al-Khubani, Mouhamed Nabhan, Mouhamed Shogair, Nedal Saadeh, Samir Abbad, and Walid Al-Faris for their great friendship.

Contents

List of Tables

List of Figures

Abstract

Name: Husam Saad Abu-Haimed

Title: Organization of Parallel Memories

Major Field: Computer Engineering

Date of Degree: June 1997

The use of parallel memories has been the most promising technique to bridge the gap between high performance processors and available memories. By having N parallel memories, we aim to have a total memory bandwidth of NB, where B is the bandwidth of a single memory bank. However, using simple interleaving techniques, the effective bandwidth becomes much less than that and is about $\sqrt{N}B$ [21]. This is because of serialization of memory access which happens when a number of elements that will be referenced at the same time are stored into the same memory bank. Having low memory throughput can severely affect the overall performance of vector machines and SIMD systems.

To minimize memory conflicts, researchers have considered storage schemes for conflict-free access of frequently used patterns like rows, columns, and power of 2 patterns and strides. In this thesis, we consider power of 2 patterns as well as arbitrary strides. A new approach for combining different patterns into one linear bitwise storage scheme is proposed. We use 5 different approaches to construct combined storage schemes: 3 coloring-based heuristics, a Neural Networks approach, and a Genetic Algorithms approach.

In the case of power of 2 patterns, we were able to hit the lower bound on access time for small problems. For large problems, we achieved small deviations from the lower bound (5% -29%). In the case of arbitrary strides, access time of our schemes were 10% less than the best known bitwise schemes [41]. In addition, our schemes work with any power of 2 number of memories, while some other schemes work with a fixed number of memories like [41] which works with 8 memories.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

June 1997

خلاصة الرسالة

الاسم: حسام سعد ابو حيمد

عنوان الرسالة: تنظيم الذاكرة المتوازية

التخصص: هندسة الحاسب الآلي

تاريخ الشهادة: يونيه ١٩٩٧

يعد استخدام الذاكرات المتوازية من اكثر الطرق فعالية في وصل فجوة السرعة بين المعالجات عالية الأداء والذاكرات البطيئة الموجودة. باستخدام N وحدة ذاكرة متوازية يأمل المصمون بالحصول على سرعة كلية تساوي NB حيث أن B هي سرعة وحدة ذاكرة واحدة. لكن باستخدام طرق التخزين التقليدية تكون السرعة الفعلية لنظام الذاكرات المتوازية تساوي $\sqrt{NB}$ وليس NB [٢١]. هذا ناتج عن تسلسل عمليات الدخول للذاكرة والذي يحدث عندما يكون عدد من عناصر البيانات التي تستخدم في نفس الوقت موجودة في نفس وحدة الذاكرة.

ان انخفاض سرعة الذاكرة يؤثر بشكل كبير على الأداء العام في الأجهزة التجهيية (Vector Machines) وأجهزة (SIMD). لذا فقد أجريت عدة دراسات لحل هذه المشكلة منذ طرح الأجهزة التجهيية في منتصف الستينيات. فطور الباحثون الكثير من طرق التخزين لإزالة تعارضات الذاكرة (Memory conflicts) عند الدخول إلى النماذج (Patterns) كثيرة الاستخدام كالسطور، الأعمدة، الخطى (Strides) ونماذج القوة ٢ (Power of 2 patterns).

في هذا البحث سنقوم بدراسة طرق تخزين نماذج القوة ٢ بالإضافة إلى خطى الولوج العامة (Arbitrary strides). سنقدم طريقة جديدة لدمج نماذج مختلفة في خطة تخزين واحدة حيث نستخدم ٥ طرق مختلفة لبناء خطط التخزين (Storage schemes): ٣ طرق تقوم على تلوين الرسوم (Graph coloring)، طريقة تستخدم الشبكات العصبية، وطريقة تستخدم الخوارزميات الجينية.

في حالة نماذج القوة ٢، استطعنا إن نحقق الحد الأدنى لوقت الدخول لهذه النماذج في المسائل الصغيرة والمتوسطة. أما في المسائل الكبيرة فقد استطعنا تقليل الزيادة في وقت الدخول عن الحد الأدنى إلى ٥٪ – ٢٩٪ في حالة خطى الولوج العامة تمكنا من تحقيق تحسن بقدر ١٠٪ عن أفضل ما تمكن من تحقيقه الباحثون [٤١] بالإضافة إلى أن طريقتنا تعمل مع أي عدد (من القوة ٢) من الذاكراة على عكس ما في [٤١] والذي يعمل مع ٨ ذاكرات فقط.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران– المملكة العربية السعودية

يونيه ١٩٩٧

Chapter 1

1. Introduction

The processing power needed by scientific and engineering applications like numerical analysis, image processing, artificial intelligence, and many other applications has been ever increasing. To satisfy these needs, the processing power of high performance processors has been increasing in a similar pace. However, for many applications like image processing and numerical analysis, the overall performance is highly affected by how fast a processor can get the information to be processed. In other words, the speed of the memory system can significantly affect the overall performance.

The bandwidth mismatch between fast processors and the available memories has been a main design problem for long time. Researchers and designers have been always looking for the right way of combining processing power and memory system architecture. Different ways have been in use to bridge the gap between processors and memories. The available techniques fall mainly into three categories : technology, software, and architecture.

1.1 Technology

In the technology side, scientists have been pushing hard to go around undesired physical characteristics of memory devices. Physical characteristics like signal delay, races, and capacitive behavior have been the most significant obstacles of speeding up memory. Many of these characteristics have been negligible for long time, but started to cause problems for larger scales of integration. As the integration technology advances, many negligible effects become significant. Solving these problems is the aim of most technological research. In general, technology contributes to speeding memory up by the use of alternative technologies, wider buses, SRAM, and special interfaces.

1.1.1 Alternative Technologies

Today, most of the industry relies on the use of CMOS technology which is cheaper than other technologies. With CMOS, much higher densities of memory devices can be integrated into a chip, which gives larger storage. Moreover, it consumes less power than other technologies like TTL and ECL. However, undesired physical characteristics like signal delay and capacitive behavior are encountered more in CMOS technology, which makes it slower. To gain speed, other technologies like TTL and ECL (which have much faster switching) can be used, though that will be on the expense of cost and power consumption. ECL is much faster, but the most power consuming. BiCMOS is a hybrid technology of TTL and CMOS. It is faster than CMOS and less power consuming than TTL [32].

1.1.2 Wider Buses

Memory speed can also be gained by increasing the memory bus width. Doing so allows transferring more data bits at the same memory cycle. Memory bus width ranges from 8-bit to 32-bit in different systems. The optimal width is the one that matches the memory bus with the system bus [32].

1.1.3 Use of SRAM

The main problem of the main memory is being dynamic. In Dynamic RAM (DRAM), data bits are stored in capacitors. A data bit is read by draining out the current from the capacitor. So, the data bit has to be written back and the capacitor has to be recharged before reading it again. In addition, capacitors need to be refreshed because of the current leakage they have. This makes the memory cycle much longer. Another type of RAM is the Static RAM (SRAM). In SRAM, a data bit is stored in a flip-flop and a read cycle can start as soon as the previous one is complete. This makes SRAM much faster than

DRAM. Unfortunately, SRAM is about four times larger than DRAM and hence costs four times more [32]. So, we can not build the whole memory of SRAM, but it can be used as a cache memory. Cache is a small SRAM inserted between the processor and the memory. The cache holds data with high probability of being referenced and so avoids accessing the slow DRAM main memory as shown in Figure 1-1. Design issues of cache are discussed in section 1.3.1.

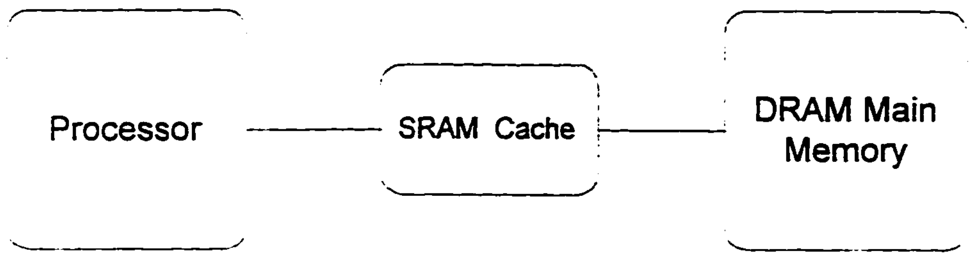


Figure 1-1 : Cache Memory

1.1.4 Special Interfaces

Another bottleneck is the interface between memory and processor. Several input/output modes were invented to enhance and speed up this interface [32]. When a memory row is selected by stropping its address, all data bits in that row appear on the output amplifiers. In random access mode, one column is selected and its bits appear on the output pins. In a page mode read operation, all the row data bits are held on the sense amplifiers while new column is selected. This way, no time is spent on writing the data on the amplifiers back to the memory cells and precharging before a new column in that row is selected. This reduces the memory cycle if the data to be read lies in the same row. Various similar modes are in use like hyperpage mode [32].

In spite of the memory speedup achieved by technological development, it has some problems. First, it is bounded by physical limits. Signal speed inside memory chips is bounded by the speed of light and we are now very close to this limit; current signal speed is about $\frac{1}{3}$ of the speed of light. Moreover, device dimensions are approaching the atomic level and further dimension reduction is becoming increasingly harder; compare the $.25\mu$ technology with the $2.64\,\text{Å}$ diameter of the Silicon atom [23]. So, speeding memory up by increasing signal speed or reducing device dimensions is almost exhausted. Second, whenever memory speed is increased by technological developments, processor speed can be similarly increased, which means that the speed gap between processors and memories is still persistent and will result in low utilization of processing power.

1.2 Software

Software and hardware are functionally equivalent, but with different cost and performance. Software can be thought of as an extension of hardware. We can not afford enough physical memory to hold all programs that may run concurrently, so we can emulate this memory in software by virtual memory and page swapping. Similarly, we can not afford a CPU for every program or user of a system, so we emulate this in software by multitasking or timesharing. In these two cases we compensated for hardware by software to save cost on the expense of performance. But how can we use similar arguments to hide memory hardware latency by software and speed up the overall memory system.

If the memory technology used is slow or the memory organization does not deliver the needed throughput, the software can not change these inherent physical and hardwired characteristics of the memory system. The software can not boost the memory speed to match the processor speed. However, it can improve overall performance by trying to hide these undesired characteristics by mainly two techniques. First, by reducing the number of services requested from the systems slowest part, the memory. Second, by keeping the processor busy during the portion of time spent on waiting for memory.

1.2.1 Minimizing Memory References

We can reduce the number of memory references by two means. First, by reducing the frequency of load and store operations. This is a compiler task. The compiler optimizes the code and efficiently schedules internal registers to minimize the number of load and store operations. Second, if it is necessary to use load or store operations, then try to make them to the cache instead of the main memory. This task is performed by the operating system. The memory management part of the operating system uses different algorithms to keep the most likely to be referenced pages in the cache. By doing so, the operating system can hide the memory latency from the processor. There are many cache management algorithms and techniques in the literature. The performance of these algorithms is application dependent [45,48].

So, no matter how the compiler will optimize, the program must need access to the memory. And no matter what clever techniques the operating system will use to manage the cache, there will be cases where the main memory will be accessed (cache misses). So, software can virtually reduce the memory latency, but can not completely hide it.

1.2.2 Reducing Idle Time

If the software can not completely hide the memory latency, it still has a chance of partially or completely hiding its effect on performance. This can be achieved by keeping the processor busy performing other tasks instead of waiting for the memory operation. Many techniques are in use to keep the processor busy while waiting for a memory request. This is an operating system and compilers task. The operating system can switch to another process after issuing a memory request to minimize idle time. The compiler also can find pieces of code that can run in parallel so that the operating system can switch from one to another. The trend of multithreading is an example of such techniques that can be used to reduce the processor idle time.

1.3 Architecture

As discussed in previous sections, technology and software approaches have been used to speed up the memory. However, these approaches still have their limitations. Beyond these two approaches we are left with the other alternative approach, which is the architectural organization of the memory system. Research in memory architecture has been going in two directions : cache architectures and parallel architectures.

1.3.1 Cache Architecture

As mentioned in section 1.1.3, cache is a small SRAM inserted between processor and main memory. The cache memory function is based on the concept of data locality. The processor accesses a relatively small amount of data in each time window. The cache

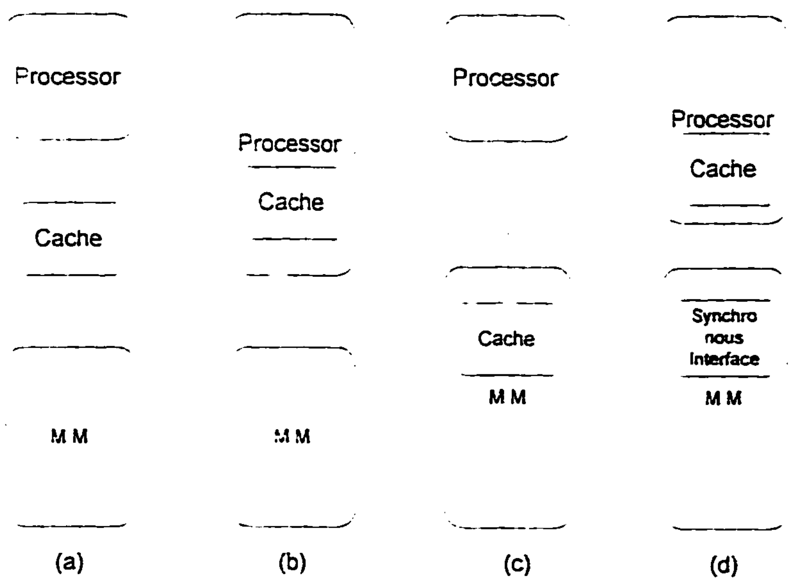


Figure 1-2 : Cache Topologies

memory is intended to capture the essential data by the program so that, main memory access time is hidden by the small access time of the cache. There are various cache topologies having different properties. Figure 1-2 [32] shows four different topologies. In the simplest form, cache is inserted between processor and main memory as in (a). In (b), the cache is embedded to the processor to avoid input/output delays and to match the memory bus with the wide processor bus. In (c), it is embedded to the main memory to connect the fast cache to the wide internal DRAM bus. In (d), the cache is embedded in the processor and a synchronous interface is embedded in the main memory to deliver fast bursts of data to the cache [32].

It is observed that the cache memory concept has some limitations. It seams that there is some limitation on increasing throughput beyond some cache size and organization. The limit results from two conflicting effects : temporal locality and spatial locality. The first requires small page size while the second requires the opposite. Since any program contains a mixture of the two (temporal locality and spatial locality), it is possible to find some cache size and organization that is suitable for a typical program but not necessarily for other programs [32].

1.3.2 Parallel Memory Architecture

The other architectural direction is the parallel memory organization. This approach can be divided into two categories: time parallelism and space parallelism.

1.4 Time Parallelism

In this approach, the memory access process is decomposed into a number of independent stages. When a memory access request is done with one stage, it can go to the next one and the previous stage can start processing another request. In other words, different memory requests can be overlapped. This is memory pipelining. A possible memory access decomposition is shown in Figure 1-3. In this example, it is decomposed into three stages. Unlike instruction execution pipelines, where up to 20 stages or more can be exploited, it is hard to decompose memory access process into more than about three stages. This puts a limit on the gain of memory pipelining or time parallelism [21].

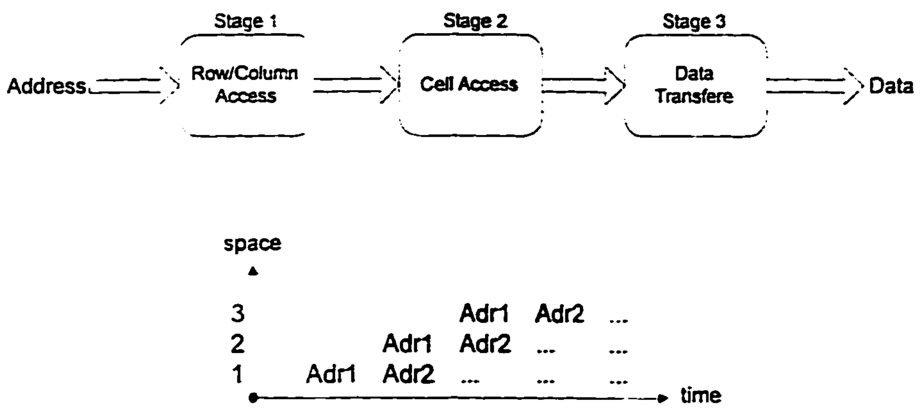


Figure 1-3 : Memory Pipelining

1.5 Space Parallelism

In time parallelism, different requests are processed by different stages representing different processing steps. In space parallelism, however, identical units process in parallel different requests. In other words, the memory is divided into identical memory banks so that, a number of memory requests can be processed at the same time. In this scheme, in general, a number of memory banks or memory modules (M) is connected through a network to a processor or in the general case to a number of processors (P). So, any processor p_i can access memory bank m_j by alignment through the network. One well-known example of this model is the SIMD computer shown in Figure 1-4. By using this scheme, we can achieve in the optimal case a memory speedup of M, which means

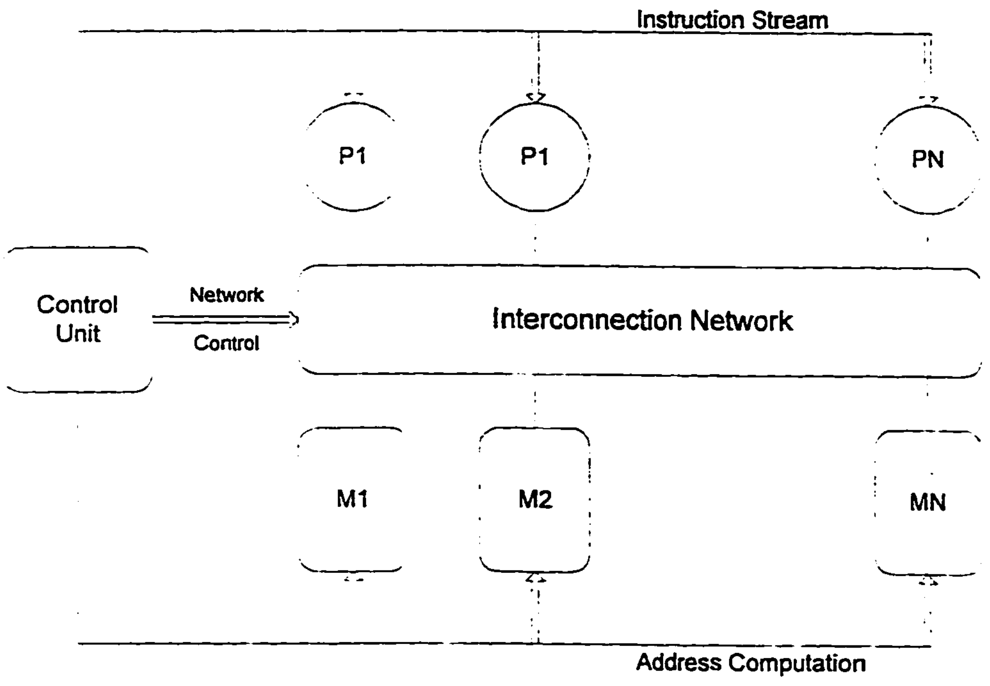


Figure 1-4 : SIMD Model

that the processor/memory speed gap will be significantly reduced or completely eliminated. By using M parallel memories, it is expected that the average bandwidth is MB_m, where B_m is the bandwidth of one memory bank. Unfortunately, due to serialization of addresses at the input of some memory banks, the effective bandwidth is far from being linear (MB_m). The reason is that due to data organization a request can not be decomposed into requests to all M memory banks, because many data elements fall into the same memory bank. An effective bandwidth given by Hwang [21], is $B_{parallel} = \sqrt{M} B_m$.

To achieve an optimal memory access, all the data elements that will be accessed at the same time should be stored into different memory banks. In many cases, this condition is difficult or impossible to be met. That may result in severe performance degradation of many orders. Consider the case were we have k data elements that will be needed at the same time, but are stored into the same memory bank. At least k memory cycles will be needed to access these k elements instead of $\left\lceil \frac{k}{M} \right\rceil$ cycles in the optimal case. To avoid or minimize this problem, we have to consider the data patterns that will be accessed by the program and predefine storage mappings that minimize access conflicts for these specific patterns. The task of finding out these data patterns is performed by a parallelizing compiler. Parallelizing compilers have to perform data dependence analysis in order to parallelize a given code.

As an example of simple data patterns, consider the case where the pattern is a set of consecutive data elements in an array. Low Order Interleaving (LOI) can be used in this case. In LOI, consecutive array elements are stored into consecutive memory banks.

For example, if element e_i is stored into memory bank m_j, then element e_{i+1} is stored into memory bank $m_{(j+1) \bmod M}$. If we have 2^m memory banks and d-bit array element address, then LOI can be implemented by taking the least significant n bits of the d-bit ($n \le d$) address as the memory bank number in which that array element should be stored. For example, the element e_i whose address in the array is $i_{d-1}i_{d-2}...i_{n-1}...i_0$ will be stored into memory bank m_j where $j = i_{n-1}i_{n-2}...i_0$. LOI helps only with patterns in which the least significant n bits of addresses take all the possible combinations for all the elements in the pattern. A set of consecutive elements is a special case of such patterns. However, many other patterns are frequently needed by different types of applications. Consider the situation where a program references array elements with stride 2 (stride is the difference between two consecutive elements in the pattern) and we have 2^m memory banks. Now if LOI is used, only half of the memory banks will be used, those even numbered, and the memory throughput will drop to half the optimal. In general, if the stride is 2^s with 2^m memory banks and LOI, 2^s cycles will be needed to access 2^m elements instead of 1 cycle in the optimal case.

1.5.1 Prime Memory Systems

A more general case is for any stride S and M memory banks (S,M are any integers). Let g be the greatest common divisor of S and M, $g = GCD(S, M)$. Then, it can be shown using number theory that g cycles will be needed to access a pattern of M elements if

LOI is used [7,25]. Many techniques were proposed to solve this problem [7,14,25,38]. One proposed solution is to have S and M relatively prime [7], i.e., $GCD(S, M) = 1$. This leads to the optimal case and M elements can be read in one cycle. If M is a prime number, then for any stride S that is not a multiple of M then, $GCD(S, M) = 1$ and we can achieve the perfect access.

This scheme will guarantee conflict-free access for any number of patterns P with strides $s_1, s_2, \ldots, s_P$ such that, $GCD(s_i, M) = 1;\ 1 \leq i \leq P$. However, this scheme is computationally expensive and time consuming because it needs to compute the *modulo* function. In the case where S and M are power of two, the *modulo* function is just a shift operation. In the general case where S and M can be any numbers, however, it will cost much more. Implementing the modulo function in hardware is difficult while implementing it in software is time consuming.

1.5.2 Skewing Schemes

Other approaches were introduced to avoid computing the *modulo* function [4,13,18,19,20,31]. Many of these approaches rely on observing the data patterns frequently needed by different applications and then constructing priori mappings of data elements so that these patterns could be accessed with minimum conflicts, though other patterns might not. Examples of frequently used patterns are columns, rows, diagonals, and square blocks. Figure 1-5 shows three simple storage schemes of such patterns. The numbers in bold are the row and column indices. The numbers inside indicate the memory bank number in which the corresponding array element should be stored. In (a), elements in one column are stored into different memory banks which means that they can be accessed in parallel. However, elements in one row are stored into one memory bank. The scheme in (b) gives conflict-free access to columns as well as to rows as can be seen. Scheme (c) adds another pattern and gives conflict-free access to square blocks of size 2×2. By having these patterns known in advance, storage schemes or mappings that give conflict-free access to these patterns are predefined. Such schemes are good for applications were predefined patterns constitute most of the actually accessed patterns. Other applications with more general or irregular patterns will suffer severe degradations.

$$
\begin{array}{ccccc}
 & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\
\mathbf{0} & 0 & 0 & 0 & 0 \\
\mathbf{1} & 1 & 1 & 1 & 1 \\
\mathbf{2} & 2 & 2 & 2 & 2 \\
\mathbf{3} & 3 & 3 & 3 & 3
\end{array}
\qquad
\begin{array}{ccccc}
 & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\
\mathbf{0} & 0 & 1 & 2 & 3 \\
\mathbf{1} & 1 & 0 & 3 & 2 \\
\mathbf{2} & 2 & 3 & 0 & 1 \\
\mathbf{3} & 3 & 2 & 1 & 0
\end{array}
\qquad
\begin{array}{ccccc}
 & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\
\mathbf{0} & 0 & 2 & 1 & 3 \\
\mathbf{1} & 1 & 3 & 0 & 2 \\
\mathbf{2} & 2 & 0 & 3 & 1 \\
\mathbf{3} & 3 & 1 & 2 & 0
\end{array}
$$

(a) (b) (c)

Figure 1-5 : Simple Storage Schemes

1.5.3 Bitwise Address Transformation Schemes

Looking for more general and efficient schemes, several Bitwise Address Transformations (BAT) were proposed. In this section, linear BAT schemes will be discussed, while general BAT schemes will be discussed in section 2.5. Linear transformations map the addresses generated by the program into memory banks. The *d*-bit address generated by the program is multiplied by a linear transformation matrix $M_{d \times n}$ to generate an *n*-bit address that is the memory bank number in which that array element is to be stored. In the matrix multiplication process, the logical AND corresponds to scalar multiplication while XOR corresponds to addition. This transformation is simple and efficient; all what it needs is ANDing followed by XORing. Its simplicity makes it suitable for hardware implementation which means fast operation. Many known schemes can be represented by linear transformations. The schemes shown in Figure 1-5 can be represented in linear transformation forms as shown in Figure 1-6.

$$(a)\begin{pmatrix} m_0 \\ m_1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} j_1 \\ j_0 \\ i_1 \\ i_0 \end{pmatrix}$$

$$(b)\begin{pmatrix} m_0 \\ m_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}\begin{pmatrix} j_1 \\ j_0 \\ i_1 \\ i_0 \end{pmatrix}$$

$$(c)\begin{pmatrix} m_0 \\ m_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}\begin{pmatrix} j_1 \\ j_0 \\ i_1 \\ i_0 \end{pmatrix}$$

Figure 1-6 : Bitwise Transformations

These are linear transformations of the form $m_{2\times1} = M_{2\times4} \times X_{4\times1}$, where m is the memory bank number, M is the transformation matrix, X is the array element address, and j and i represent the column index and row index, respectively. Linear transformation schemes can be used with many other types of regular or irregular data patterns. They were used in situations with single power-of-two stride and perfect access was guaranteed. However, with no satisfying solutions arise in situations with multiple patterns and with nonpower-of-two strides.

Chapter 2

2. Literature Review

2.1 Background

Rather than just increasing the speed of scalar computations, vector parallelism has been heavily studied and implemented since 1965. This is due to the ease of detecting and exploiting this type of parallelism [25,38]. Vector processing is utilized in vector machines and in SIMD machines. In such machines, a set of memory words belonging to some vector are read and processed in parallel. Sohi [41] noticed that cache memories were not able to provide vector machines with high-bandwidth access to elements of large data structures, which necessitates the use of parallel or interleaved memories. Seznec and Lenfant [37] noticed that parallel memories in SIMD machines have to be synchronized. They showed that an SIMD machine without synchronization will have about 38% of the theoretical throughput, while this throughput can be increased to 50% just by synchronizing the memories without changing the storage scheme.

The unability to access vector elements in parallel can significantly decrease the memory throughput and affect the overall performance. Such cases happen when more than one element belonging to the same vector are stored into the same memory bank, which causes memory access conflict. This problem has been the subject of many studies since the late 1960's. The degree of conflict depends on the access stride. If the stride is relatively prime with the number of memory banks, perfect access can be easily achieved. Problems, however, arise with other strides. In [37], Seznec and Lenfant considered a stride distribution collected from real applications. The distribution was as follows :

- 80% of the vectors are accessed with stride 1.
- 10% of the vectors are accessed with odd strides other than 1.
- for $k \geq 1, \frac{10}{2^k}\%$ of the accessed vectors have strides of the form $r2^k$, where r is an odd number.

The study showed that having $N = 2^n$ memories, then accessing 100 N-element vectors using simple interleaving techniques will result in 55% of the throughput for $n = 9$. Though only 10% of the strides are not conflict-free accessible, this small portion may cause severe degradation.

As shown above, using simple interleaving techniques, one can waste about 50% of the memory throughput. Since the introduction of Illiac IV in the late 1960's, researchers have developed many techniques to avoid this waste by avoiding memory conflicts. These techniques belong to three classes : Prime Memory Systems (PMS), Skewing schemes, and Bitwise Address Transformations.

2.2 Regular Data Patterns

Before discussing the different storage schemes, let's look at the kind of data patterns usually considered in these schemes. In a system of M memories each of K words, Budnik [7] showed that the ratio between all possible M-element vectors and those that are accessible in parallel is approximately $\left(\frac{1}{2\pi M}\right)^{\frac{1}{2}} e^{M}$. So, there is storage scheme(s) that allow conflict-free access to all possible patterns. So, it is beneficial to consider only those patterns that are expected in practice rather than considering arbitrary patterns. That is why all studies in the literature considered frequently used patterns like rows, columns, diagonals, and power-of-two patterns and strides of the form $r2^{x}$ for odd r. Budnik also showed that the ratio between differently ordered vectors and uniquely ordered vectors is M^{M-1}. So, it is of great benefit to have a reskewing interconnection scheme between processors and memories. By having such scheme, we can consider a vector instance without considering its order. A crossbar switch gives optimal connection and can reskew a vector in any order, but it is very expensive for large systems. Multistage interconnection networks are the best and most widely used alternative. Constructing a storage scheme for these networks is more difficult because it has to avoid network conflict in data realignment or reskewing.

2.3 Prime Memory Systems (PMS)

Budnik [7] and Lawrie [25] showed that if the number of memory banks and the access strides are relatively prime, then these strides can be accessed without conflict. Budnik proposed row rotation scheme, in which, every row of the array is rotated with respect to the previous row with a distance d. They showed that when d is relatively prime with the number of memory banks, then columns, rows, and square blocks can be accessed in parallel. They applied their scheme for the case of $2^l + 1$ memories and $d = 2^{2l} + 1$ and for other cases also.

In a conventional PMS, the mapping is defined as follows :

$$F(A) = A \bmod M$$
$$r(A) = A / M$$

Where F is the memory bank to which A is mapped and r is the offset of A in that memory bank. Computing this address is expensive; it needs computing the *modulo* function in addition to integer division to find $r(A)$. These two operations are complex and can cause a lot of overhead. Gao [14] proposed a high speed multiple-bit division approach which includes the approach to find " $A \bmod M$ " by parallel " l -bit cycle adder" in one step when $M = 2^l \pm 1$. Many other techniques were proposed to compute the *modulo* function. Lawrie [25] proposed the use of prime number of memories and $P = 2^p$ processors. He redefined the PMS mappings to be :

$$F(A) = A \bmod M$$
$$r(A) = A / P$$

By dividing over P instead of M, scheme avoids dividing by a prime number and instead uses a very simple division, since P is a power of two. This fast address computation is on the expense of memory wastage. Since $P < M$, only $\frac{P}{M}$ of the memory space is used, while $1-\frac{P}{M}$ of the memory is wasted in what is called *holes*. A possible combination could be $M = 257$ and $P = 256$. In this case, $\frac{1}{257}$ of the memory is wasted. Figure 2-1 shows this mapping for $M = 5$ and $P = 4$.

	M0	M1	M2	M3	M4
0	0	1	2	3	hole
1	5	6	7	hole	4
2	10	11	hole	8	9
3	15	hole	12	13	14
4	hole	16	17	18	19
5	20	21	22	23	hole
6	25	26	27	hole	24

Figure 2-1 : Lawrie's Scheme with M=5 and P=4

Gao [14] and Seznec and Lnfant [38] proposed a new mapping :

$$F(A) = A \bmod M$$
$$r(A) = A \bmod C$$

Where $C = 2^c$ is the number of words per memory bank. By having $M = 2^c \pm 1$, we make M and C relatively prime. They showed using the Chinese remainder theorem that this mapping is a one-to-one mapping from the address space $\{0,\ldots, M \times 2^c - 1\}$ onto the set of memory words of the memory system. In this scheme integer division is avoided and in computing $r(A)$ only *modulo* with respect to $C = 2^c$ is computed which is a very simple operation; it is just the least significant c bits of A.

A major problem with PMS schemes is realigning data to processors. This alignment is very complex. In [38], they proposed a special network called the Chinese remainder network. However, this network is complex and specific to this application. In addition to this, computing the addresses in PMS schemes is very complex. Although fast hardware was proposed to compute the *modulo* function, it still causes some overhead and requires very complex hardware. Moreover, it is desired to have power-of-two number (that is not prime) of memories and processors for the following reasons [13] :

1. It simplifies address computation.

2. It allows the use of most efficient interconnection networks.

3. It achieves optimal utilization of address decoders.

2.4 Skewing Schemes

A skewing scheme is defined in the literature to be a scheme that does not use a prime number of memories. Most skewing schemes are linear. Linear schemes are of the form $F(i,j) = (ai + bj) \bmod M$ for some constants a and b, where i and j are the indices of the corresponding element. In computing $F(i,j)$, arithmetic and modulus operations are needed. When bitwise logical operations are used instead of arithmetic and modulus operations (for example, $F(i,j) = i \oplus j$), the scheme is called bitwise transformation. All skewing schemes are periodic. The class of linear schemes is a proper subclass of the class of periodic schemes [11].

Patterns like rows and columns can be accessed without conflict using linear skewing schemes. However, Lawrie [26] and Budnik [7] showed that patterns like diagonals and coils are nonlinear and there is no linear skewing scheme to access such patterns with 2^n memories, where $n > 1$. Shapiro [39] claimed that *T={row, column, diagonal, square, block}* has no valid skewing scheme (a scheme that gives conflict-free access), but Deb [10] presented a counter example to that claim by presenting a valid nonlinear scheme of T for $n = 2$. Then, Raghavendra [4] gave a general algorithm that finds a nonlinear skewing scheme for any n.

Wijshoff and Leeuwen [47] studied periodic skewing schemes and showed various properties of these schemes in light of foundations in the mathematical theory of integral latices and $\check{Z}$ -modulus.

Harper [18] proposed the use of dynamic storage schemes, in which a storage scheme is selected for each vector based on the access patterns used with that vector. He synthesized storage schemes for $r2^x$ stride families, where r is an odd number. It was shown that if a storage scheme gives conflict-free access to a stride 2^x, then it will also give access to any stride $r2^x$ for any odd r, but not for any other x.These schemes rely on skewing every period of addresses generated by a linear scheme with respect to the previous period. This scheme is shown in Figure 2-2. The performance of this scheme was studied for arbitrary strides and simulation showed that using buffers can improve throughput for nonconflict-free strides.

	M0	M1	M2	M3
0	0	1	2	3
1	7	4	5	6
2	10	11	8	9
3	13	14	15	12
4	16	17	18	19
5	23	20	21	22
6	26	27	24	25

Figure 2-2 : Row Skewing

To solve the problem with nonlinear patterns like diagonals and coils, Deb [11] proposed a multiskewing scheme. In multiskewing, an array is divided into parts and each part is stored using a different linear storage scheme. Deb showed that storing an array with different linear schemes enables conflict-free access of nonlinear patterns. He then gave an instance of such schemes and showed that all conventional linear patterns in addition to diagonals and coils can be accessed without conflict.

2.5 Bitwise Address Transformations

Bitwise address transformation schemes are skewing schemes that use bitwise logical instead of arithmetic operations in address computations. In [19,34,41,46] the advantages of using bitwise address transformations over the other schemes were discussed:

1. Use of simple logical operations rather than arithmetic and modulus operations.
2. Simple control of cheap, general, and well studied interconnection networks.
3. Compact representation as logical operations and Boolean matrices.
4. Bitwise operations have no carry which makes the complexity of address computation independent of the number of memories.
5. Data movement is fast and efficient and can be pipelined through the interconnection network.

Frailong, Jalby, and Lenfant [13] developed a general power-of-two data pattern that covers many conventional patterns like rows, columns, grids, and rectangles. They proposed an XOR-scheme of the form: $F(i,j) = Ai \oplus Bj$, where A and B are Boolean matrices. They identified the necessary and sufficient conditions for conflict-free access of these patterns and identified the conditions for their scheme to pass the Ω network. Norton and Melton [31] synthesized the first bitwise linear transformation scheme for conflict-free access of power-of-two strides and formulated the conditions for their scheme to pass the baseline network. They also gave an efficient procedure for constructing Boolean matrices representing these storage schemes.

Lee and Wang [27] tried to apply bitwise schemes for the pattern set T. To achieve so, they used a nonlinear bitwise scheme defined as follows :

$$\begin{aligned} F(i,j) &= i \oplus Pj \oplus j_0 \\ r(i,j) &= i \end{aligned}$$

Here P is a Boolean matrix. They showed that this scheme gives conflict-free access of T. However, data realignment is very complex. Processors are connected to memories with an indirect binary n-cube network in addition to a reverse shuffle network connecting processors with each other. Raghavendra and Boppana [34] defined a similar mapping :

$$\begin{aligned} F(i,j) &= i \oplus Pj \\ r(i,j) &= i \end{aligned}$$

This is a valid scheme for T. They used an Ω network with some additional hardware for data realignment. This solution uses half the number of memories and network size of those used in Lawrie's scheme. Boppana and Raghavendra modified this scheme to work with the inverse Ω network [6,34].

Most of the storage schemes try to provide conflict-free access of known patterns like T or strides of the form $r2^x$. But the use of these schemes with other patterns and strides significantly degrades the throughput [18,41,46]. [18,41,46] studied this problem and proposed the use of buffers at memory inputs and outputs (Figure 2-3) and presented simulation results that showed the profitability of using these buffers. Valero [46] uses buffers with a reordering technique. In reordering, elements of a given nonconflict-free stride are read into buffers in consecutive memory cycles (other vectors can be read during these cycles) and then reordered and given to processors. Valero showed that using buffers without reordering can lead to a maximum latency of $\frac{L(T-1)}{T}$ cycles, where L is the length of the vector stream to be read and $T = 2^t$ is the ratio between processor and memory cycle. This needs $\frac{L}{T} - \left\lceil \frac{L}{T^2} \right\rceil$ buffers per memory bank. With reordering, however. at most $T-1$ extra cycles will be needed to read a stream of length L.

Harper [19] expressed the conflict-free access conditions of a $r2^x$ stride in terms of the periodicity of the transformation matrix and used elementary matrices of known periods to compose transformation matrices of maximum period for optimal access.

In all the previous schemes, a space multiplexed memory system was assumed. Seznec and Lenfant [37] proposed the use of time multiplexed as well as space multiplexed memory system. This system is shown in Figure 2-4. The system consists of P processors and N logical memory banks where $P = N = 2^n$. Each logical memory bank consists of 2^d physical banks. They used XOR scheme on different fields of the array element address to compute the physical memory bank. Their scheme gives conflict-free access to any slice of 2^{d+q+n} consecutive vector elements with stride $r2^k$, $k \leq q$. So, this scheme restricts the size of accessible vectors. The evaluation of this scheme, using the stride distribution presented at the beginning of the chapter, showed that it boosts the throughput to 90% rather than 50% with LOI.

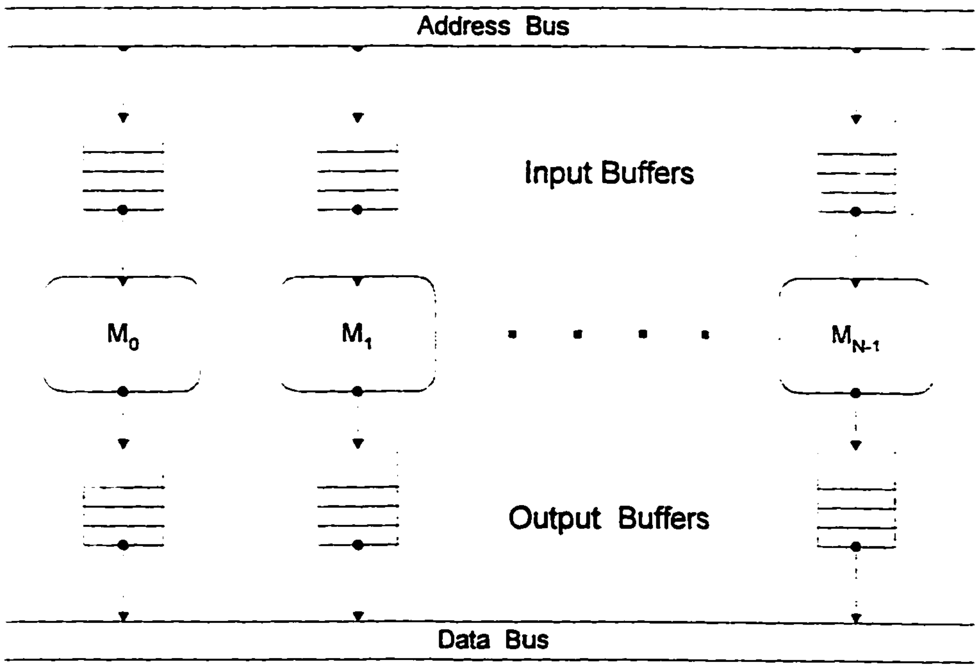


Figure 2-3 : Memories with Buffers

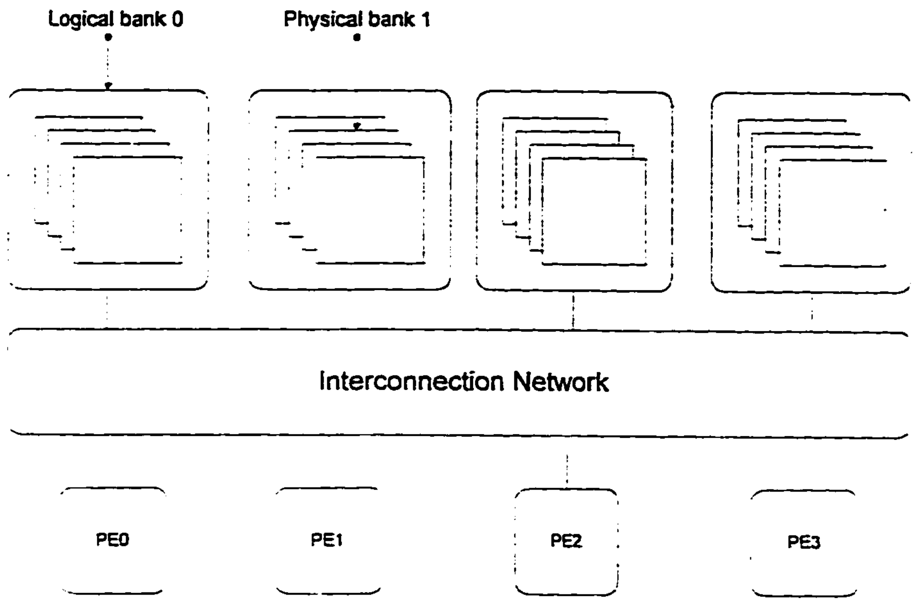


Figure 2-4 : Space-Time-Multiplexed Memory

Chapter 3

3. Linear Bitwise Storage Schemes

3.1 Introduction

Since the knowledge of data patterns or patterns is within the capability of the compiler, storage schemes that offer higher flexibility have been proposed [13] as linear transformations from the processor address to the storage location. Frailong [13] presented a necessary and sufficient condition for conflict-free access of data patterns. The image by the storage scheme of all the elements of a given pattern should map into different memory modules. Therefore, the columns or the rows of the needed transformation matrix should be linearly independent.

Norton [31] synthesized a transformation matrix that allows conflict-free access to a number of power of 2 strides. The scheme was intended for the RP3 multiprocessor. As the interconnection network should provide data alignment between processors and memories, other constraints [5] can then be used for finding the storage matrix. By using bitwise linear transformation matrices, Boppana [6] proposed a conflict-free storage scheme to the row, column, main-diagonal, and square blocks. The above matrix is characterized by non-singular diagonal sub-matrices.

The data patterns 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 patterns. Dynamic storage schemes make the hardware transparent to the user and avoid reorganizing the data, but require the address transformation to 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 patterns.

For array references, vectorizing loops in presence of loop-carried-dependency (LCD) may constrain the SIMD load and store operations to some specific data patterns such as the row (LCD across the columns), the column (LCD across the row), stride access, or arbitrary blocks. The problem is to find how arrays can be stored into parallel memories in order to force the elements of a given data pattern to be uniformly distributed over the memories. Each uniformly distributed data pattern can then be accessed in one memory cycle so that each PE is mapped to one element of the pattern.

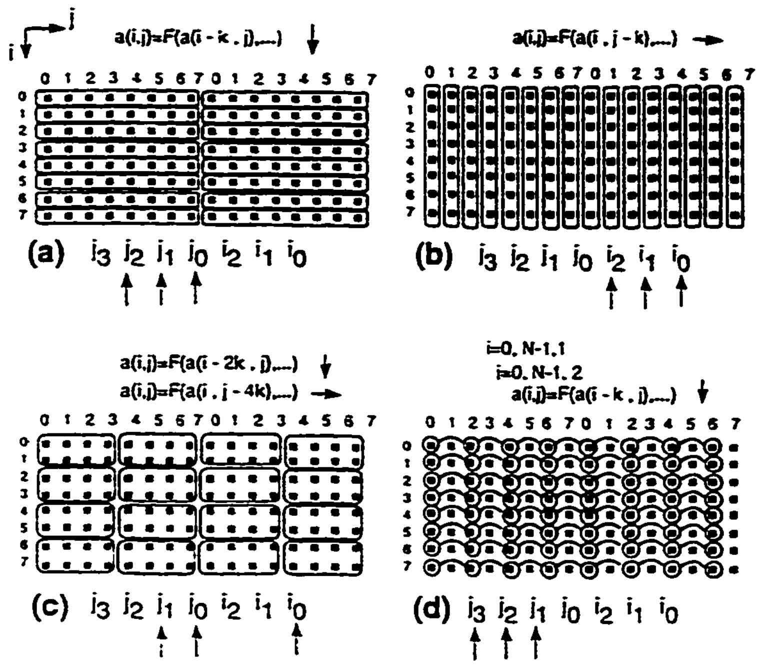
Given a program that requires access to a set of data patterns, our objective is to find a cost-effective storage scheme that minimizes overall memory access time. A general approach for combining the constraints of different data patterns into one single bit-wise address transformation is proposed. We will see that finding the address transformation that minimizes overall access time of a given set of data patterns is reduceable to m-coloring, where 2^m is the number of memories. Since the transformation should be implemented by each PE of some SIMD system, it is interesting to optimize its hardware requirements. Optimizing the transformation is investigated by using coloring heuristics and the access frequency of each data pattern. In this case, the problem consists of minimizing the access time as well as the number of gates required to implement the transformation.

In section 3.2, we give a background on power of 2 patterns and their representation. In section 3.3, linear bitwise schemes are analyzed with some definitions and results known in literature. In section 3.4, we present our combined storage scheme. Access of arbitrary strides is discussed in section 3.5. The chapter is concluded by stating the NP-Completeness of the problem of constructing combined storage schemes in section 3.6.

3.2 Background

Consider an SIMD computer that consists of a number of processing elements and memory units connected by a network as shown in Figure 1-4. Any processor can access any memory unit through an interconnection network. We assume that there is an equal number N of processors and memory units, and that N is a power of two $N = 2^m$. 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. Let the statement $a(i,j) = F(a(i,j-1),\ldots)$ be the body of a loop, where $a(.,.)$ is a 2-D array and F is some function. Due to data dependency between $a(i,j)$ and $a(i,j-1)$ across the iterations, one way to vectorize the loop is to distribute the code so that the PEs evaluate elements along rows of the array. Thus the PEs need to perform simultaneous access to a column of data element from the array which allows defining a column data access pattern. Note that a column access may be translated within the array to allow different instances of column patterns be accessed in parallel.



a(i,j)=F(a(i - k , j),...)
a(i,j)=F(a(i , j - k),...)
a(i,j)=F(a(i - 2k , j),...)
a(i,j)=F(a(i , j - 4k),...)
i=0, N-1,1
j=0, N-1, 2
a(i,j)=F(a(i - k , j),...)
(a)
(b)
(c)
(d)

Figure 3-1 : Patterns a) sub-row, b) sub-column, c) 2X4 blocks, and d) row/stride access

A *pattern* is defined as a collection of array elements whose addresses are related by some relationship. The *origin* of a pattern is the coordinate of its upper leftmost element. Changing the origin allows access to different instances of the pattern. Figure 3-1 shows an 8×16 array that is partitioned into a set of 8-elements row-pattern, column-pattern, 2×4 block-pattern, and a stride-2 row-pattern. A pattern that is needed for some SIMD program is a collection of array elements that any instance (its origin) of it can be accessed in parallel, by all the PEs, during the running of the program. We are interested in patterns of arbitrary dimension but having power-of-2 elements.

We would like to find a function that allows us to access all patterns in a *conflict free* manner. By this, we mean that for all given patterns, for all pattern instances, all of the elements of any pattern 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 pattern, and each row (column) is a pattern 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 conflict-free *storage scheme* [13,31] for the row and the column.

Without loss of generality, the memory is considered as a single two dimensional array such that the element in the ith row and the jth column is denoted by (i,j). The upper left-most element is (0,0). To simplify the notation, the sizes of the horizontal and vertical dimensions are both 2^d. By convention, the array is always indexed by a variable i in the vertical direction, and a variable j in the horizontal direction.

The binary representation of an integer i is $i_{d-1}...i_1i_0$. In other words, the least significant bit of i is i_0, the first bit is i_1, etc.

A row position i can also be thought of as a vector, over the finite field Z_2, the integers modulo 2. In Z_2, addition corresponds to logical **exclusive or**, and multiplication corresponds to logical **and.** $(i_{d-1}...i_1i_0)$ is the vector representation of i, in terms of the bits of i. We define a vector space $F = Z_2^d$ for horizontal position. Let $B(F) = \{f_{d-1}, \ldots, f_0\}$ be the canonical basis of F, i.e. $f_0 = (0,\ldots,0,1), f_1 = (0,\ldots,1,0)\ldots, f_{d-1} = (1,0,\ldots,0)$. Each row has a unique representation as a vector in F. A row i is expressed as $i_{d-1}f_{d-1} \oplus \ldots \oplus i_1 f_1 \oplus i_0 f_0$ in terms of components over $B(F)$. For example, 11 in binary is 1101, 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 $1f_3 \oplus 0f_2 \oplus 1f_1 \oplus 1f_0$:

$$1.\overset{f_3}{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}} \oplus 0.\overset{f_2}{\begin{pmatrix}0\\1\\0\\0\end{pmatrix}} \oplus 1.\overset{f_1}{\begin{pmatrix}0\\0\\1\\0\end{pmatrix}} \oplus 1.\overset{f_0}{\begin{pmatrix}0\\0\\0\\1\end{pmatrix}}$$

Without loss of generality, we assume $d \times d$ 2-dimensional arrays to simplify the notation, but both sizes can be arbitrary as well. We similarly define vector spaces G for column positions and with canonical bases $B(G) = \{g_{d-1}, \ldots, g_1, g_0\}$. There are 2^m memories and that the number of PEs is identical to the number of memories. Let H be a vector space for memory unit numbers and its basis $B(H) = \{h_{m-1}, \ldots, h_1, h_0\}$.

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

Consider the parallel access to 8 memories for the set of elements $\{(i,j)\} = (i_{d-1},\ldots,i_0,j_{d-1},\ldots,j_3,-,-,-)$, where $i_{d-1},\ldots,i_0,j_{d-1},\ldots,j_3$ are constant for all the 8 elements and components i_2,i_1,i_0 take all possible combinations. The accessed pattern (T_1) consists of a sub-row of 8 successive elements of array A. Pattern T_1 is associated a basis $B(T_1) = \{g_2,g_1,g_0\}$. We note that i_2,i_1,i_0 are the components of (i,j) over the basis $B(T_1)$, i.e. projection of (i,j) over $B(T_1)$. The set of components $i_{d-1},\ldots,i_0,j_{d-1},\ldots,j_3$ represent the pattern origin and used to define one instance of T_1 By changing the origin we can access different instances of T_1. For example, we can position the origin of T_1 into 2^{2d-3} different locations within array A.

To access a sub-column of 8 successive elements, we define pattern T_2 that accesses the set $\{(i,j)\} = (i_{d-1},\ldots,i_3,-,-,-,j_{d-1},\ldots,j_0)$ The basis of T_2 is $B(T_2) = \{f_2,f_1,f_0\}$. Finally, we define a pattern T_3 that consists of 2 successive rows by 4 successive columns for which the set of elements is $\{(i,j)\} = (i_{d-1},\ldots,i_1,-,j_{d-1},\ldots,j_2,-,-)$. The basis of T_3 is $B(T_3) = \{f_0,g_1,g_0\}$.

To cause the parallel access to all the 8 elements of T_1, for a given instance of the origin, one can choose to store each element $A(i,j)$ into memory (i_2,i_1,i_0). In this case, all the elements of T_1 will be distributed over all the memories and parallel access to T_1

becomes possible. Therefore, accessing one single pattern does not pose any problem. We are interested in finding storage schemes that allow an array to be accessed in parallel with respect to an arbitrary set of data patterns.

3.3 Analysis of Storage Schemes

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 m vectors out of the $2d$ canonical vectors of $B(F \times G)$, where $n \leq 2d$. From now on we consider vector x (also y) as the projection of some (i,j) over V.

Consider the following two Boolean matrices:

$$M_1 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Matrix M_1 represents a linear permutation defined over the set of integers (0,....7). M_1 achieves the permutation (0,1,2,3,4,5,6,7)$\rightarrow$(0,7,2,5,4,3,6,1). Each source has a unique image by M_1. However, the image by M_2 of the ordered set of elements (0,1,2,3,4,5,6,7) is (0,7,3,4,4,3,7,0). We note that each image by M_2 has two sources. We are interested in finding Boolean matrices M that causes an array to be distributed over the memories for a given set of data patterns. Each element $A(i,j)$ of some array A will be stored into memory module Mx, where x is the projection of $A(i,j)$ over the basis of the corresponding pattern, and M will be called storage scheme. In the following we present a theorem that states the necessary and sufficient condition to cause the elements of an array to be distributed over the memories to allow parallel access to a given data pattern.

Theorem 1 *Two different vectors $x, y \in V$ always have different images by an $m \times m$ matrix M if and only if M is non-singular.*

The proof of this theorem can be found in any book on linear algebra. As an example of theorem 1, matrix M_1 allows one-to-one mapping of the source elements (0,...,7) to the image (0,7,2,5,4,3,6,1) which causes the source elements to be distributed over the memories if each array element $A(i,j)$ is stored into memory M_1x and M_1 is non-singular, where x is the projection of (i,j) over some three canonical vectors of $B(F \times G)$. Note that M_2 does not allow the same result because M_2 is singular. For example, in row-major memory organization element $(i,j) = (i_{d-1},...,i_0,j_{d-1},...,j_0)$ is stored into memory module Mx, where $x = (j_2, j_1, j_0)$ and M is the 3×3 identity matrix.

Theorem 2 *A pattern T that is defined by its basis $B(T) = \{t_{m-1},...,t_0\} \subset B(F \times G)$ can be accessed in parallel using 2^m memories if each element $x = x_{m-1}t_{n-1} \oplus ... \oplus x_0t_0$ of T is stored into memory module Mx and matrix M is non-singular.*

Proof As M is non-singular, then elements $x, y \in T$ such that $x \neq y$ satisfy $Mx \neq My$ as shown in Theorem 1. Therefore, the elements of T will be distributed over the memories whenever each element $x \in T$ is stored into memory Mx and M is non-singular.

3.4 Combined Storage Schemes

Consider a set $\Gamma = \{T_1, ..., T_q\}$ of data patterns so that each pattern consists of 2^m elements. The basis of each pattern T_k is denoted by $B(T_k) = \{t_{k,m-1}, ..., t_{k,0}\}$, where $t_{k,m-1}, ..., t_{k,0}$ are some canonical vectors chosen from $B(F \times G)$. Since each pattern instance has 2^m 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^m elements of any pattern instance should be distributed over the 2^m memories in order to allow parallel access to that pattern.

Definition 1 *The basis B of a set $T_1, ..., T_q$ of data patterns is the set of all distinct canonical vectors of the bases of all patterns $T_1, ..., T_q$: $B = \bigcup_{1 \le k \le q} B(T_k) = \{t_{m-1}, ..., t_0\}$, where t_k is a canonical vector of B and $n = Card(B)$*

The bases of two data patterns may or may not share a number of canonical vectors. Therefore, the number of distinct vectors n in the union of all pattern bases should always satisfies $m \leq n \leq 2d$. Consider the previously defined set of patterns $\{T_1, T_2, T_3\}$ with bases $B(T_1) = \{f_2, f_1, f_0\}$, $B(T_2) = \{g_2, g_1, g_0\}$, and $B(T_3) = \{f_0, g_1, g_0\}$. which allows finding $B = \bigcup_{1 \leq k \leq 3} B(T_k) = \{f_2, f_1, f_0, g_2, g_1, g_0\}$.

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 T_k in parallel. The constant components specify the origin of the pattern instance. For parallel access of sub-columns of 2^3 elements for some array $A(10 \times 12)$, the element address will be $(i,j) = (i_9, \ldots, i_0, j_{11}, \ldots, j_0)$ and its projection over the pattern (sub-row) basis is $\{i_2, i_1, i_0\}$. When accessing a sub-column in parallel, the components over (g_2, g_1, g_0) of all addresses located in a sub-column instance take all possible combination of bits in $\{i_2, i_1, i_0\}$. Moreover, the projection of all elements of some pattern T_k that is accessed in parallel take all possible combinations over the canonical vectors of $B(T_k)$.

We similarly define the projection of vector x over the basis B of all the data patterns.

Definition 3 *The projection of vector* $(i,j) \in F \times G$ *over the basis* B *is the vector* $x^b = x_{n-1}t_{n-1} \oplus \ldots \oplus x_0 t_0$ *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 $(i,j) = f_{d-1}i_{d-1} \oplus \ldots \oplus f_0 i_0 \oplus g_{d-1}j_{d-1} \oplus \ldots \oplus g_0 j_0$ has a projection $x^b = f_3 i_3 \oplus f_2 i_2 \oplus f_0 i_0 \oplus g_3 j_3 \oplus g_2 j_2 \oplus g_0 j_0$ over the basis $B = \bigcup_{1 \le k \le 3} B(T_k)$. In the following we define the combined storage matrix M that is used to evaluate the memory module number where the (i,j)th element of some array should be stored.

Definition 4 *The combined storage associated to all data patterns* $\{T_k : 0 \le k \le q\}$ *is a matrix* M *of dimension* $m \times n$ *such that each vector* (i,j) *is stored into memory location* Mx^b, *where* 2^m *is the number of memory modules and* n *is the number of distinct vectors in the union of all pattern bases.*

The combined storage matrix M can be seen as a collection of columns vectors, i.e. $M = [C_{n-1}, \ldots, C_0]$, where C_u is an $m \times 1$ 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 $0 \le u \le m$. 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, and T_3 is a 3×6 matrix because $B = \{f_2, f_1, f_0, g_2, g_1, g_0\}$. If one choose arbitrary data for the matrix M, we obtain:

$$Mx^b = \begin{array}{c} \begin{array}{cccccc} f_2 & f_1 & f_0 & g_2 & g_1 & g_0 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array} . \begin{pmatrix} i_2 \\ i_1 \\ i_0 \\ j_2 \\ j_1 \\ j_0 \end{pmatrix}$$

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}, and M_{T_3} are the following:

$$M_{T_1} = \begin{array}{c} \begin{array}{ccc} f_2 & f_1 & f_0 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \qquad M_{T_2} = \begin{array}{c} \begin{array}{ccc} g_2 & g_1 & g_0 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{array} \qquad M_{T_3} = \begin{array}{c} \begin{array}{ccc} f_0 & g_1 & g_0 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

In the following we summarize our analysis. For some array A, we consider a set $T = \{T_1, \ldots, T_q\}$ of data patterns that will be accessed by some program at run-time. Each pattern has 2^m elements and there are 2^m parallel memories. We are interested in finding a storage scheme so that each of these pattern can be accessed in one memory cycle. This requires that the elements of each pattern be distributed over the memories so that parallel access can occur to all the memories. Each power of 2 pattern T_k can be uniquely associated a basis $B(T_k)$. While each pattern basis has m canonical vectors, the set B of all distinct canonical vectors of all bases has $n \geq m$ canonical vectors because some of these vectors cannot be shared among patterns. To define the storage scheme, the address (i,j) of array element $A(i,j)$ is restricted to its components over B and this restricted vector is denoted by x^b. A storage matrix M $(m \times n)$ is used to find the memory module Mx^b where element $A(i,j)$ of array A should be stored.

Theorem 3 *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^m elements of some pattern $T_k = \{(i,j)\}$ that should 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 $x^{b(T_k)} = x_{m-1}t_{k,m-1} \oplus \ldots \oplus x_0 t_{k,0}$. When T_k is accessed in parallel, the components $x_{m-1} \ldots x_0$ take all possible combinations (2^m) 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^m combinations when considering all accessed elements of T_k.
- The projection of (i,j) over the remaining canonical vectors of B, which we denote by $R = B - B(T_k)$. The Projections over R are constant when T_k is accessed in parallel because $R \bigcup B(T_k) = \theta$.

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:

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

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.

Consider the parallel access to some instance of pattern T_3 for which the origin is defined by $(i_0, j_0) = (i_{d-1}, \ldots, i_1, 0, j_{d-1}, \ldots, j_2, 0, 0)$. Note that the basis of T_3 is $B(T_3) = \{i_0, j_1, j_0\}$. Let x^b be the projection of (i,j) over $B = \bigcup_{1 \le k \le 3} B(T_k)$ so that the accessed elements of T_3 are defined by $\{x^b\} = \{(i_2, i_1, i_0, j_2, j_1, j_0)\}$ = $(1, 0, -, 1, -, -)$, where i_0, j_1, j_0 take all possible combinations of bits when accessing pattern T_3. The origin of T_3 is defined by the bits $i_{d-1},\ldots,i_2,i_1,j_{d-1},\ldots,j_2$ that can be arbitrarily chosen. For example, assume the origin is chosen so that $i_2 = 1$, $i_1 = 0$, and $j_2 = 1$. In this case, the element $A(i,j)$ is stored into memory:

$$Mx^b = \begin{array}{cccccc} f_2 & f_1 & f_0 & g_2 & g_1 & g_0 \\ \left(\begin{matrix} 1 \\ 0 \\ 0 \end{matrix}\right. & \begin{matrix} 0 \\ 1 \\ 0 \end{matrix} & \begin{matrix} 1 \\ 0 \\ 1 \end{matrix} & \begin{matrix} 1 \\ 0 \\ 1 \end{matrix} & \begin{matrix} 1 \\ 1 \\ 0 \end{matrix} & \left.\begin{matrix} 1 \\ 0 \\ 0 \end{matrix}\right) \end{array} \cdot \begin{pmatrix} 1 \\ 0 \\ i_0 \\ 1 \\ j_1 \\ j_0 \end{pmatrix}$$

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

$$Mx^b = \begin{array}{ccc} f_2 & f_1 & g_2 \\ \left(\begin{matrix} 1 \\ 0 \\ 0 \end{matrix}\right. & \begin{matrix} 0 \\ 1 \\ 0 \end{matrix} & \left.\begin{matrix} 1 \\ 0 \\ 1 \end{matrix}\right) \end{array} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \oplus \begin{array}{ccc} f_0 & g_1 & g_0 \\ \left(\begin{matrix} 1 \\ 0 \\ 1 \end{matrix}\right. & \begin{matrix} 1 \\ 1 \\ 0 \end{matrix} & \left.\begin{matrix} 1 \\ 0 \\ 0 \end{matrix}\right) \end{array} \cdot \begin{pmatrix} i_0 \\ j_1 \\ j_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \oplus M_{T_3} \cdot \begin{pmatrix} i_0 \\ j_1 \\ j_0 \end{pmatrix}$$

The image by M_{T_3} of elements $(i_0, j_1, j_0) \in \{0,1,2,3,4,5,6,7\}$ is (0,4,6,2,5,1,3,7) and the image by M is (1,5,7,3,4,0,2,6). As one can see, the distribution of the array element over different memory modules 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 maintains one-to-one mapping between the elements of the accessed pattern and the memories.

3.5 Accessing Arbitrary Strides

In previous sections, we have seen the conditions and formulations for accessing power-of-2 patterns. In this section, we will show how we can use the same techniques to improve access time of arbitrary strides (nonpower-of-2).

Given an origin a (offset within the accessed array) and a stride s, the problem of stride access is to access the following sequence of addresses in an array A:

$$A(a), A(a+s), A(a+2s), \ldots, A(a+(N-1)s).$$

This is a vector of length N within the accessed array. In our case we assume that the vectors accessed are of length 2^m where we have 2^m memories. If we can access a vector of length 2^m in C cycles, then a vector of length N can be accessed in $\left\lceil \frac{N}{2^m} \right\rceil \cdot C$ cycles. So, restricting the vector length to 2^m dose not restrict the solution.

Any stride s can be written as $r2^x$, where r is an odd number. A power of 2 stride is a power of 2 pattern and can be characterized by its basis vectors. A power of 2 stride 2^x has the basis $\{f_x, f_{x+1}, \ldots, f_{x+m-1}\}$ for a vector of length 2^m Figure 3-2 shows this for $s = 2^1$ and $m = 3$. Now for any stride of the form $r2^x$, in the sequence of addresses generated, the bits $i_x, i_{x+1}, \ldots, i_{x+m-1}$, will take all the 2^m possible combinations as in the case of the power of 2 stride (2^x). However, with power of 2 strides, only those m bits change, while with arbitrary strides other bits may also change. This is shown in Figure 3-3 for stride $s = 6 = 3 \cdot 2^1$. Now, if the problem is to access an array with only one stride of the form $r2^x$, we can store it as we store the corresponding stride 2^x and use the bits $(i_x, i_{x+1}, \ldots, i_{x+m-1})$ of the address to indicate the memory bank number and we can guarantee perfect access. In other words, we will consider it as a

Changing bits

i_4	i_3	i_2	i_1	i_0
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0

Figure 3-2 : Address Sequence for s=2 and m=3

Other bits change

Basis bits

i_6	i_5	i_4	i_3	i_2	i_1	i_0
0	0	0	0	0	0	0
0	0	0	0	1	1	0
0	0	0	1	1	0	0
0	0	1	0	0	1	0
0	0	1	1	0	0	0
0	0	1	1	1	1	0
0	1	0	0	1	0	0
0	1	0	1	0	1	0

Figure 3-3 :Address Sequence for s=6 and m=3

power of 2 pattern with basis $\{f_x, f_{x+1}, ..., f_{x+m-1}\}$. If, however, more than one stride are to be accessed by a storage matrix, we can not guarantee perfect access even if the corresponding restricted matrix is nonsingular. This is because other bits in the address may change and cause two different addresses to be mapped to the same memory bank. There are some cases, however, where we can access more than one arbitrary stride without conflicts. One case could be if all strides belong to the same family $r2^x$ (they all have the same x, but may differ in r), because they will have the same basis. For example, the strides 1,3,5,7,... (they belong to the family $r2^0$) can be accessed without conflict in a combined storage scheme. Other cases with strides from different families can also be accessed without conflict (when the varying nonbasis bits of one stride do not fall in the corresponding restricted matrix of any other accessed stride) like 1,8,10. We will see also that in most of the other cases, a stride can be accessed with less than 3 cycles. So, to access any arbitrary stride $s = r2^x$, we will first convert it into a power of 2 pattern with basis $\{f_x, f_{x+1}, ..., f_{x+m-1}\}$ and use the same techniques used for power of 2 patterns to construct the storage matrix.

3.6 NP-Completeness

Suppose we are given a vector space Z_2^m, a set of variables $B = \{t_{m-1}, ..., t_0\}$, and a set $\Gamma = \{T_1, T_2, ..., T_q\}$ 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 satisfaction* (LIS).

Theorem 4 *Linear independence satisfaction is NP-complete.*

This theorem is proved in [1]. By stating this theorem, we conclude this chapter keeping in mind the NP-Completeness of the problem of constructing combined storage schemes which suggests resorting to heuristics. In the next chapter, we use three heuristics based on graph coloring.

Chapter 4

4. Heuristic Approaches

4.1 Introduction

In the previous chapter, we have seen that the problem of constructing combined storage schemes is NP-Complete. In this chapter we will describe three heuristics which we use to construct combined storage schemes. These heuristics are based on graph coloring. We will seen how a given set of patterns can be represented as a conflict graph. In [1], it was shown that constructing a *conflict-free* combined storage scheme for P patterns and 2^m memories is possible if and only if the corresponding conflict graph is m-colorable. So, our approach is to construct a combined storage scheme by coloring the corresponding conflict graph.

We assume that the compiler is capable of finding the patterns that will be accessed by a given program. 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 known. The access frequency of $w(T_k)$ pattern T_k is the number of times this 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:

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

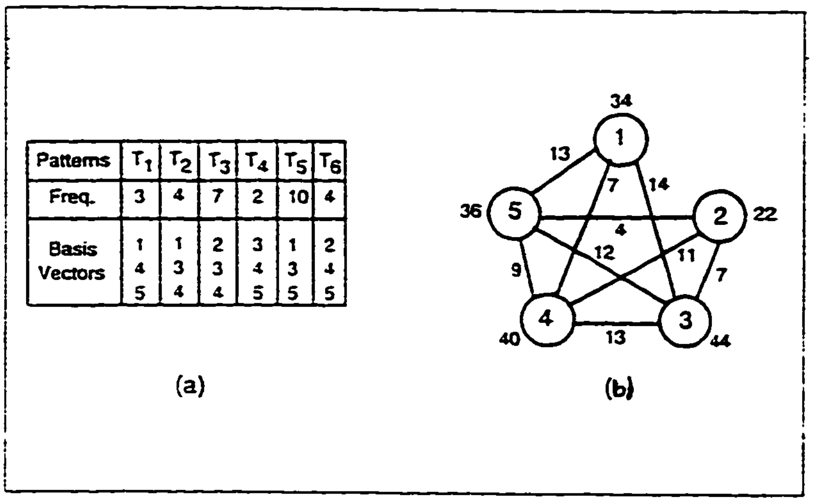
Where t and t' are two basis vectors. Each vector is assigned a weight which is the sum of edge weights linking this vector to all others:

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

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 $\{T_1,\dots,T_6\}$ that are defined using their basis vectors $B(T)$ which are denoted here by $1,\dots,5$ and their access frequency $w(T)$ as shown in Figure 4-1 (a).

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



Patterns	T_1	T_2	T_3	T_4	T_5	T_6
Freq.	3	4	7	2	10	4
Basis Vectors	1 4 5	1 3 4	2 3 4	3 4 5	1 3 5	2 4 5

Figure 4-1 : Set of Pattern Bases and their Conflict Graph

4.2 Coloring heuristics

In the following we use Weighted Graph Coloring for allocating values (colors) to the basis vectors. Initially, the basis vectors are associated an undirected graph in which a node u represents a basis vector and assigned the weight $w(u)$. An edge (u,v) indicates that the basis vectors u and v are member of at least one pattern basis. Edge (u,v) is given the weight $w(u,v)$ 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 total number of nodes in the graph is identical to the number of basis vectors in the union of all the pattern bases (m). The number of available colors is identical to the number of vectors in the pattern basis (n).

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 a number of coloring heuristics that are adequate for synthesizing storage schemes by using our approach.

4.3 Weighted coloring with node splitting

A first coloring heuristic, denoted by (GC) for Greedy Coloring, operates on weighted conflict graphs and perform node splitting when it fails in coloring a node.

GC 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. If all the available colors have been assigned to the neighbors of the current node v, then v is split into two nodes v' and v''. The splitting operation must divide the pattern bases that contain v as basis vector into two groups which nearly have equal weights. Whenever a node is split, GC re-evaluates the weights for all uncolored nodes and restarts again.

A node u that is present in only one pattern basis has necessarily $m-1$ 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 m neighbors that are assigned all the m 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 basis by $v^{''}$. Node splitting has the effect of reducing the degree of conflicts with other vectors at the cost of duplicating the representation of vectors in the storage matrix.

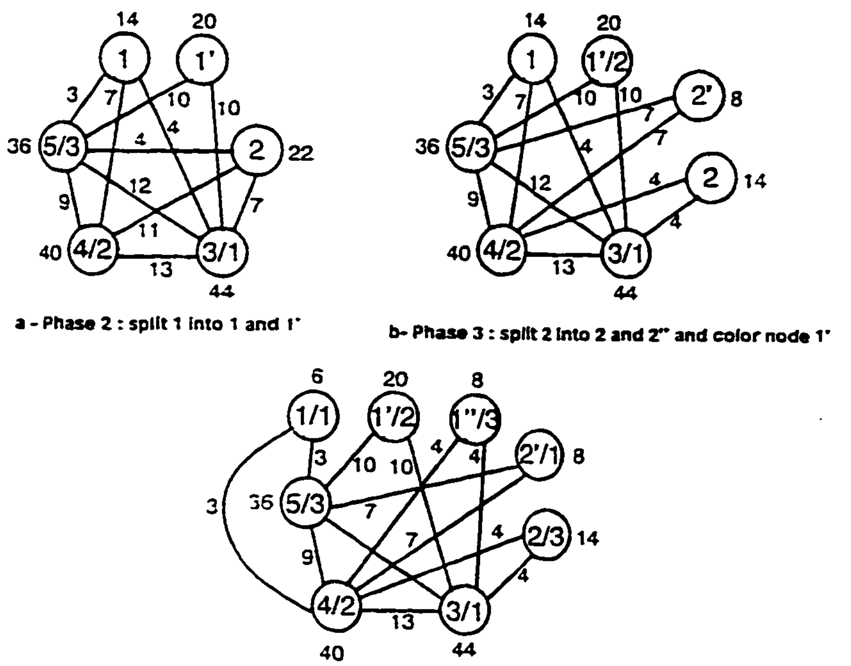

a - Phase 2 : split 1 into 1 and 1'
b- Phase 3 : split 2 into 2 and 2'' and color node 1'
c- Phase 4 : split 1 into 1 and 1'' and color (2, 1'', 2', and 1)

Figure 4-2 : Node Splitting by GC

Figure 4-2 shows the coloring by GC of the set of patterns displayed in Figure 4-1. In Phase 1, the heuristic color nodes 3, 4, and 5 in the order of decreasing weight. In phase 2, it splits node 1 into 1 and 1^{*} because the neighbors of 1 have all available colors as shown in Figure 4-2 (a). In phase 3, it split node 2 into 2 and 2^{*} for the same reason and color 1^{*} as shown in Figure 4-2 (b). Finally, in phase 4 the heuristic splits 1 again into 1 and 1^{**} which makes all the remaining nodes colorable as shown in Figure 4-2 (c).

In the worst case, we can have a fully connected graph, i.e., every node has $n-1$ edges. Splitting a node with e edges, may result in two nodes: one with 1 edge and the other with $e-1$ edges. An upper bound on the number of splits of a node with $n-1$ edges is $n-m$ where we have m colors. So, an upper bound on the number of splits needed to color any graph is $n(n-m)$. A split operation needs to update the graph. This update process takes $O(n)$ time. So, the time complexity of this algorithm is $O(n^3 - n^2 m)$.

4.4 A clustering-based heuristic

This simple heuristic considers each pattern basis and uniformly distributes its vectors over a number of clusters, where each cluster is associated to a color. The distribution is uniform because when handling one pattern basis, no pair of vectors are assigned to the same cluster. To minimize the conflicts, a vector is mapped to the cluster whose conflict with the vector is the least among all clusters. In the following we present this heuristic in more detail.

Initially, for each color j such that $0 \le j \le n-1$ 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 B_T be the current pattern basis. $B_T = \{v_0, \ldots, v_{n-1}\}$ consists of n distinct vectors. 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 $m \times m$ array. A cluster can be either empty or it contains a sub-set of basis vectors. Each cluster receives one basis vector after examining each pattern basis except when this basis vector is already present in the cluster. The value of the array conflict $conf(i,j)$ is the sum of conflicts between vector $v_i \in B_T$ and all the vectors $\{e_{j,0}, e_{j,1}, \ldots\}$ that are the current members of cluster C_j :

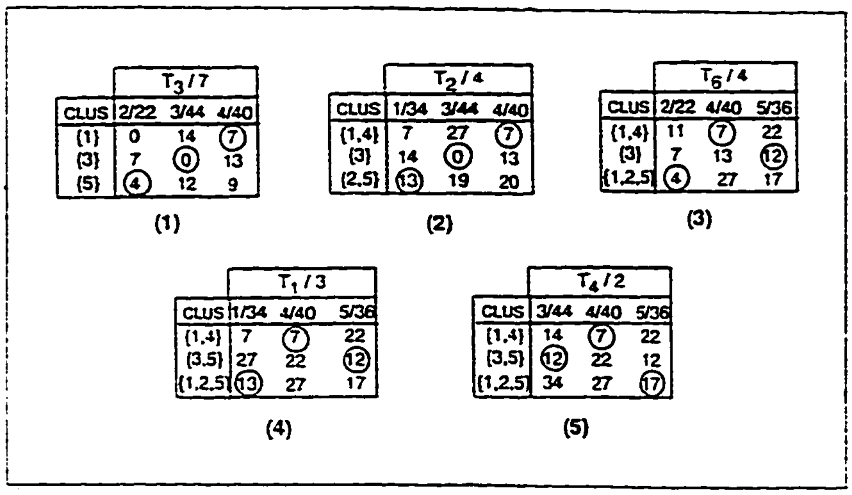
$$conf(i,j) = \sum_{e_{j,k} \in C_j} w(v_i, e_{j,k})$$

Note that vector $e_{j,k}$ is member of the union of all pattern bases and also member of at least one pattern basis. Since the clusters are initially empty, we set $w(v_i, \varnothing) = 0$ and $w(v_i, v_i) = 0$. Now, the basis vectors $\{v_0, \ldots, v_{m-1}\}$ are taken in the decreasing order of their weights. Vector v_i is inserted into cluster C_j if $conf(i,j)$ is the least among all the m clusters. However, if C_j already contains a copy of v_i, then the insertion is omitted because the copy is also representative of v_i. Next, we lock cluster C_j to ensure that no other vector from the same pattern basis will be inserted into C_j. Each cluster receives one and only one vector of each pattern basis. This guarantees the pattern basis vectors are uniformly distributed over the clusters, a necessary condition to ensure that the basis vectors of any given pattern basis will receive linearly independent values after completion of the algorithm. 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 the m basis vectors of each pattern are uniformly distributed over the m available clusters and each cluster contain basis vectors that have the least degree of conflict. Thus the heuristic is useful to partition the set of basis vectors into groups so that: 1) intra-cluster conflicts are minimized and it is more likely to cluster independent vectors, and 2) avoid as possible placing copies of the same vector in distinct clusters (duplication) or duplicate the least weighted vectors.

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 filled by 1 in each column j where $t_i \in c_i$, otherwise the row is completed with zeros. The heuristic is better explained using the previous example of 6 patterns $T_1,\dots,T_6$ and their access frequencies which are shown in Figure 4-1.

The clusters are initialized with the pattern basis that is the most frequently accessed which is T_5' and the clusters become $c_0 = \{1\}$, $c_1 = \{3\}$, $c_2 = \{5\}$. Now we consider the next pattern that is T_3 and build the conflict diagram between the clusters and T_3 as shown in the first block of



	$T_3/7$		
CLUS	2/22	3/44	4/40
{1}	0	14	$\textcircled{7}$
{3}	7	$\textcircled{0}$	13
{5}	$\textcircled{4}$	12	9

(1)

	$T_2/4$		
CLUS	1/34	3/44	4/40
{1,4}	7	27	$\textcircled{7}$
{3}	14	$\textcircled{0}$	13
{2,5}	$\textcircled{13}$	19	20

(2)

	$T_6/4$		
CLUS	2/22	4/40	5/36
{1,4}	11	$\textcircled{7}$	22
{3}	7	13	$\textcircled{12}$
{1,2,5}	$\textcircled{4}$	27	17

(3)

	$T_1/3$		
CLUS	1/34	4/40	5/36
{1,4}	7	$\textcircled{7}$	22
{3,5}	27	22	$\textcircled{12}$
{1,2,5}	$\textcircled{13}$	27	17

(4)

	$T_4/2$		
CLUS	3/44	4/40	5/36
{1,4}	14	$\textcircled{7}$	22
{3,5}	$\textcircled{12}$	22	12
{1,2,5}	34	27	$\textcircled{17}$

(5)

Figure 4-3 : Conflict Diagrams in Clustering

Column *CLUS* list the current content of the clusters. The right block contains the pattern, its weight, its basis vectors and their weights, and conflict weights between the vectors (columns) and clusters (rows). The mapping of vectors to clusters is taken in the decreasing order of vector weights. In the first block of Figure 4-3, vector 3 is taken first because it has the highest weight (44) and mapped to cluster {3} because it has the least conflict (0) with this cluster compared to other clusters (14 with {1} and 12 with {5}). Since vector 3 is already in {3} which leave the cluster unchanged. Now no more vectors from the basis of T_3 can map onto {3} because these vectors must be distributed over cluster so that each cluster will map to only one vector of a any given pattern basis.

Similarly, vectors 2 and 4 are mapped to {5} and {1} which after updating become {2,5} and {1,4}, respectively.

The overall result is the state of the clusters after the last step is completed (here conflict diagram 5). The clusters are {1,4}, {3,5} and {1,2,5} which shows that 1 and 5 are duplicated in the solution. The basis vectors of each cluster will receive the same coloring vector. As basis vectors may appear in more then one cluster, then the final coloring of each basis vector must be the sum (exclusive-or) of all the coloring vectors it received. Clusters {1,4}, {3,5} and {1,2,5} are assigned the coloring vectors (1,0,0), (0,1,0), (0,0,1), respectively. Since 1 is in the first and last clusters, then the final color of 1 must be (1,0,1) and that of 5 is (0,1,1). The storage matrix M is directly obtained from the clusters where each row of M corresponds to one cluster:

$$M = \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

One can easily verify that the restricted sub-matrices to each data pattern are all non-singular. For example, matrix M_{T_3} whose basis is $B(T_3) = \{2,3,4\}$ is the following.

$$M = \begin{array}{c} \begin{array}{ccc} 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

Building the conflict or weight matrix takes $O(n^2m\,P)$ time. Constructing the conflict table for a given pattern takes $O(m^2n)$ time and $O(m^2n\,P)$ for P patterns. This is in the worst case where every cluster has n vectors. Selecting the least conflicting cluster for one vector takes $O(m)$ time and $O(m^2\,P)$ to map all vectors of all patterns to clusters. So, the overall time complexity is $O(n^2m\,P + m^2n\,P)$.

4.5 The merge-split heuristic

One important feature of a coloring heuristic is the ability to early detect largest independent sets of node and color these sets. An independent set contains the nodes that are not related pairwise. In our approach we use the concept of largest independent set within a general node merging (MERGE) procedure that assembles independent node from the conflict graph but without coloring them. The objective of MERGE is to find early sets of independent nodes so that all the nodes that belong to a given independent set is renamed and given one single name. There are two results from this operation: 1) all the nodes which bear the new name will be colored later by the same color and, 2) the set of bases is reduced by the renaming process which reduces the complexity of the problem. For example consider two bases $b_1 = (e_1, e_2, e_3)$ and $b_2 = (e_1, e_2, e_4)$ with frequencies f_1 and f_2, respectively. Assume (e_3, e_4) forms an independent set, then we can rename (e_3, e_4) by an unused name say e_5 and the bases become $b_1 = (e_1, e_2, e_5)$ and $b_2 = (e_1, e_2, e_5)$ which can be combined into one single basis say $b = (e_1, e_2, e_5)$ with frequency $f_1 + f_2$. Later when e_5 is colored by a color say k both e_3 and e_4 will be colored by k. The benefit of this approach is to keep in the graph only one single copy of a set of nodes that can be given the same color. Delaying the coloring of new names has the benefit of handling the new name and its new weight with respect to all the other nodes which is likely to produce more adequate coloring.

The MERGE procedure works as follows: It chooses node u with the highest weight and places it in a set S. Next it selects nodes in decreasing order of weights so that each of the selected nodes is not a neighbor to any node already in S. Every time it finds such a node, it inserts it into S. This process ends when no more independent nodes can be found. The nodes of S are renamed except when S contains only u. If some nodes are renamed, it reduces the current set of bases and re-evaluate the new weights.

The second important operation is the node splitting which occurs when it becomes impossible to color a graph because it contains a clique of $m+1$ nodes or more and there is only m colors. A clique is a set of nodes that are connected to each other within the set. For example consider three bases $b_1 = (e_1, e_2, e_3)$, $b_2 = (e_1, e_2, e_4)$, and $b_3 = (e_1, e_3, e_4)$ to form a clique of 4 nodes which cannot be 3-colored. Splitting node u into two new nodes u_0 and u_1 consists of partitioning the set of bases that contain u, so that some of these bases map to u_0 and the other map to u_1. We must create a merge-inhibitor edge for (u_0, u_1) to avoid any attempt to reverse the splitting operation. This is similar to providing two different mapping of node u which indicates that the resulting storage scheme will incorporate two distinct accesses for pattern bases that contain the split node u. This defines the procedure (SPLIT).

In the above example, we split an arbitrary node say e_3 and create two new nodes e_{30} and e_{31} so that the bases become $b_1 = (e_1, e_2, e_{30})$, $b_3 = (e_1, e_{31}, e_4)$, and b_2 remains unchanged. Apparently, pairs of nodes (e_2, e_{31}) and (e_{31}, e_4) become independent sets and a 3-coloring can be found.

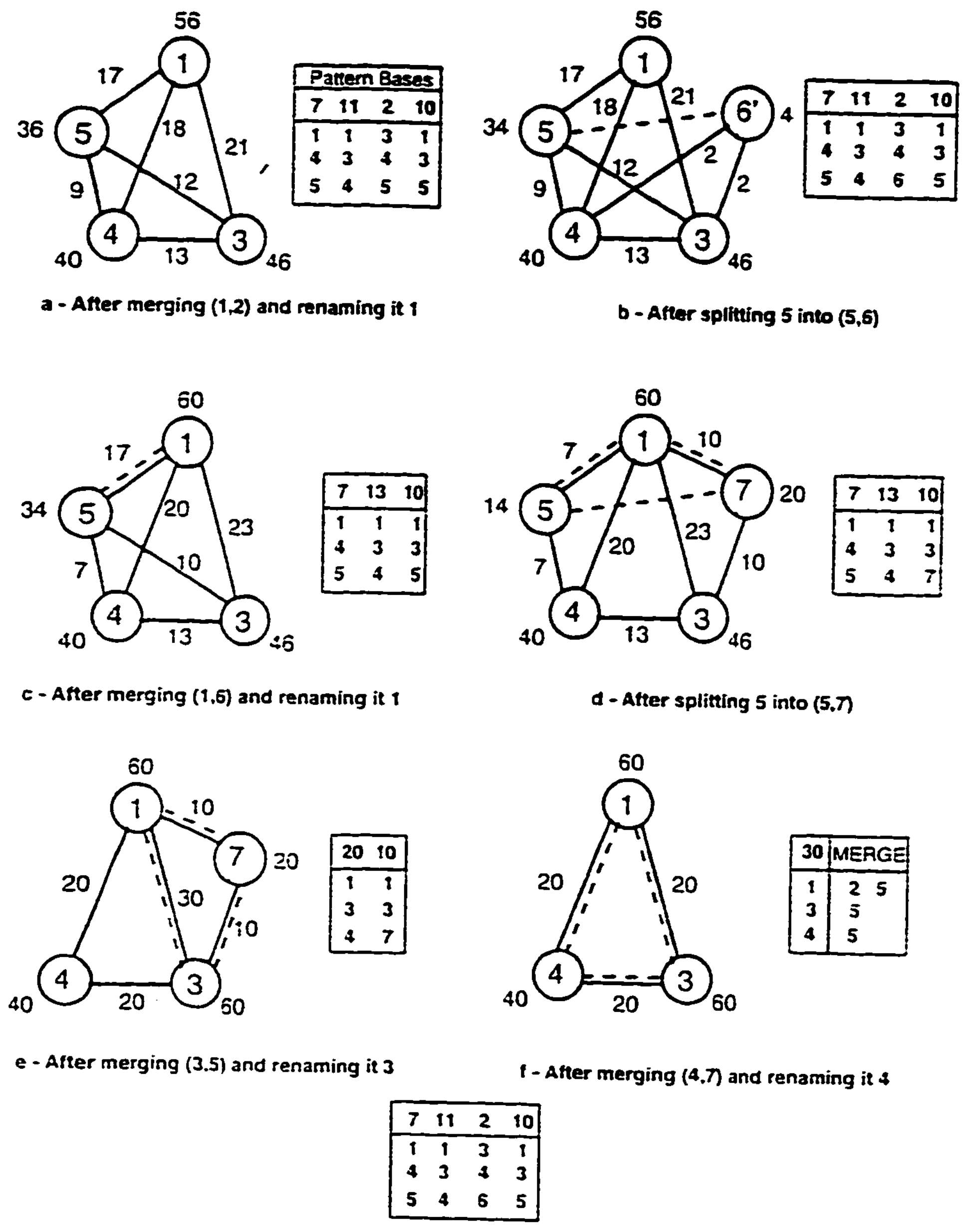
A coloring heuristic called *Merge-Split* (MS) is presented. The strategy of this heuristic is to perform alternate merge and split operations until the conflict graph become m-colorable, where m is the number of memories. The outcome of MERGE is one of the following cases: 1) reducing the number of bases but the conflict graph is still uncolorable, 2) the number of bases remains unchanged, and 3) the conflict graph becomes colorable. In the first case, there is opportunity for merging but there is still a clique of m or more nodes. In the second case, there is no opportunity for merging. In the third case, some node merging is performed which reduces the number of bases to 1. Since each base forms a clique of m nodes, then reducing the number of bases to 1 means that the problem is now m-colorable which terminates the algorithm. Now the algorithm retrieve all the previous merge and split decisions which are simple renaming operations and construct the solution based on the lastly colored m nodes. Let us consider the example of 6 patterns $T_1, \ldots, T_6$ that have been previously defined in Section 4.4. The list of these patterns and their associated conflict graph are shown in parts a and b of Figure

4-1. Edge (1,2) is not used in any base which makes its weight to be zero ($w(1,2) = 0$). The heuristic first merges nodes (1,2), rename them 1, and substitute 1 to each occurrence of 2 in the pattern bases as shown in Figure 4-4 (a) which also displays the conflict graph of the reduced set of pattern bases. The number of pattern bases reduces from 6 to 4.

To reduce the number of patterns bases further, the heuristic selects node 5 that has the least weight and splits it into two nodes 5 and 6. Now node 5 is contained in pattern bases (1,4,5), (3,4,5), and (1,3,5) as shown in Figure 4-4 (a). Since node 1 has now the highest weight and it appears in two bases out of three which all contain 5. In order to create an opportunity for merging later 1 with 6, the name 5 is changed 6 in (3,4,5) which becomes (3,4,6). Figure 4-4 (b) shows the new list of patterns and its conflict graph. A merging-preventing edge is added to (5,6) to avoid undoing the previous operation. Note that splitting a node does increases the number of pattern bases which remains four.

During the next merging step, node 1 is merged to 6 and renamed 1. The number of pattern bases becomes three. This represents the only opportunity offered at this stage and the results are shown on Figure 4-4 (c). Next, the least weighted node that is 5 is split again into nodes 5 and 7 as shown in Figure 4-4 (d). This operation is straightforward because 5 is contained in two pattern bases.

In the next merging step, nodes 3 and 5 are merged and renamed 3. Additional merge preventive edges are used for (1,3) and (3,7). This reduces the number of pattern



a - After merging (1,2) and renaming it 1

Pattern Bases			
7	11	2	10
1	1	3	1
4	3	4	3
5	4	5	5

b - After splitting 5 into (5,6)

7	11	2	10
1	1	3	1
4	3	4	3
5	4	6	5

c - After merging (1,6) and renaming it 1

7	13	10
1	1	1
4	3	3
5	4	5

d - After splitting 5 into (5,7)

7	13	10
1	1	1
4	3	3
5	4	7

e - After merging (3,5) and renaming it 3

20	10
1	1
3	3
4	7

f - After merging (4,7) and renaming it 4

30	MERGE
1	2 5
3	5
4	5

7	11	2	10
1	1	3	1
4	3	4	3
5	4	6	5

Figure 4-4 : The Merge/Split Heuristic

bases to two (Figure 4-4 (e)). Another merging opportunity is that of nodes 4 and 7 which are renamed 4. The set of pattern bases becomes one which indicate that the merge-split phases are complete and the previous conflict graph is now reduced to a clique of 3 nodes ({1},{3},{4}) which is 3-colorable (Figure 4-4 (e)). Now we need to propagate the previous decisions (from last to first) of merging and splitting in order to find the coloring solution. Merging (4,7) gives the coloring ({1},{3},{4,5}) because 7 is a copy of 5. Merging (3,5) gives ({1},{3,5},{4,5}). Merging (1,6) gives ({1,5},{3,5},{4,5}) because 6 is another copy of 5. Finally, merging (1,2) gives ({1,2,5},{3,5},{4,5}) which is the solution.

Assume {1,2,5}, {3,5}, and {4,5} are assigned (1,0,0), (0,1,0), and (0,0.1), respectively. vector 5 will be assigned the sum (exclusive-or) of (1,0,0), (0,1,0), and (0,0,1) which gives (1,1,1). This is equivalent to assigning 1's in the columns of the storage matrix M so that row 1, 2, and 3 map to {1,2,5}, {3,5}, and {4,5}, respectively. This gives the storage matrix:

$$M = \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

This storage matrix allows conflict-free access to all the patterns because each restricted matrix M_{T_i} is non-singular, where $1 \le i \le 6$. For example, the restriction M_{T_i} that is formed by columns (1,3,5) of M is non-singular:

$$M_{T_5} = \begin{array}{c} \begin{array}{ccc} 1 & 3 & 5 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

The time complexity of this heuristic is $O(n^2 m P + M N^2)$. The first term is the time to build the conflict matrix. In the second term, M is the total number of merges and splits the algorithm makes and N is the maximum number of nodes in the graph after some number of merges and splits.

4.6 Evaluation

In this section, we experimentally test and compare the three heuristics described in this chapter. These heuristics will be compared in two types of problems: access of power of two patterns and access of arbitrary strides. In each of these two types of problems, the comparison between the different heuristics will be in two criteria: cost of solutions and execution time.

4.6.1 Power-of-2 Patterns

Given a solution ($m \times n$ Boolean matrix), let C be the number of memory cycles needed to access all patterns using this matrix,

$$C = \sum_{i=1}^{P} f(T_i) \cdot c_i$$

where P is the number of patterns, $f(T_i)$ is the frequency (weight) of pattern T_i, and c_i is the number of memory cycles needed to access T_i for one time. If the restricted matrix M_{T_i} has a rank of m, then $c_i = 1$. If, however, the rank of M_{T_i} is $m-k$, then $c_i = 2^k$. This is how we compute c_i and consequently C. We define C_L as the lower bound on C:

$$C_L = \sum_{i=1}^{P} f(T_i) \cdot c_L$$

Where c_L is a lower bound on the number of cycles needed to access one pattern for one time and is chosen to be 1 memory cycle. So,

$$C_L = \sum_{i=1}^{P} f(T_i)$$

We evaluate a given solution by measuring its deviation (D) from the lower bound. This deviation is defined as follows :

$$D = \frac{C}{C_L} - 1$$

A good solution is a one with small D, and a worst solution is with $D = 2^m - 1$.

We tested the different heuristics with a range of problem sizes. A given problem i is defined by the number of memories 2^{m_i} and the number of patterns P_i in that problem and is represented by (m_i, P_i). In our experiments, we varied the number of memories from 8 to 256 $(3 \le m \le 8)$ which is typically the case for VLIW machines. The number of patterns was varied from 3 to 20 $(3 \le P \le 20)$.

(m_i, P_i) is a problem consisting of P_i patterns each consisting of m_i basis vectors. These m_i basis vectors are randomly generated out of a range of 3 m_i possible vectors. The generated vectors are normally distributed as shown in Figure 4-5. In this Figure, the y-axis is the frequency of the corresponding vector out of 500 randomly generated vectors. Normal distribution forces patterns to use neighboring vectors. This situation makes vectors and patterns more correlated and finding a solution becomes more difficult, but it is more realistic.

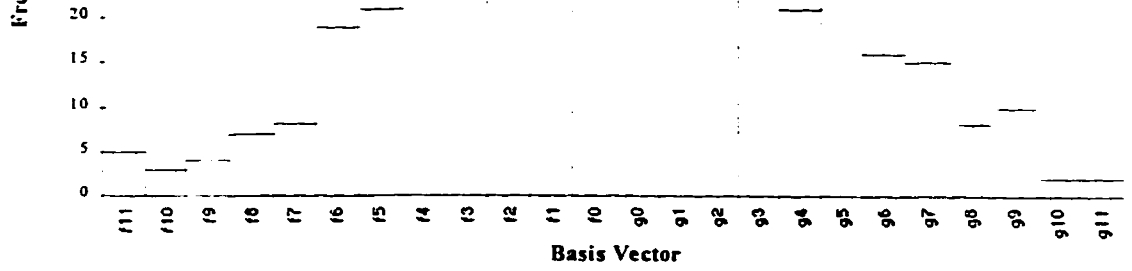


Figure 4-5 : Normal Distribution of Vectors

For every problem (m_i, P_i), 50 random instances were randomly generated. In any instance, P_i patterns are generated with weights uniformly distributed between

1 and $2 \cdot P_i$. Now, C of the problem (m_i, P_i) is computed as the average of C of the 50 instances of that problem.

4.6.2 Arbitrary Strides

In the case of arbitrary strides, given a Boolean matrix, then C_{s_i} is the average number of memory cycles needed to access the stride s_i by this matrix. In accessing an arbitrary stride s_i in 2^m memories, we will have the following addresses :

$$A(a), A(a+s_i), A(a+2s_i), \ldots, A(a+(2^m-1)s_i)$$

where a is the origin within the array. We varied the origin a from 0 to $2^m - 1$. For every origin, we computed the number of cycles needed to access this sequence of addresses and then averaged this number for all origins to get the average number of memory cycles needed to access the stride s_i, C_{s_i}. Then C of the given solution is :

$$C = \frac{\sum_{i=1}^{S} C_{s_i}}{S}$$

where S is the number of strides.

In our experiments, we varied the number of memories again from 8 to 256. The problem considered here is to construct a storage matrix that minimizes the average memory access time of the strides (1, 2, 3,..., 64). So, the cost of a given solution in the case of stride access is :

$$C = \frac{\sum_{i=1}^{64} C_i}{64}$$

Where C_i is the average number of cycles needed to access the stride i.

Throughout this thesis, we compare our approaches in stride access with Sohi's solution. Sohi [41] gave a Boolean matrix for stride access and no body in the literature gave a better solution. The matrix was for 8 memories and 12-bit array addresses and was as follows :

$$M_{Sohi} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Sohi, however, did not give a procedure to construct such matrices for arbitrary number of memories and arbitrary size of addresses.

4.6.3 Results

In the plots and tables of this section, we evaluate and compare the three heuristics. In this section we will be referring to Merge/Split, Greedy Coloring (H1), and Clustering by MS, GC, and Clust, respectively.

The first three plots show the deviation (D) of the three heuristics versus the number of patterns. Figure 4-10 compares the three heuristics in the same plot for the case of 2^8

memories. Figure 4-11 shows the execution time per one problem instance versus the number of memories (P is fixed to 20). Figure 4-12 shows the execution time versus the number of patterns (m is fixed to 8). Figure 4-13 to Figure 4-16 compare the stride access of the three heuristics and Sohi's solution. The x-axis is the stride being accessed and the y-axis is the average number of memory cycles needed to access that stride. In these plots, m was set to 3. Figure 4-17 compares the heuristics' average stride access versus the number of memories. Figure 4-18 compares the average stride access time for the heuristics and Sohi's solution for $m = 3$. The first three tables show the worst case of (D) between different instances of a problem for the three heuristics with various problem sizes in pattern access. The other three tables show the worst case, C, and the variance of the three heuristics.

It is clear from the plots that GC outperforms the other two heuristics in the case of pattern access. It may seem surprising since the quality of this heuristic in graph coloring problems is less than other heuristics. This is because in graph coloring, splitting a node directly means an increase in the solutions cost. In our problem, however, a split can increase the cost only if it leads to making two or more vectors in the storage matrix linearly dependent. Any neighbor vectors (vectors that are common in at least one pattern basis) will not be given the same color because there will be an edge between them and if needed one of them will be split and given more than one color. This split may not increase the cost and on the contrary will likely lead to a set of linearly independent vectors. So, in our problem, there is a good chance of getting a good solution by splitting.

Although MS performs well in graph coloring, it dose not in pattern access. This is due to two things. First, it dose not utilize the chance of splitting (as GC does) because it splits only when it can not convert the graph into m-clique. Second, it goes to merging whenever it is possible. It seems that merging in the context of constructing storage matrices is destructive because it means giving two nodes the same color or giving two vectors the same value.

Due to the high coupling or correlation between different vectors in the case of stride access (see Figure 4-6), GC will have to make a lot of splits before being able to color the graph. By doing so, the chance in splitting will be exhausted and extra penalized splits will be needed. This is why GC is not the best in stride access as it was in pattern access. It seems that high correlation also restricts the freedom of MS and prevents it from merging neighbor vectors and that is why MS is outperforming the other heuristics in stride access.

Clustering takes the least execution time. GC takes more time than MS dose. This is because it keeps splitting nodes and increasing the number of nodes and graph size which directly increases the execution time. MS keeps the number of nodes minimum by merging. This reduces the execution time of MS.

Frequency	Basis Vectors			
32	0	1	2	3
16	1	2	3	4
8	2	3	4	5
4	3	4	5	6
2	4	5	6	7
1	5	6	7	8
1	6	7	8	9

Figure 4-6 : Patterns Corresponding to Strides (1,2,...,64)

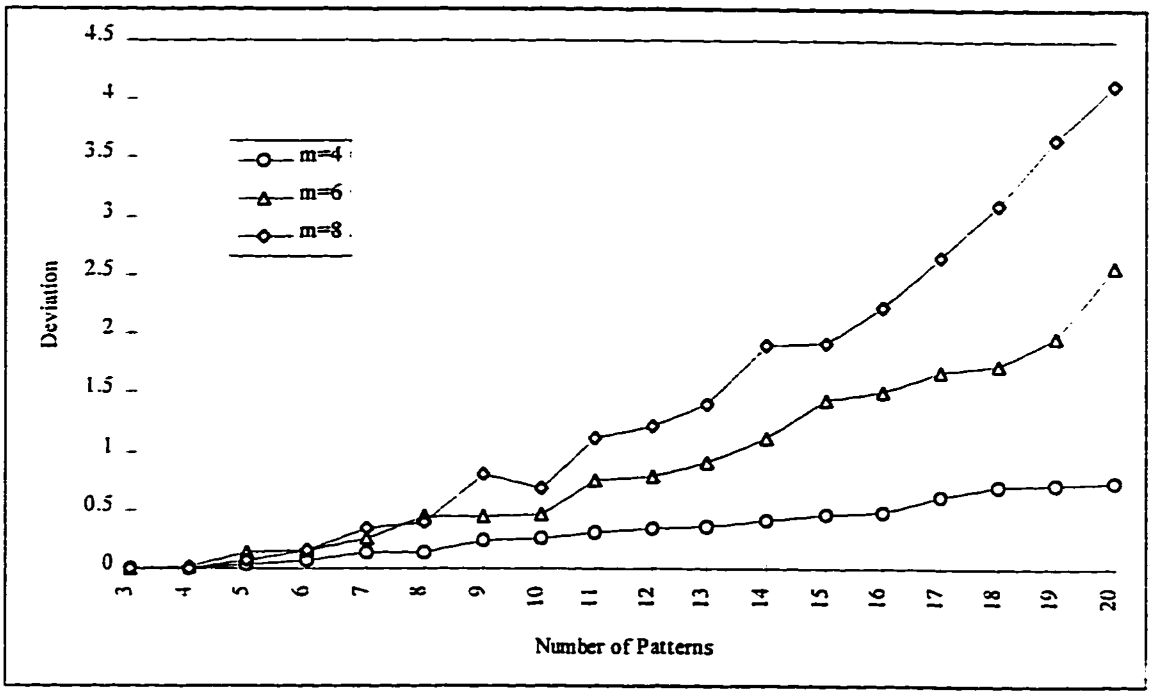


Figure 4-7 : Pattern Access of MS

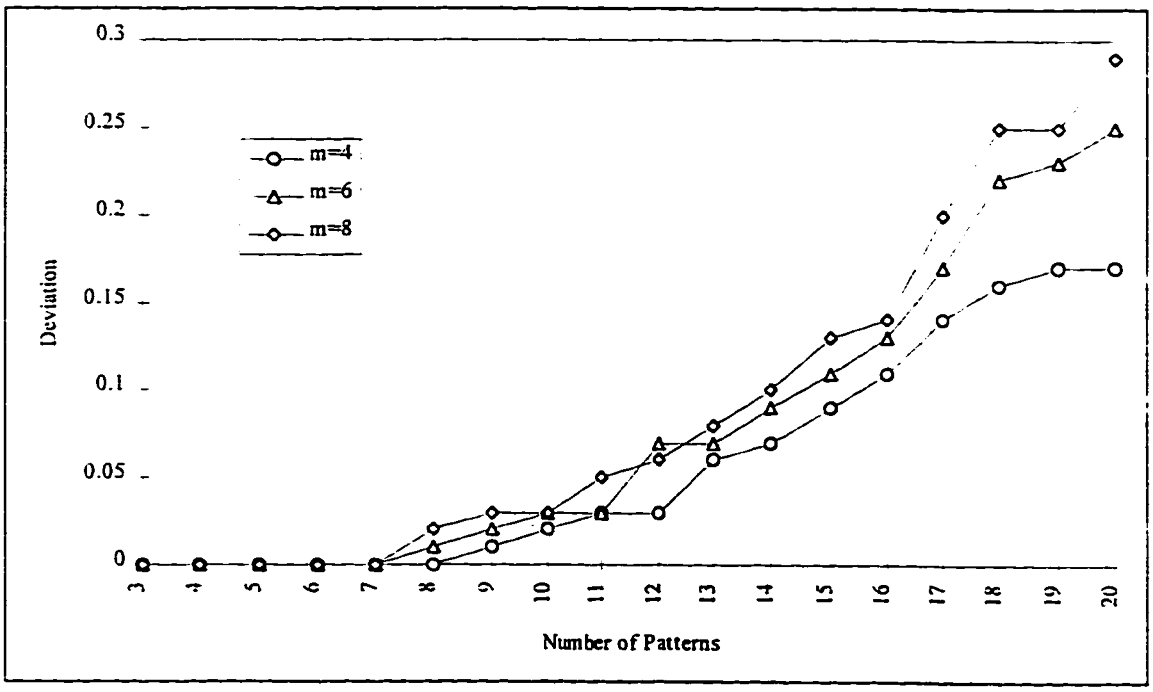


Figure 4-8 : Pattern Access of GC

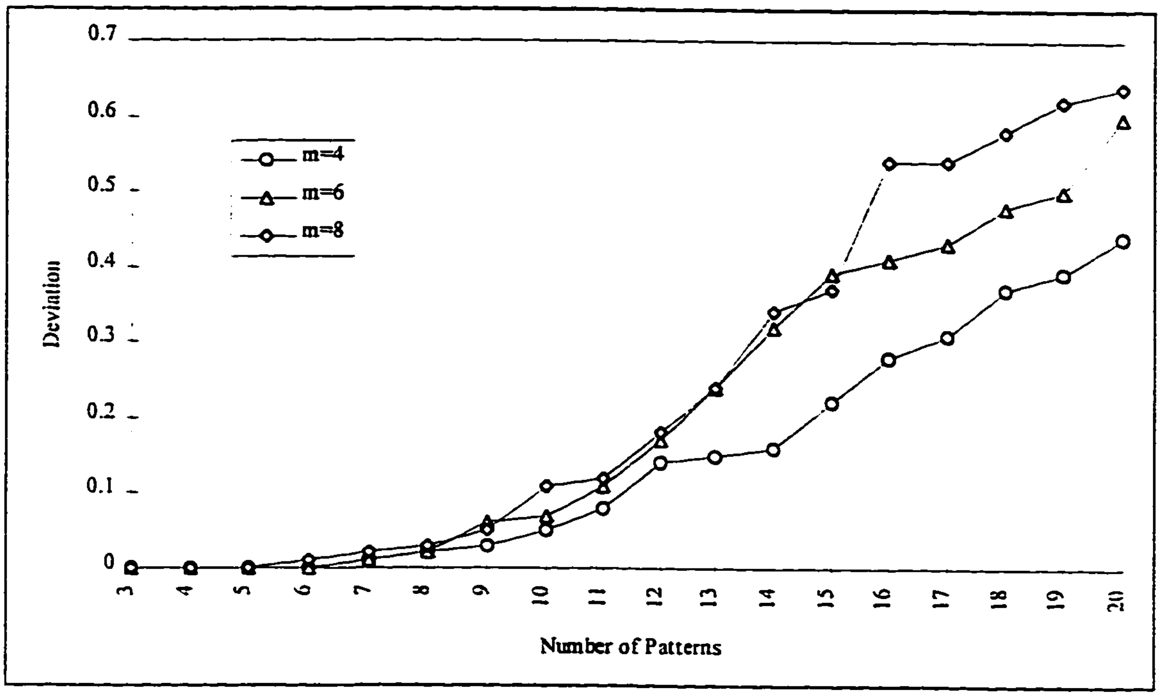


Figure 4-9 : Pattern Access of Clust

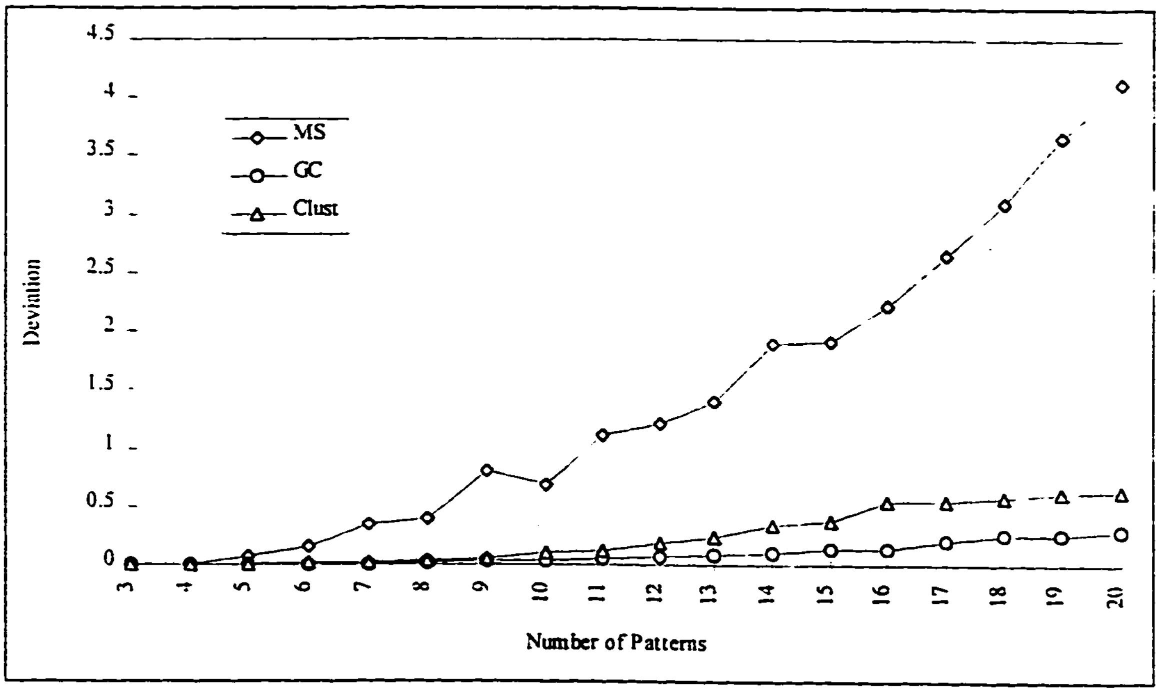


Figure 4-10 : Pattern Access for m=8

		m					
		3	4	5	6	7	8
P	3	0.00	0.00	0.00	0.00	0.00	0.00
	8	0.27	0.53	0.40	1.37	0.89	0.95
	15	0.91	0.97	2.10	3.67	3.23	4.38
	20	0.92	1.51	3.03	4.16	7.07	8.20

Table 4-1 : Worst Case Pattern Access of MS

		m					
		3	4	5	6	7	8
P	3	0.00	0.00	0.00	0.00	0.00	0.00
	8	0.11	0.14	0.13	0.00	0.17	0.00
	15	0.32	0.30	0.47	0.36	0.48	0.23
	20	0.57	0.49	0.60	0.71	0.56	0.84

Table 4-2 : Worst Case Pattern Access of GC

		m					
		3	4	5	6	7	8
P	3	0.00	0.00	0.00	0.00	0.00	0.00
	8	0.18	0.22	0.47	0.23	0.41	0.83
	15	0.85	0.88	1.03	0.97	0.94	1.12
	20	1.09	1.06	1.34	1.41	1.38	1.22

Table 4-3 : Worst Case Pattern Access of Clust

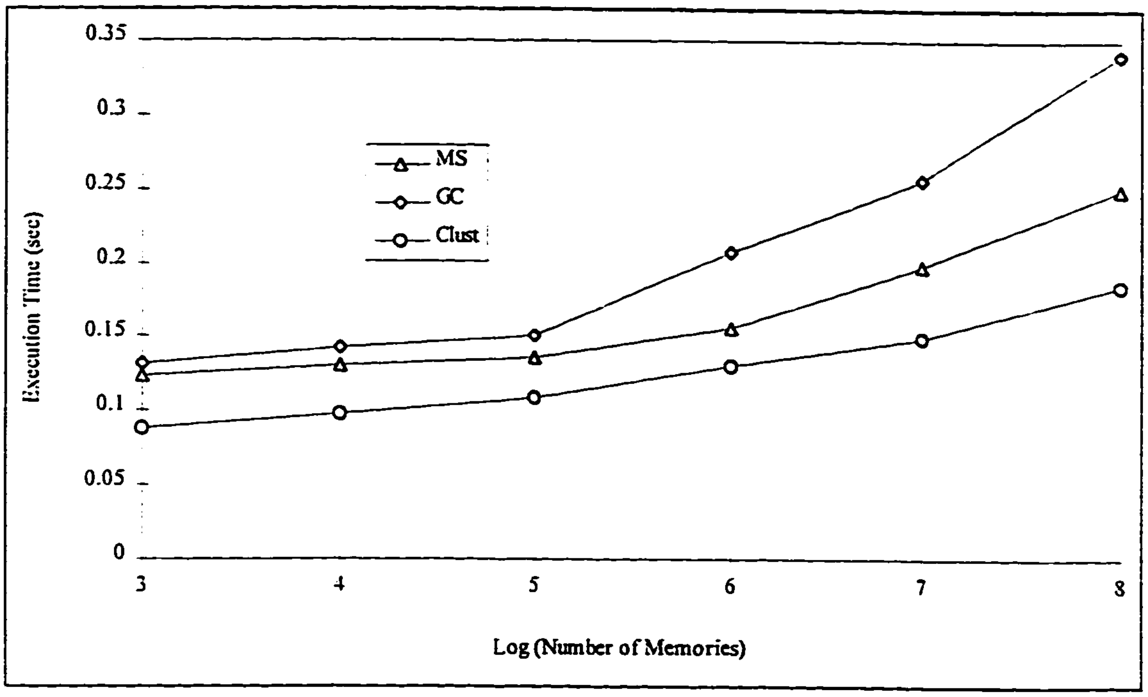


Figure 4-11 : Execution Time Vs. m for Pattern Access with P=20

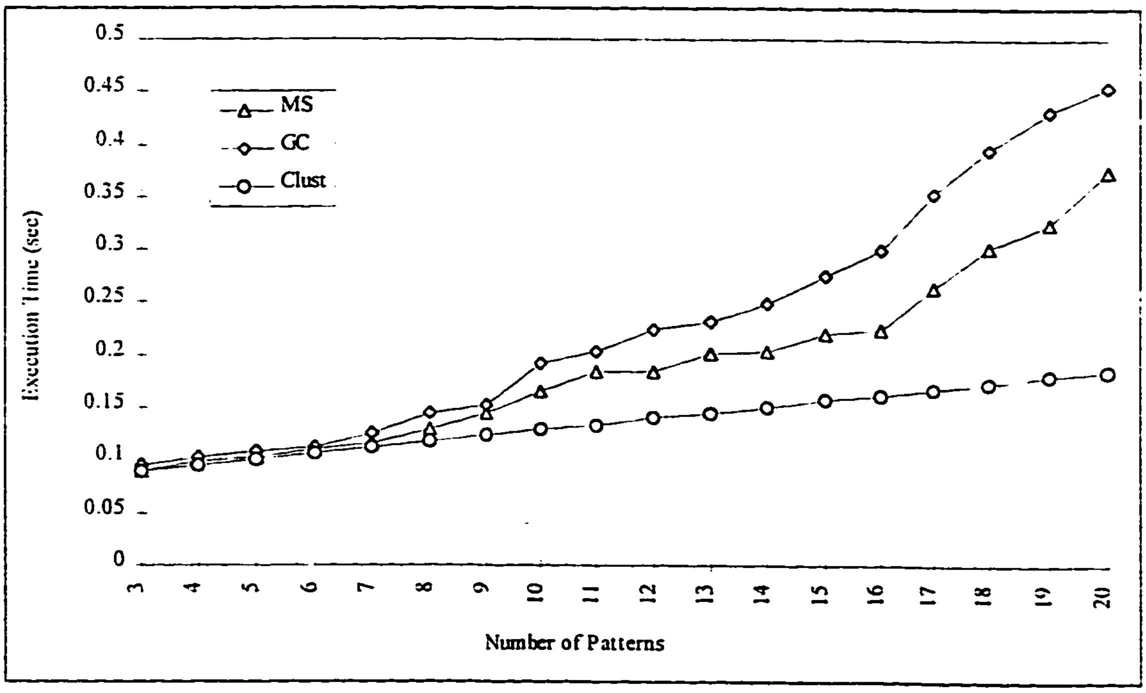


Figure 4-12 : Execution Time Vs. P for Pattern Access with m=8

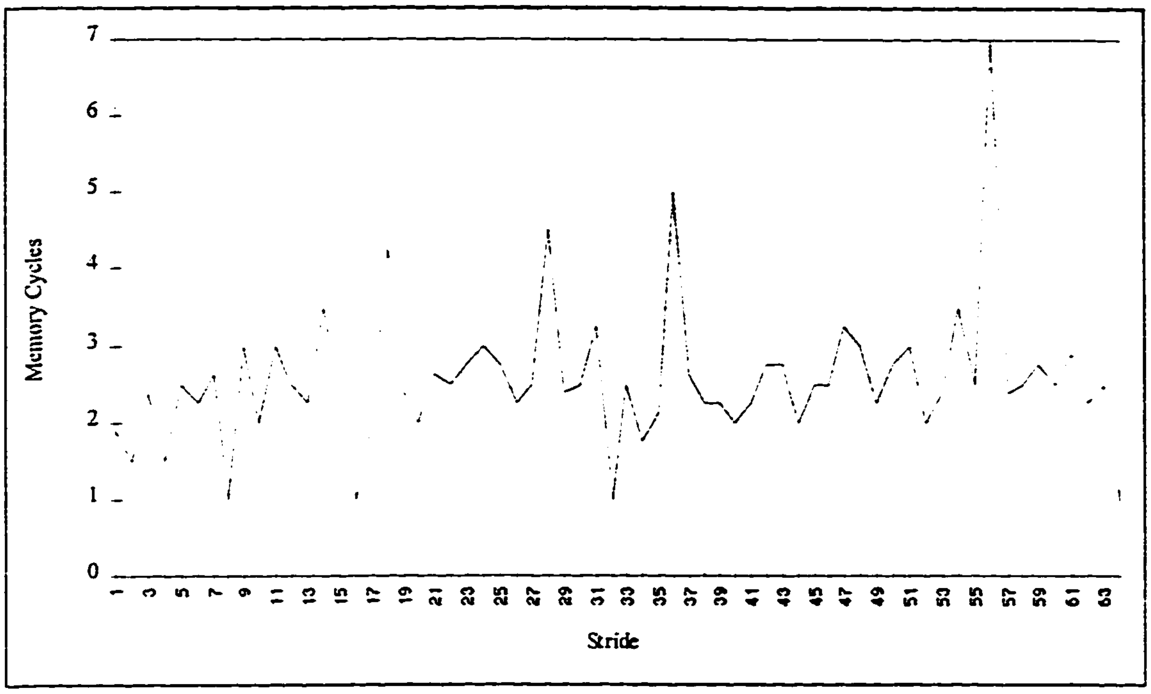


Figure 4-13 : Stride Access of MS for m=3

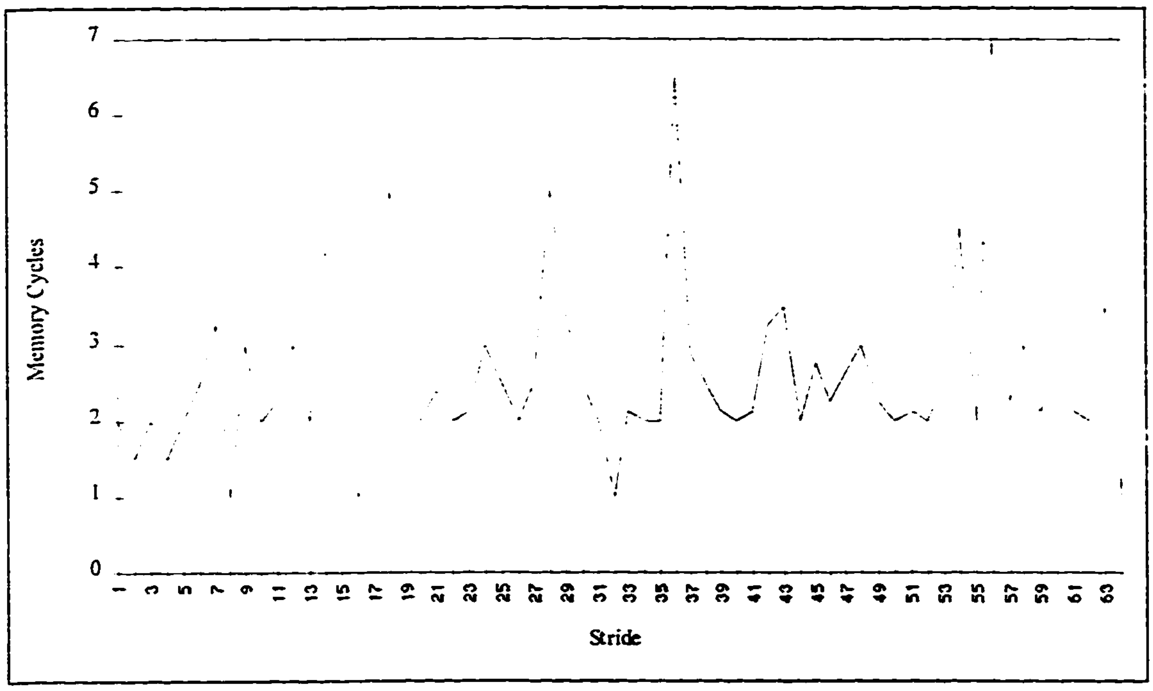


Figure 4-14 : Stride Access of GC for m=3

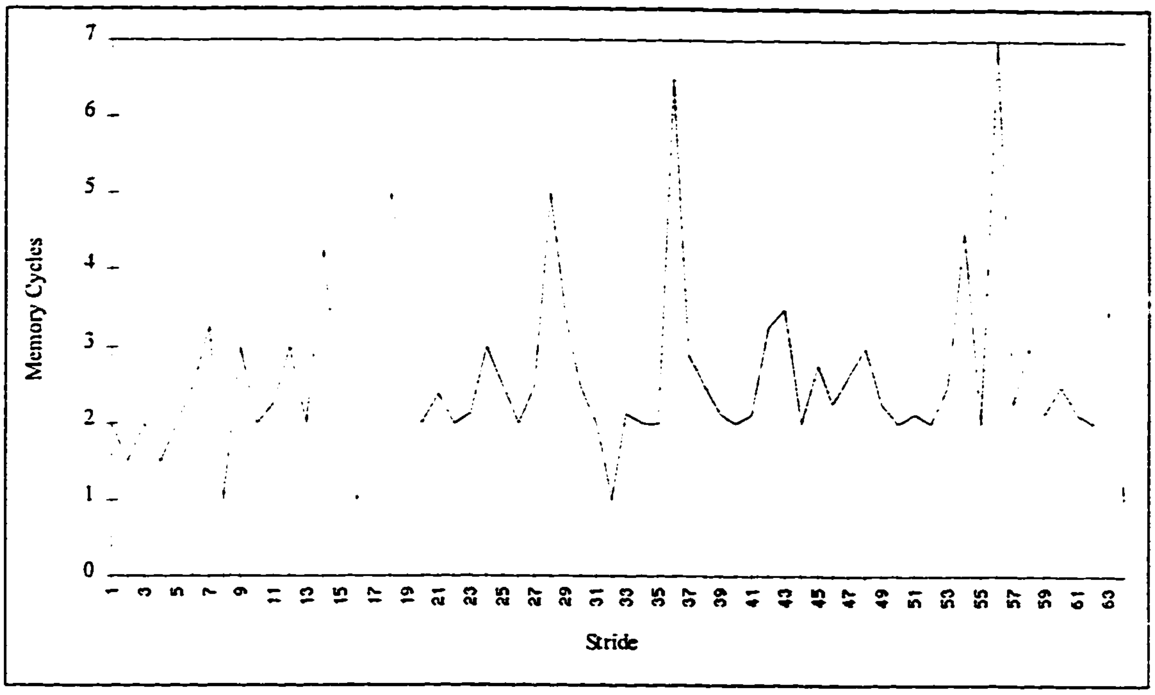


Figure 4-15 : Stride Access of Clust for m=3

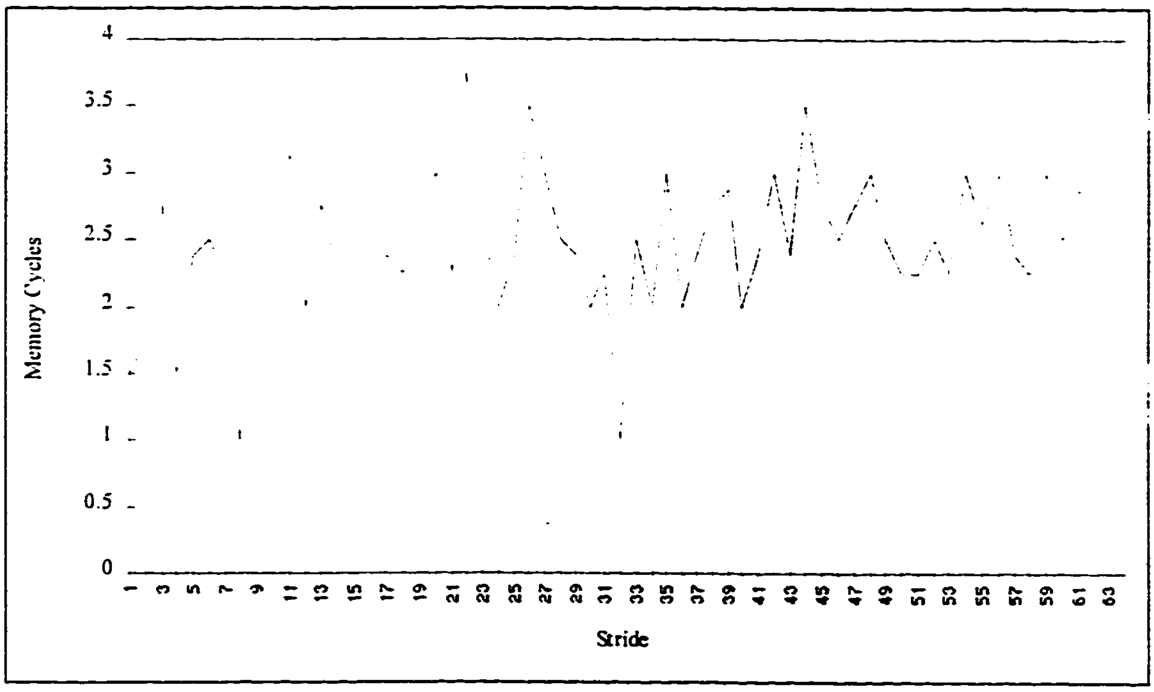


Figure 4-16 : Stride Access of Sohi's Solution

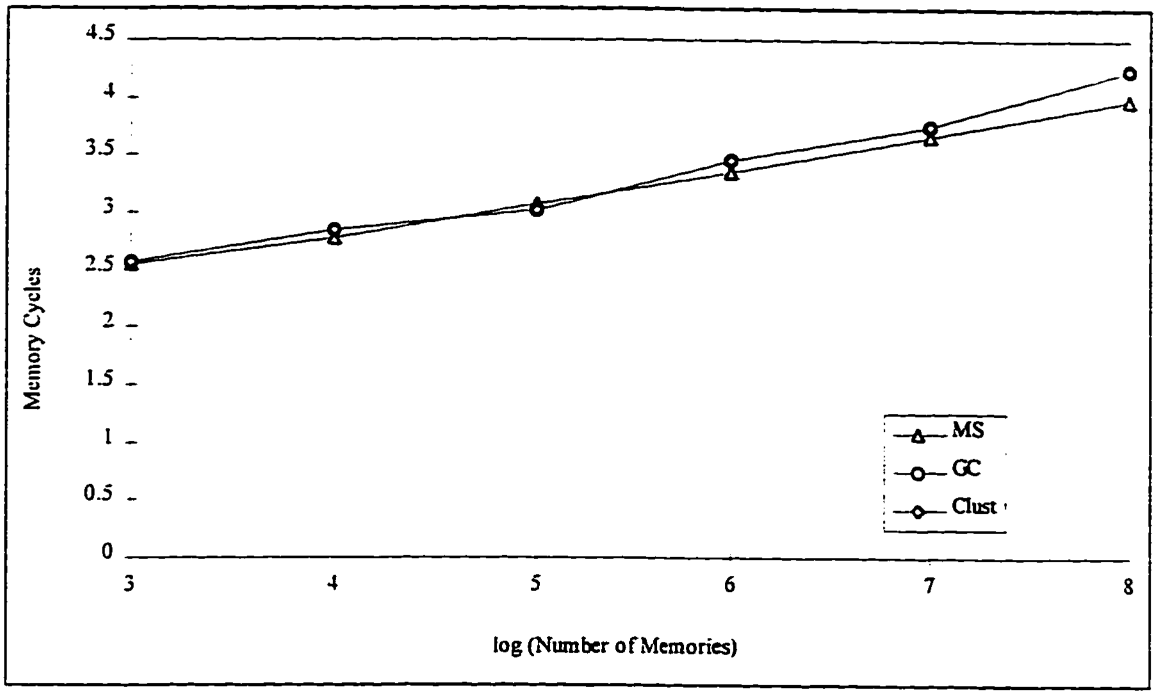


Figure 4-17 : Average Stride Access

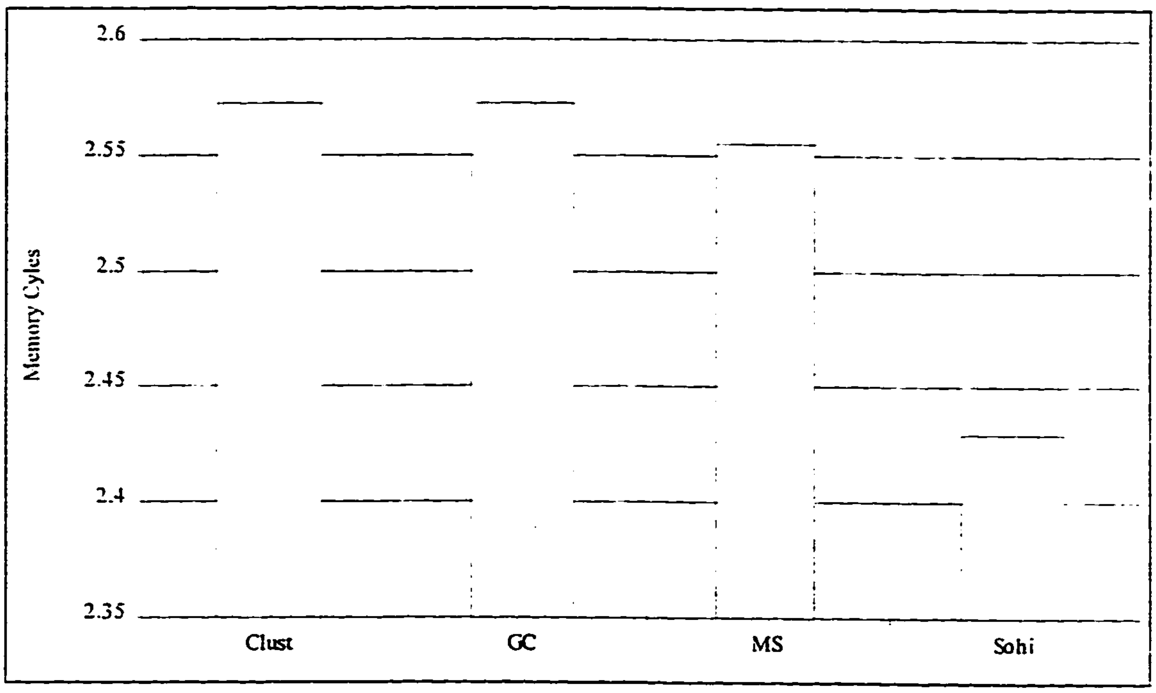


Figure 4-18 : Average Stride Access for m=3

	m					
	3	4	5	6	7	8
Worst	7.00	4.25	5.31	4.97	5.31	6.72
Avg	2.56	2.77	3.07	3.35	3.66	3.97
Var	0.82	0.41	0.68	0.58	0.75	1.26

Table 4-4 : Stride Access of MS

	m					
	3	4	5	6	7	8
Worst	7.00	8.25	11.50	14.13	13.25	17.47
Avg	2.57	2.84	3.02	3.45	3.75	4.23
Var	1.23	1.85	3.28	6.37	2.30	0.58

Table 4-5 : Stride Access of GC

	m					
	3	4	5	6	7	8
Worst	7.00	8.25	11.50	14.13	13.25	17.47
Avg	2.57	2.84	3.02	3.45	3.75	4.23
Var	1.23	1.85	3.28	6.37	2.30	0.58

Table 4-6 : Stride Access of Clust

Chapter 5

5. Neural Networks

Neural Nets (NN's) have been used to solve a variety of problems in diverse fields. Such fields include combinatorial optimization, vision, pattern recognition, classification, and many other fields. NN's are characterized by their ability to find solutions faster than solutions found by conventional approaches. They can be implemented both in hardware and in software, but are much faster when implemented in hardware. In this section, I will use a neural net to build Boolean storage matrices. The neural net was tested for power-of-2 patterns as well as for arbitrary strides. In the case of arbitrary strides, the NN found solutions that are of the same quality as other techniques, but was much faster. With power-of-2 patterns, the NN was also faster and found good solutions for problems of large sizes.

5.1 Background

A variety of approaches have been in use to solve different problems in the fields of science and engineering. Some problems have satisfactory algorithmic solutions. Algorithmic approaches, however, may not be suitable for other problems (like NP problems). For such problems, other heuristic approaches were employed. General purpose heuristics like Simulated Annealing, Simulated Evolution, and Genetic Algorithms have been successful in many fields. There are, however, other types of problems that do not have specific algorithmic or heuristic procedures to solve them. Such problems like image recognition, speech recognition and similar problems. The human brain is known to be successful and better than the most powerful computer in this type of problems. So, to develop approaches that are successful in these problems, it is logical to study the human brain and try to imitate it. This is actually what is happening in the field of Neural Networks.

Neural Networks are artificial simulations of the human nervous system. They imitate the operation of human brains by having similar structures and operation to solve problems that the human brain can solve. Although our problem of synthesizing storage schemes have heuristic techniques, we will use a neural net in trying to increase speed and quality. Neural nets can be mainly divided into two categories [24]:

- *Biological type*. In this type, artificial nets are in high resemblance of human neural nets in structure and operation. These nets have similar cell structure of the biological neurons shown in Figure 5-1 [22]. Such nets are usually used in learning machines, classifiers, and pattern recognition. To achieve these functions, the net has also to operate in a way similar to that of the human brain.

- *Application-driven* type. In this class, neural nets are affected more by the nature of the application which they perform rather than the biological neural nets. There are many models of this class of which the most known is the Hopfield model. Our discussion will be limited to application-driven which are of interest to us.

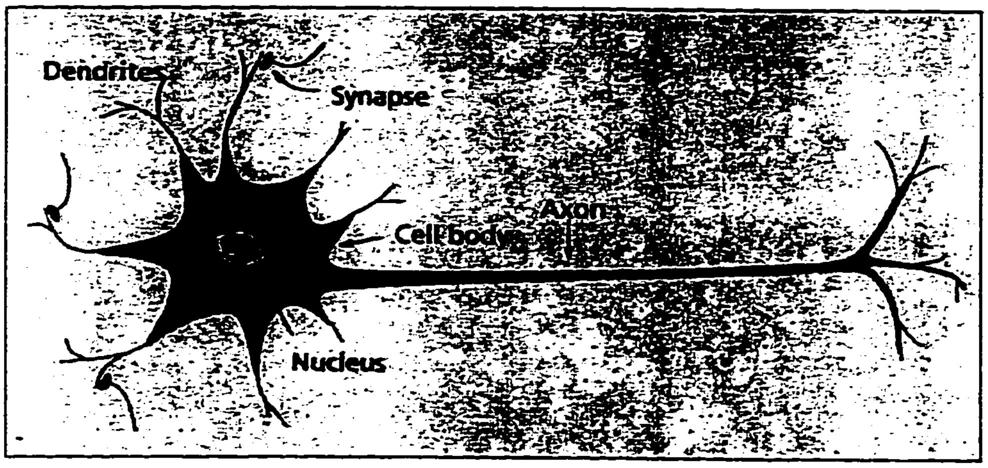


Figure 5-1 : Biological Neuron

5.2 Why Use a NN ?

In this section, we will briefly discuss the advantages of using a NN in constructing storage matrices rather than using other approaches. we can identify the following merits of using NN's in our problem :

1. NN's have the potential of massively parallel processing. Since in a parallel or hardware implementation of NN's neurons operate in parallel, we can gain a speedup linear in the number of neurons.

2. They are faster than other approaches. As will be seen in the comparisons, the software implementation is much faster than all other approaches except the clustering heuristic which is almost of the same speed.

3. Hardware implementation is more feasible than other approaches. Since a hardware neuron operates faster than its software simulation by several orders, we can get speedups of several orders beyond the parallelism speedup.

4. The object oriented nature of NN's (being composed of independent discrete neurons) makes them easily scaleable and flexible for architecture improvement.

5. Current VLSI technologies are in favor of making hardware implementation of NN's cheaper and more flexible.

6. In our problem, the NN works directly on the storage matrix without the need for graph representation or any kind of conversion as needed in other approaches.

7. The simplicity of the architecture as will be seen makes the found solutions very attractive.

5.3 Biological Neural Nets

The human nervous system is composed of basic nervous cells or neurons as shown in Figure 5-1. A neuron is composed of three main parts: the cell body and two tree-like branches: dendrites and axon. The cell body contains the nucleus which has the plasma and other chemical materials needed to be produced by the neuron. Dendrites work as the receivers of the neuron. So a neuron receives signals from other neurons through its dendrites and transmits its signals to other neurons through the axon. The axon is long branch coming out of the cell body and branching at its end into strands and sub strands. To deliver the signal from one neuron to another, strands have to meet with dendrites. The junction point between the strands of one neuron and the dendrites of another is called the synapse [22,24].

There are two types of synapses: excitatory and inhibitory. When the signal of one neuron is transmitted through a synapse to the dendrite of another neuron, it will either excite or inhibit the output of the other neuron depending on the type of the synapse. Every synapse is characterized by a specific synaptic strength or weight. The significance of an input signal on the output of a neuron is proportional to the weight of the synapse through which it was received. Synaptic weights characterize the operation of a given NN and varying them may completely change the operation of the net. In biological neural nets, synaptic weights change in the long term as signals pass through them. This change is believed to be responsible for learning and memory in human brains [22,24].

The human brain contains about 10^{11} neurons. Every neuron is connected to about 10^3 to 10^4 other neurons which means that there are about 10^{14} to 10^{15} connections in the brain. Signals in biological neural nets are about 1 million times slower than signal in modern computers, though, human brains can do jobs like image recognition hundreds or thousands of times faster and more accurate than computers. This is due the massive parallelism in human brains, where 10^{11} processors or neurons work in parallel and the high coupling where every neuron is connected to 10^3 to 10^4 other neurons [22,24].

5.4 Artificial Neural Nets

An artificial NN is a collection of interconnected Neurons. Every neuron is an abstracted model of the biological neuron. Figure 5-2 shows a simple neuron model. It mainly consists of three parts: input synapses, body, and output synapse. Similar to the biological case, input synapses carry signals from other neurons and the output synapse transmits the output of the neuron to other neurons. A specific synaptic weight is associated with every input synapse.

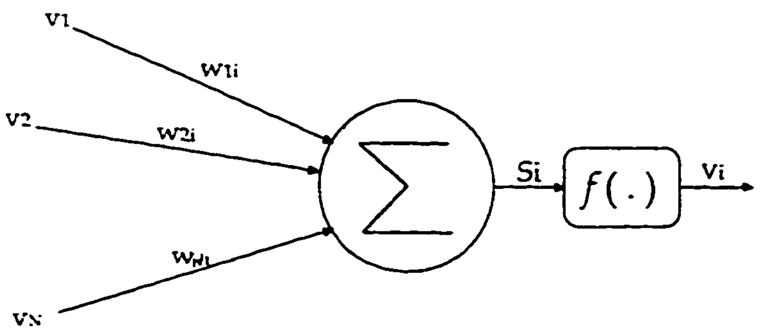


Figure 5-2 : Artificial Neuron

The neuron body is responsible for the main job of a neuron. It performs a summation operation of the weighted input synapses. Then, it outputs a nonlinear function of the weighted sum S_i (for neuron *i*) and the neuron threshold θ_i. Every neuron has a number of input synapses carrying the output of other neurons to it. The weight w_{ij} of the input synapse s_{ij} coming from neuron n_i to neuron n_j represents the significance of the output of n_i on the output of n_j. Let v_i be the output of neuron n_j Then the output of n_i is computed as follows :

$$v_i = f(\sum_{j=1}^{N}(w_{ji}v_j) - \theta_i)$$

where N is the total number of neurons and f is a nonlinear function. Figure 5-3 shows four possible forms of the function f. The unit step function shown in (a) is the most widely used function with various neural nets models.

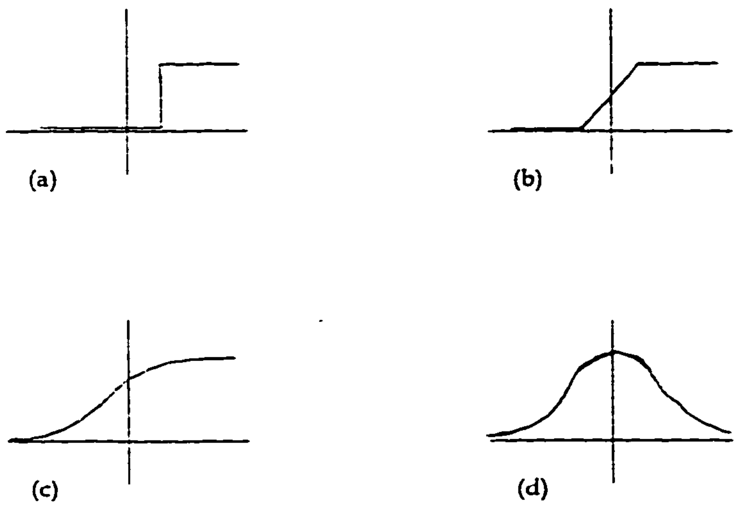


Figure 5-3 : Forms of Output Function

The neuron model described above is the standard model in most of the neural nets in various applications. The difference between different neural net models is in the way neurons are connected (NN architecture) and the way the weights are set and adjusted. To describe the architecture, the neural net can be abstracted as a directed graph where nodes are neurons and directed edges are synaptic links. Neural network architectures can be grouped into two classes [24]:

1. *feed-forward* networks: in which neurons are arranged in layers without feedback. If the net is represented as a directed graph, then the feed-forward networks are those graphs without loops.

2. *feedback* networks: which are nets that have feedback connections or the corresponding graphs contain loops.

In feed-forward networks, for any input pattern, there is a single output pattern. The output pattern is found in one pass over the layers of the network. Having feedback connections in feedback networks makes the network iterate several times and give several intermediate output patterns before reaching a stable output pattern. The set of all inputs and outputs of the network define the state of the network. When the outputs of the network change, inputs of some neurons accordingly change which moves the network into another state. This way the network takes several iterations of changing inputs and outputs until it reaches or converges into a stable state.

There are several models of each of the classes described above. The most well-known models of the feedback networks are: Competitive networks, Kohonen's networks, and Hopfield networks. Every model of these has special fields of use. The Hopfield model is mostly used in combinatorial optimization problems. The network architecture that we will use in our problem belongs to this model. So, in the next section we describe the Hopfield model before presenting our neural net.

5.5 The Hopfield Model

One of the most known application-driven models is the Hopfield model. This model is mostly used in optimization problems. It is characterized by the feedback between deferent layers of the NN (Figure 5-4 [22]). Having feedback, implies that the NN may take several number of iterations to converge or to be stable.

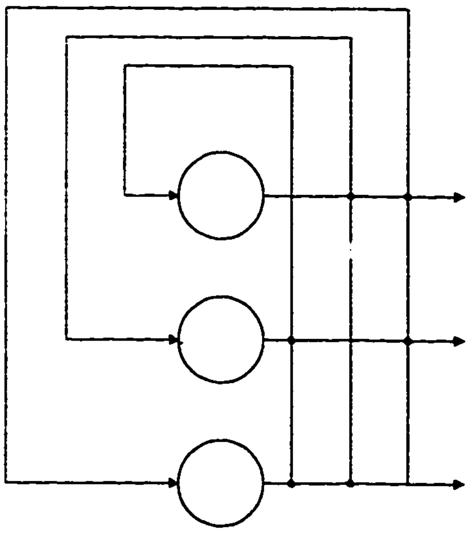


Figure 5-4 : Feedback Model

To fully describe any NN, it is enough to give two things. First is the weight matrix W where $W(i,j)$ is the weight w_{ij} of the synapse s_{ij}. So, if we have N neurons then W is an $N \times N$ *2-D* array. Second is the threshold *1-D* array θ of size N., where $\theta(i)$ is the threshold of the neuron n_i. These two things define the architecture of the NN. After the architecture, remains the operation dynamics.

Let the output of neuron n_i at iteration *t* be v_i^t and V^t be the array of all neuron outputs, i.e., $V^t(i) = v_i^t$. To find the output v_i^{t+1} of neuron n_i at iteration *t*+1, we need first to compute the weighted sum S_i of the inputs to that neuron. S_i is defined as follows:

$$S_i = \sum_{j=1}^{N} (w_{ji} v_j^t)$$

Now, v_i^{t+1} is defined as follows :

$$\begin{aligned} S_i > \theta(i) &\Rightarrow v_i^{t+1} = 1 \\ S_i < \theta(i) &\Rightarrow v_i^{t+1} = 0 \\ S_i = \theta(i) &\Rightarrow v_i^{t+1} = v_i^t \end{aligned}$$

This is just another way of representing the nonlinear function in Figure 5-3 (a). If we look at S_i, we can see that it is the *i*th number in the matrix *S*, where

$$S = W \times V^t$$

So, the whole NN update process can be represented by this matrix multiplication operation and the comparison between every element in *S* with the corresponding element in θ. There are two ways, however, of this updating process: serial and parallel. In the serial update, whenever a new output v_i^{t+1} is computed, it is stored into $V^t(i)$ so that at the end of the update process V^{t+1} is equal to V^t. This approach guarantees fast convergence, but may not give the best results. In the parallel update, when v_i^{t+1} is computed, it is stored into $V^{t+1}(i)$ and $V^t(i)$ is not updated. This way, the NN takes long time to converge and may not converge. However, the solutions found by the parallel update are usually better. The NN converges, in both update approaches, when it reaches an iteration t where V^t is equal to V^{t-1}, i.e., when the tth iteration does not make any update on V^{t-1}.

To systemize the design of multilayer networks, Hopfield used a mathematical model that characterizes all networks of this type. This mathematical model is helpful in network convergence analysis. Hopfield used a mathematical quantity and called it the energy function of the network. His energy function is as follows :

$$E = -\tfrac{1}{2}\sum_i \sum_j w_{ij} v_i v_j$$

It had been shown that when a network converges, it will be in a local optima of this function E. The energy function. makes the Hofield model most suitable for optimization problems. In such problems, the designers job consists of two things. First, he should make a decoding mechanism which will decode a given network output pattern into a solution of his problem. Second, he should map his objective function into the energy function, so that his objective function will be optimized when the network converges. Mapping the objective function is achieved by setting the synaptic weights which are the remaining variables in the energy function.

Using the energy function is not the only way to design a Hopfield network. The network can be intuitively structured in a way that will lead the network to a solution as will be seen in our case.

5.6 Constructing the matrix

Our aim in synthesizing a storage matrix is to make the restricted matrix of every pattern nonsingular (in the optimal case). Designing an approach that will guarantee this condition is very complex and may not be possible. So, we will put other criteria that will give near optimal solutions and imply feasible network architectures. A matrix is nonsingular if and only if its vectors or columns are linearly independent. We will replace this condition by other conditions that are more flexible but are likely to lead to nonsingular matrices as will be seen :

1. All columns are nonzero.

2. All rows are nonzero.

3. Giving a value to one vector reduces the chance of other vectors in the same restricted matrix to take the same value.

Having a zero vector will lead to the case of trivial linear dependence. In other words, if an $m \times m$ matrix has k zero vectors, then its rank will be at most $m-k$. Conditions 1 and 2 will prevent this by preventing having zero columns or rows. If two vectors are not similar then they are linearly independent. So, if we guarantee that in a matrix there are no two similar vectors, we are sure that any two vectors are linearly independent. This, however, is not enough to guarantee a nonsingular matrix, because a vector may be a linear combination of more than one vector. In other words, to guarantee that an $m \times m$ matrix is nonsingular, we have to have a group of m linearly independent vectors rather than a group of two. Now, we have two linearly independent (not similar) vectors and we generate a third vector such that it is not similar to any of them then there is a good chance that the three vectors are linearly independent. In Figure 5-5, vector a was randomly selected and vector b was selected so that it is not similar to vector a. Now, if we want to select a third vector c to build a 3×3 matrix, we are left with 6 possible vectors as shown in the Figure. The zero vector will not be selected because of condition 1. Clearly, of the 5 remaining vectors only that framed vector will lead to a singular matrix. So, there is an 83% chance of having a nonsingular matrix. The network will not even choose that vector because it is similar to the ORing of the other two vectors as will be seen in the next section. This will make the chance even higher. This example just shows the idea, though we may not be lucky in other cases. The criteria used in constructing this matrix is to select a vector that is not similar to already selected vectors. We chose condition 3 to satisfy this criteria. Condition 3 will reduce the chance of having similar and linearly dependent nonzero vectors. In the next section, we will show how a neural network will realize these three conditions and build a storage matrix. In section 5.8, we will propose an alternative network to build the matrix in a different way.

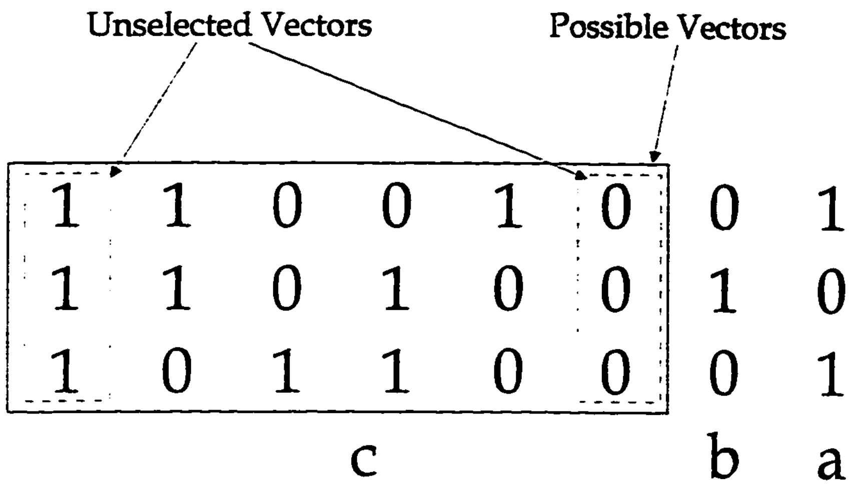
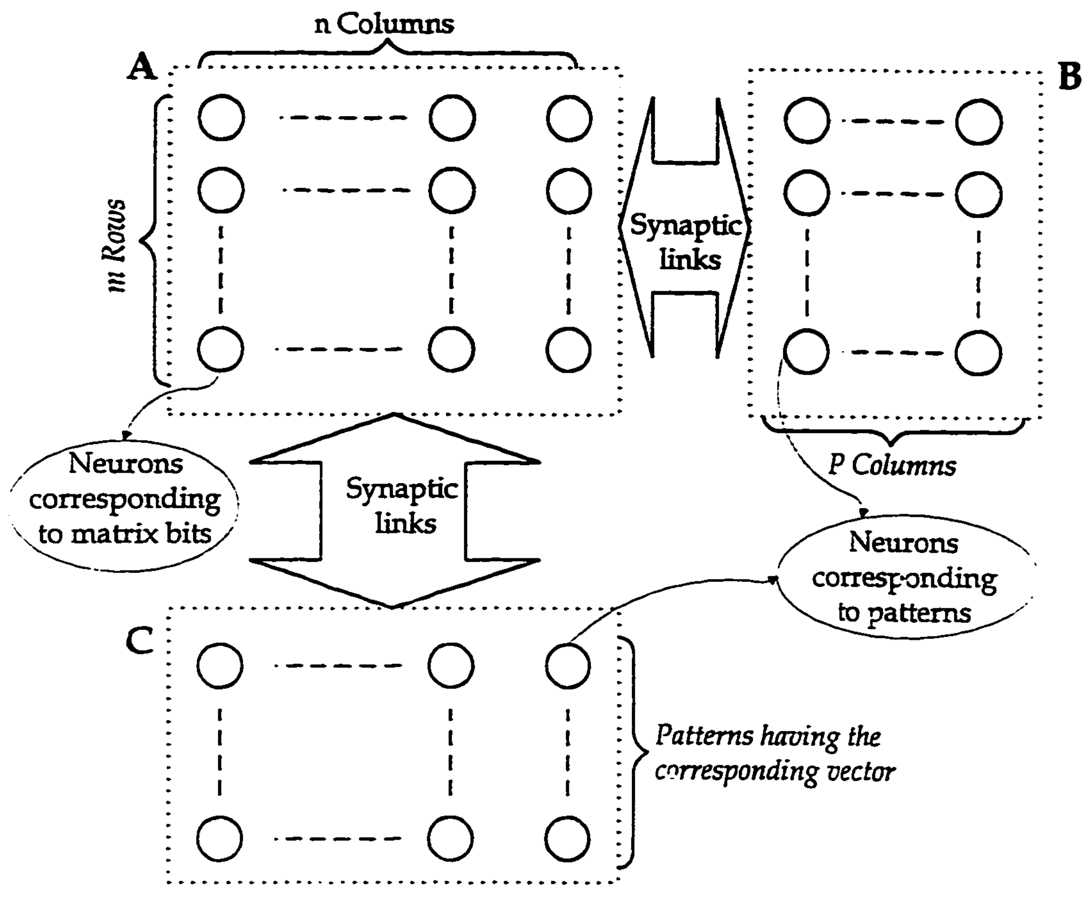


Figure 5-5 : Constructing Dissimilar Vectors

5.7 Proposed Architecture I

In this section, we will show how to construct a neural net which will construct the storage matrix depending on the three conditions stated in the previous section. The net consists of three main blocks: A, B, and C (Figure 5-6). Block A is an image of the corresponding storage matrix. For every bit in the storage matrix, there is a corresponding neuron in block A. So, block A consists of m rows and n columns. This block will contain the final solution when the network converges to a stable state, i.e., an arbitrary bit in the storage matrix will be 1 if the corresponding neuron outputs a 1 and will be 0 otherwise. All neurons in this block will have a threshold of 0. Block B consists of m



n Columns
A
B
m Rows
Synaptic links
Neurons corresponding to matrix bits
Synaptic links
P Columns
Neurons corresponding to patterns
C
Patterns having the corresponding vector

Figure 5-6 : Architecture I Layout

rows and P columns, where P is the total number of patterns. In a given row of this block, every neuron represents one of the patterns. All neurons in this block have a threshold of -0.5. Block C consists of n columns. The number of neurons in every column in this block is not fixed. The number of neurons in column i is equal to the number of patterns to which v_i (the vector corresponding to column i) belongs. In other words, for every pattern that has v_i in its basis, there is a corresponding neuron in column i of block C. Again, neurons in this block have a threshold of -0.5.

All connections in the net are interblock. In other words, there are no links within a given block. As shown in Figure 5-6, there are connections between blocks A and B and between blocks A and C. Let us look at the connections between blocks A and C . For simplicity we will describe the connections for one column and other columns are connected in the same way. The output of every neuron in a column of block C will be connected to the inputs of all neurons in that column of block A. The weight of these connections is W_p which is the weight of the corresponding pattern. Also, the output of every neuron in that column of block A is connected as an input to every neuron in that column of block C. The weight of these connections is -1. These connections are shown in Figure 5-7 for one neuron in block C. The connections between blocks A and B are similar. Here, however, a neuron (corresponding to pattern T_j) in row i of block B is not connected to all neurons in row i of block A. It is, however, connected to the subset of those neurons in columns corresponding to vectors in the basis of the pattern T_j. The number of those neurons is m and the connections between them and the neurons in block B are like those of block C and are shown in Figure 5-7.

In the previous paragraph we have described the architecture of the neural net. Now, we will describe the way it is updated. The network is updated in a serial way. It is, however, slightly different from the standard serial update method. In our update, at every iteration we select the neuron n_i whose $|S_i - \theta_i|$ is maximum. If this neuron is to be updated (from 0 to 1 or from 1 to 0) then it is updated, otherwise the next neuron is considered until one neuron is updated. After updating a neuron, the inputs of other neurons that are connected to the output of this neuron are updated and the next iteration starts. When the net reaches an iteration where no neurons need to be updated, the algorithm halts. The algorithmic form of this update procedure is shown in Figure 5-8.

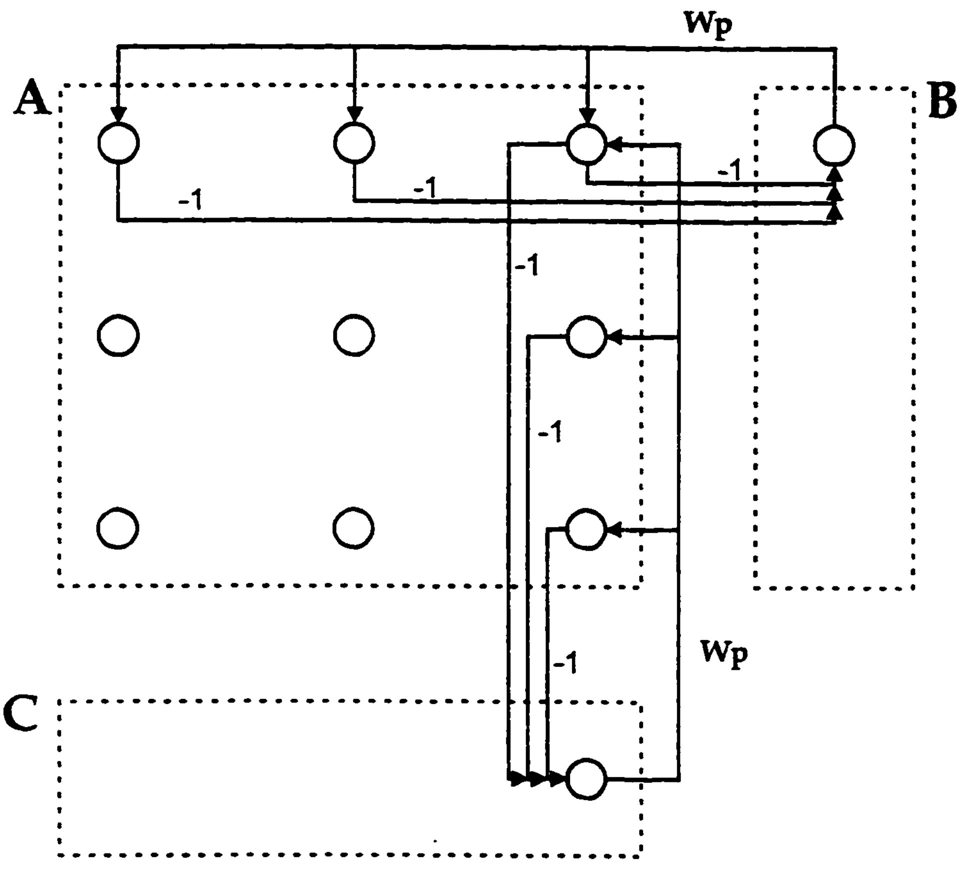


Figure 5-7 : Architecture I Connections

```
1.for all $i$ do
          $v_i = 0$;
2.Stable = 0;
3.While (! Stable )
          {
           Stable = 1;
           for all $i$ do
                    $S_i = \sum_{j=1}^{N} (w_{ji} v_j) - \theta_i$
           sort neurons in decreasing order of $S_i$.
           $i = 0$;
           While (Stable)
                  {
                   if( $S_i > 0$ and $v_i = 0$ )
                      {
                       Stable = 0;
                       $v_i = 1$;
                      }
                   if( $S_i < 0$ and $v_i = 1$ )
                      {
                       Stable = 0;
                       $v_i = 0$;
                      }
                   $++i$;
                  }
          }
```

Figure 5-8 : The Update Algorithm

The connections between blocks A and C will guarantee condition 1. This is because a neuron in block C will output a 1 as long as its inputs are all zeros (since its threshold is -0.5). Outputting a 1 will excite one of the neurons in the corresponding column of block A to output a 1. When one of these neurons (in block A) outputs a 1, it will force the outputs of all neurons in the same column of block C to output zeros. This will prevent forcing more than a 1 in the same column of block A or in the storage matrix. In the same way, connections between blocks A and B will force condition 2.

If a vector in the matrix gets a 1 in one of its rows, that will reduce the chances of another vector in the same restricted matrix to have a 1 in the same row. This is because at least one of the neurons in that row of block B will go off and hence the input sum of these vectors in that row will be less. This will guarantee condition 3. This way, we can see that the proposed network architecture will satisfy the three conditions of section 5.6. Figure 5-9 shows how the storage matrix is built through the network iterations for the same example used in the previous chapter. The row above every matrix shows the corresponding basis vectors. Figure 5-10 shows the restricted matrix of every pattern. As can be seen from the Figure, all matrices are nonsingular except the restricted matrix of P6.

The time complexity of this neural net is $O(N^2 + IN)$, where N is the number of neurons and I is the number of iterations of the network. The N^2 term is due to the for loop in step 3 of the algorithm. In every iteration, the algorithm makes a search to find the neuron with maximum S which makes the IN term. From our experiments we found that I can be quite accurately approximated by $I = m + n$.

$$\begin{array}{|ccccc|} \multicolumn{1}{c}{1} & 4 & 5 & 3 & \multicolumn{1}{c}{2} \\ \hline 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(a)

$$\begin{array}{|ccccc|} \multicolumn{1}{c}{1} & 4 & 5 & 3 & \multicolumn{1}{c}{2} \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(b)

$$\begin{array}{|ccccc|} \multicolumn{1}{c}{1} & 4 & 5 & 3 & \multicolumn{1}{c}{2} \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(c)

$$\begin{array}{|ccccc|} \multicolumn{1}{c}{1} & 4 & 5 & 3 & \multicolumn{1}{c}{2} \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \hline \end{array}$$

(d)

$$\begin{array}{|ccccc|} \multicolumn{1}{c}{1} & 4 & 5 & 3 & \multicolumn{1}{c}{2} \\ \hline 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \hline \end{array}$$

(e)

$$\begin{array}{|ccccc|} \multicolumn{1}{c}{1} & 4 & 5 & 3 & \multicolumn{1}{c}{2} \\ \hline 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ \hline \end{array}$$

(f)

$$\begin{array}{|ccccc|} \multicolumn{1}{c}{1} & 4 & 5 & 3 & \multicolumn{1}{c}{2} \\ \hline 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ \hline \end{array}$$

(g)

$$\begin{array}{|ccccc|} \multicolumn{1}{c}{1} & 4 & 5 & 3 & \multicolumn{1}{c}{2} \\ \hline 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ \hline \end{array}$$

(h)

Figure 5-9 : Building The Storage Matrix

$$\begin{array}{ccc} 1 & 4 & 5 \\ \hline 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{array}$$

P1

$$\begin{array}{ccc} 1 & 3 & 4 \\ \hline 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{array}$$

P2

$$\begin{array}{ccc} 2 & 3 & 4 \\ \hline 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{array}$$

P3

$$\begin{array}{ccc} 3 & 4 & 5 \\ \hline 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}$$

P4

$$\begin{array}{ccc} 1 & 3 & 5 \\ \hline 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{array}$$

P5

$$\begin{array}{ccc} 2 & 4 & 5 \\ \hline 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{array}$$

P6

Figure 5-10 : The Restricted Matrices

5.8 Proposed Architecture II

This architecture is also based on the idea of trying to make vectors in restricted matrices linearly independent by reducing the chance of similarity. The network will use a concept similar to the concept of force directed optimization. In this technique, every vector tries to push away neighbor vectors (vectors in the same restricted matrix) to have different values form the one it got. In the previous architecture, similar action was done, but with external forces coming from blocks *C* and *B* to block *A*. Here, these forces are internal which will reduce the number of neurons used in the network. In this architecture, there is only one block corresponding to block *A* in the previous architecture (Figure 5-11). In this block, every vector exerts forces on other vectors to push them away. The force of a vector is proportional to its weight. In the network, this means that the output of neurons belonging to a high weighted vector will be fed as inputs to neurons in neighbor vectors with weights of magnitude proportional to the weight of that vector. These weights are negative to inhibit (push) other vectors.

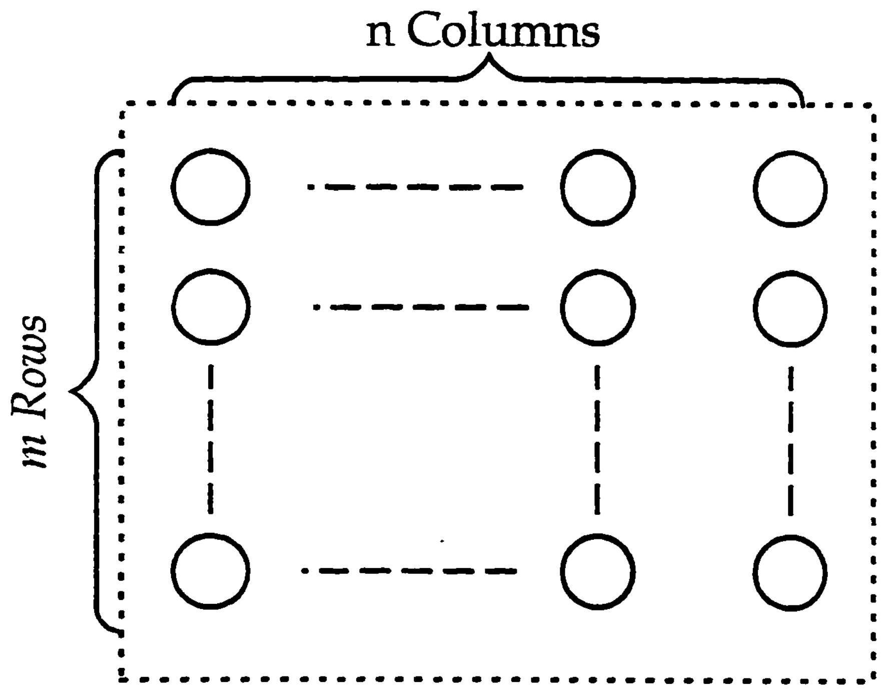


Figure 5-11 : Architecture II Layout

In this network, a neuron is connected to neurons in the same column and some of the neurons in the same row only. The output of every neuron is fed to all other neurons in the same column with a weight of W_{tp}, where W_{tp} is the total weight of all the patterns to which the corresponding vector belongs. Figure 5-12 shows the connections for one neuron. These connections prevent assigning ones to more than one row in a vector. In the coloring sense, these inhibitory synaptic links can be thought of as a way to prevent giving a node (vector) more than one color (a 1 in a row). In coloring heuristics, the least weighted vector is split if it is needed to split any node. In this network, having synaptic links of weight $-W_{tp}$ implies that high weighted vectors (large W_{tp}) will have a small chance of being split (getting 1 in more than one row).

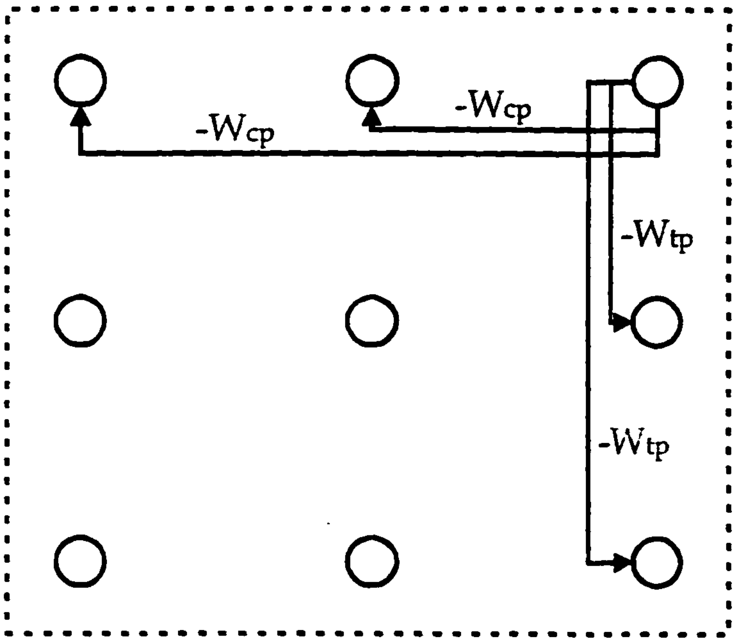


Figure 5-12 : Architecture II Connections

In the horizontal direction, any two neurons n_i and n_j in the same row are connected with weights $w_{ij} = w_{ji} = W_{cp}$, where W_{cp} is the total weight of all patterns that are common for the two vectors corresponding to the columns having the two neurons. If a vector (column in the net) gets a 1 in a row, these horizontal links will prevent other columns with common patterns to have 1's in the same row. This corresponds to giving connected nodes different colors in the coloring heuristics.

After defining the weights of all synaptic links, we are left with setting the thresholds. To give the priority of output update to neurons in high weighted vectors, the threshold will be set to $-W_{tp}$. So, at the beginning when all neurons outputs are 0, $S_i = W_{tpi}$ and the highest weighted vector (highest W_{tpi}) will be updated first, corresponding to coloring the highest weighted nodes first. The update procedure of this network is the same as the one used in the previous section

5.9 Evaluation

In this section we will use the same data sets used in the previous chapter to test the two NN architectures AI and AII for power-of-2 patterns as well as for arbitrary strides. Here we also use the same set of plots and tables for the NN architectures.

From the first Figure, we can see that AI strongly deviates for $P = 2$, 3, and 4 and then drops and starts to increase slowly and smoothly. The reason of this behavior is that for small number of patterns, less significant effect comes to block A from blocks B and C and hence the network becomes quite loose and behaves more randomly. AII dose not have this problem because of the strong coupling between neurons. We can see that in the next Figure, AII has a smooth behavior from the beginning.

Figure 5-15 shows that AII outperforms AI for up to 10 patterns. After that, A I wins. Changing the number of patterns has very little effect on AI which makes it suitable for large problems. In stride access, the two architectures have exactly the same performance.

AII takes much more execution time because it has more connections to be considered in the update procedure. Moreover, AI update procedure can be greatly simplified and accelerated by using logical operations rather than arithmetic operations. This can be done by ORing the outputs of neurons in one column and feeding the result to the corresponding neurons in block C. The same thing is done in the horizontal direction.

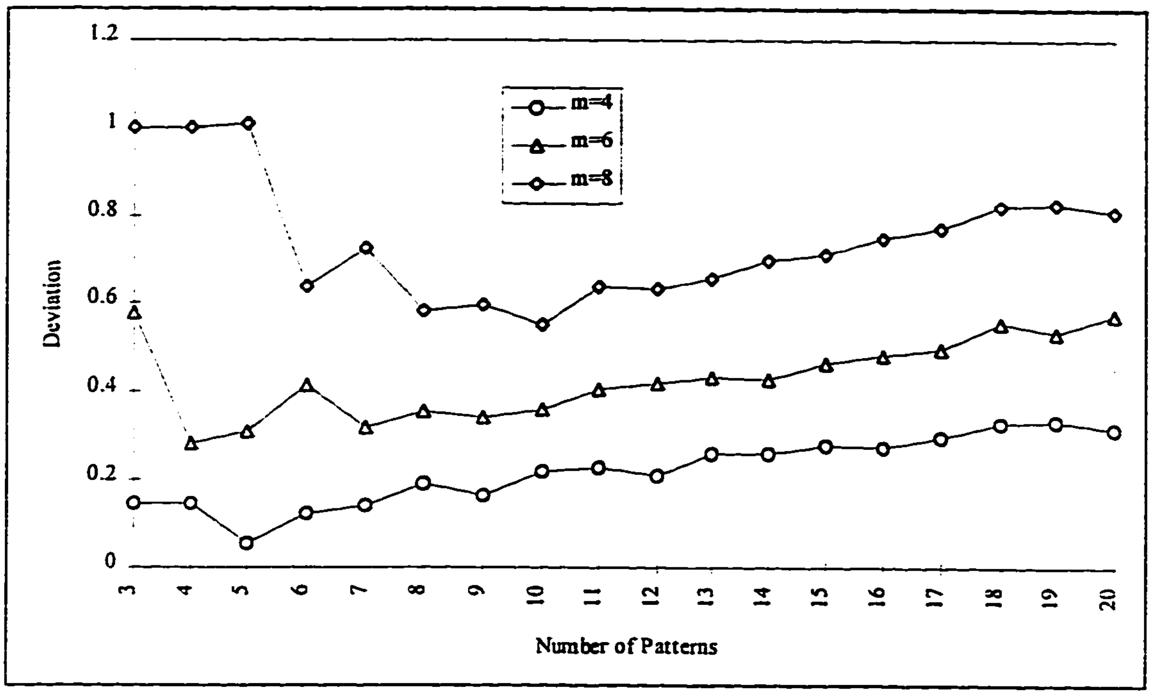


Figure 5-13 : Pattern Access of AI

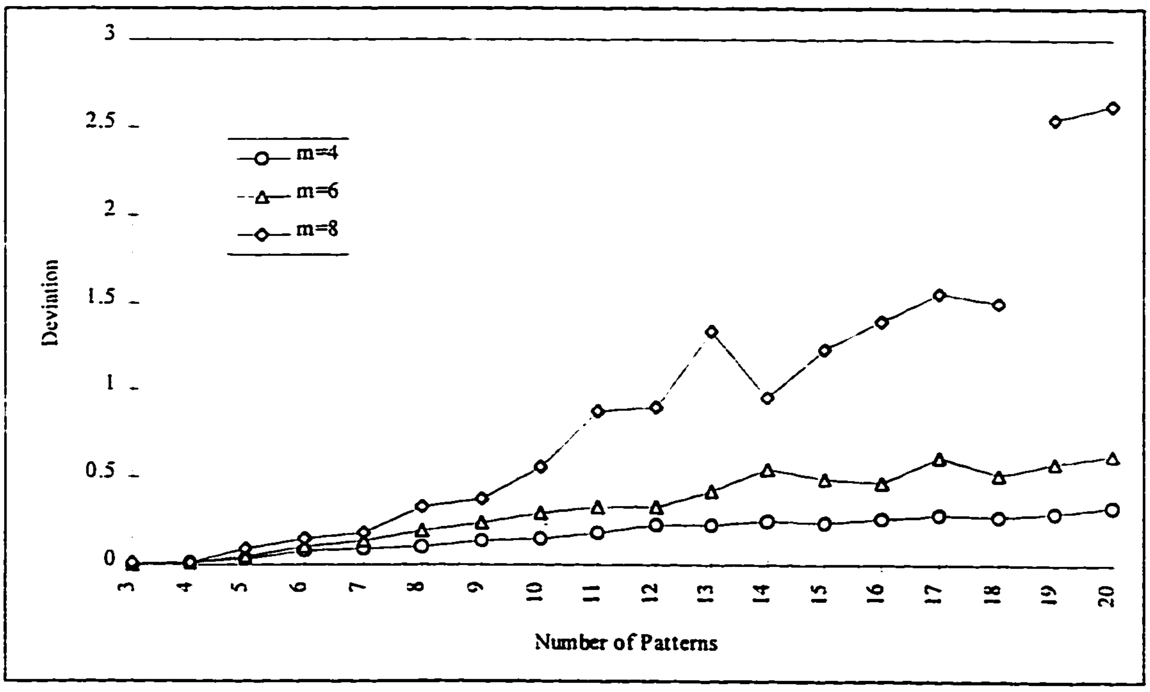


Figure 5-14 : Pattern Access of AII

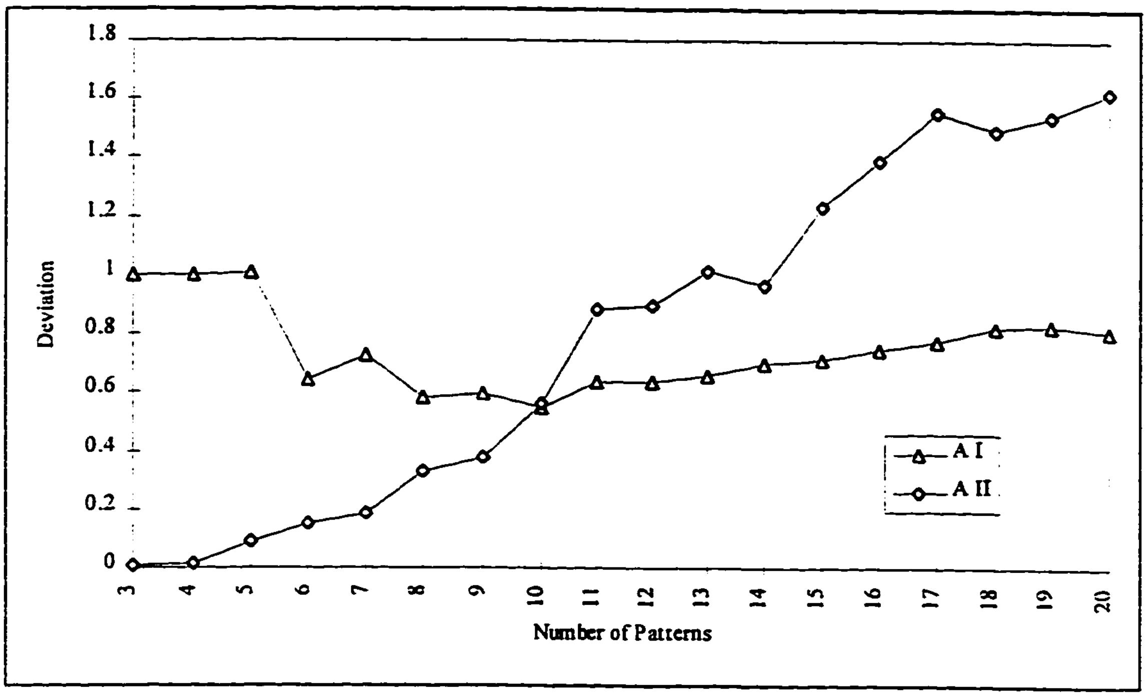


Figure 5-15 : Pattern Access for m=8

		m					
		3	4	5	6	7	8
P	3	1.00	2.00	3.00	3.00	7.00	7.00
	8	0.58	0.57	0.75	1.16	1.44	2.36
	15	0.69	0.74	0.71	0.98	1.10	1.49
	20	0.70	0.73	0.82	1.45	1.34	2.13

Table 5-1 : Worst Case Pattern Access of AI

		m					
		3	4	5	6	7	8
P	3	0.00	0.17	0.13	0.00	0.00	0.00
	8	0.20	0.28	0.27	0.23	0.28	0.29
	15	0.29	0.28	0.33	0.45	0.37	0.41
	20	0.28	0.35	0.33	0.47	0.50	0.44

Table 5-2 : Worst Case Pattern Access of AII

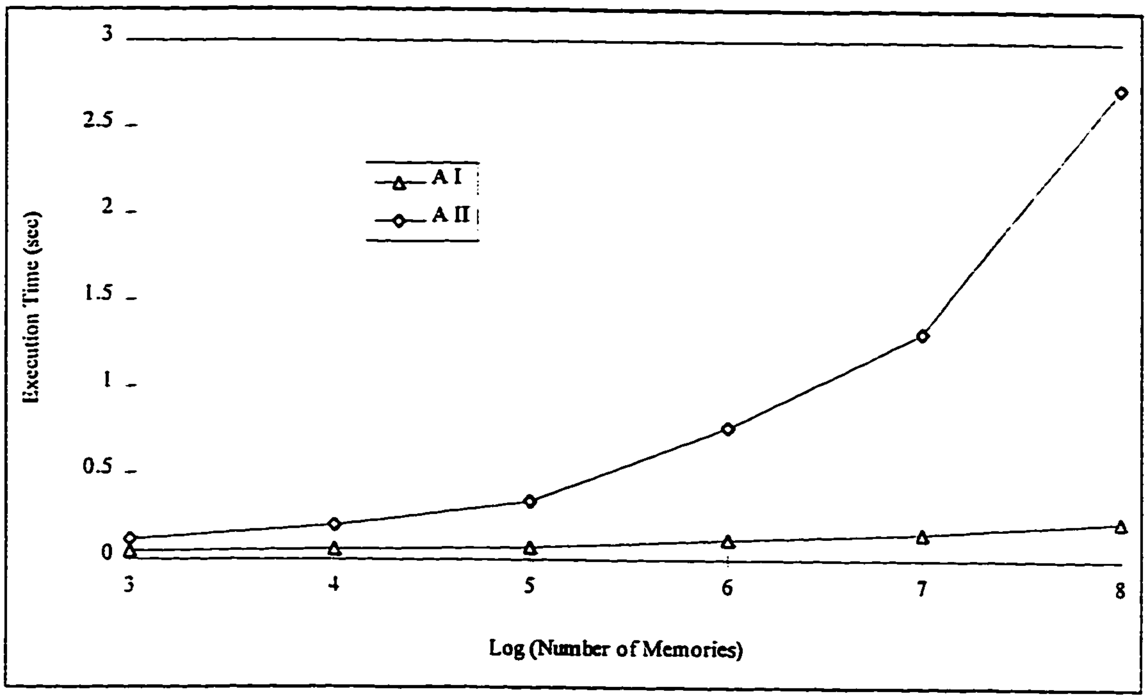


Figure 5-16 : Execution Time Vs. m of Pattern Access

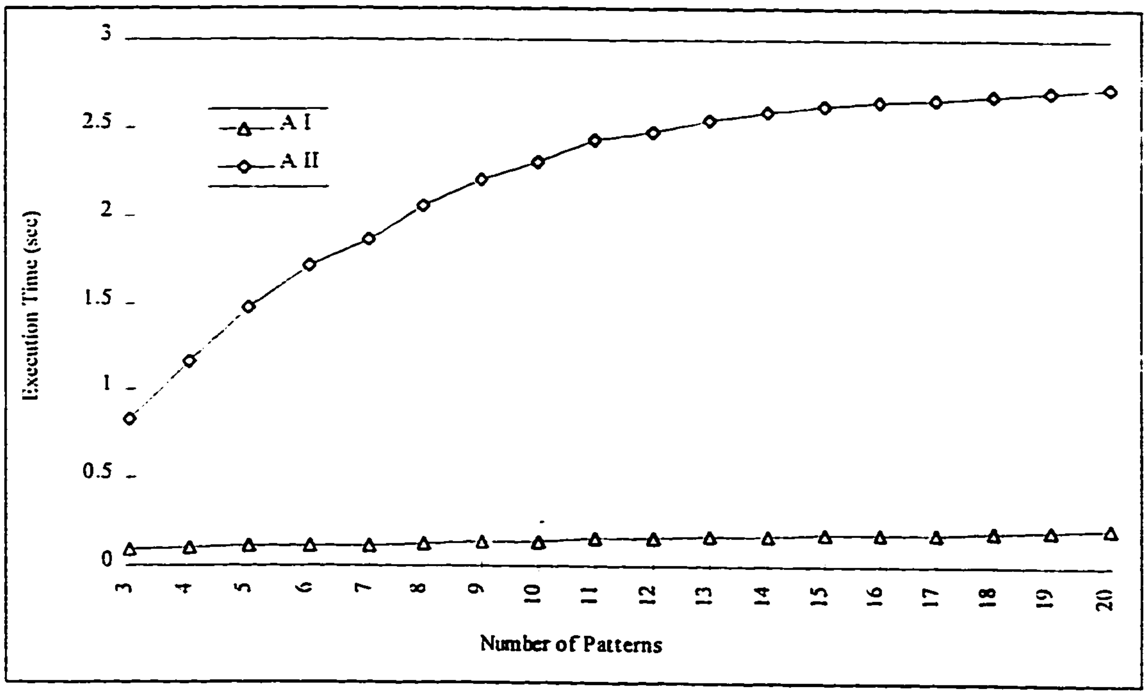


Figure 5-17 : Execution Time Vs. P of Pattern Access

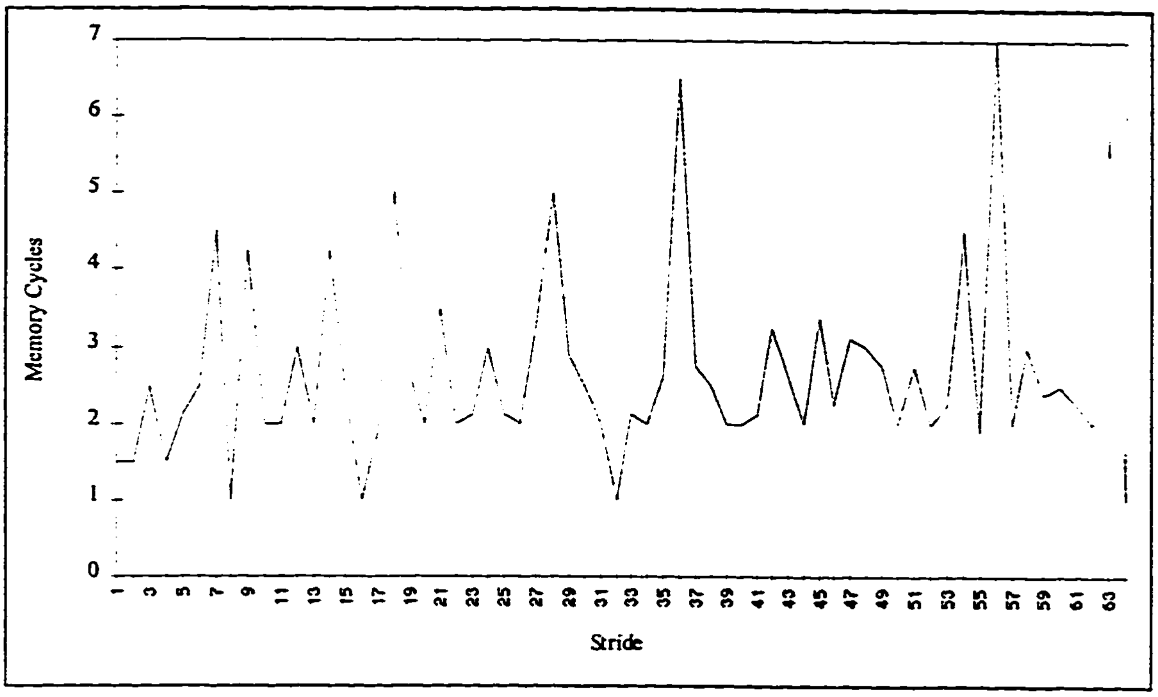


Figure 5-18 : Stride Access of AI for m=3

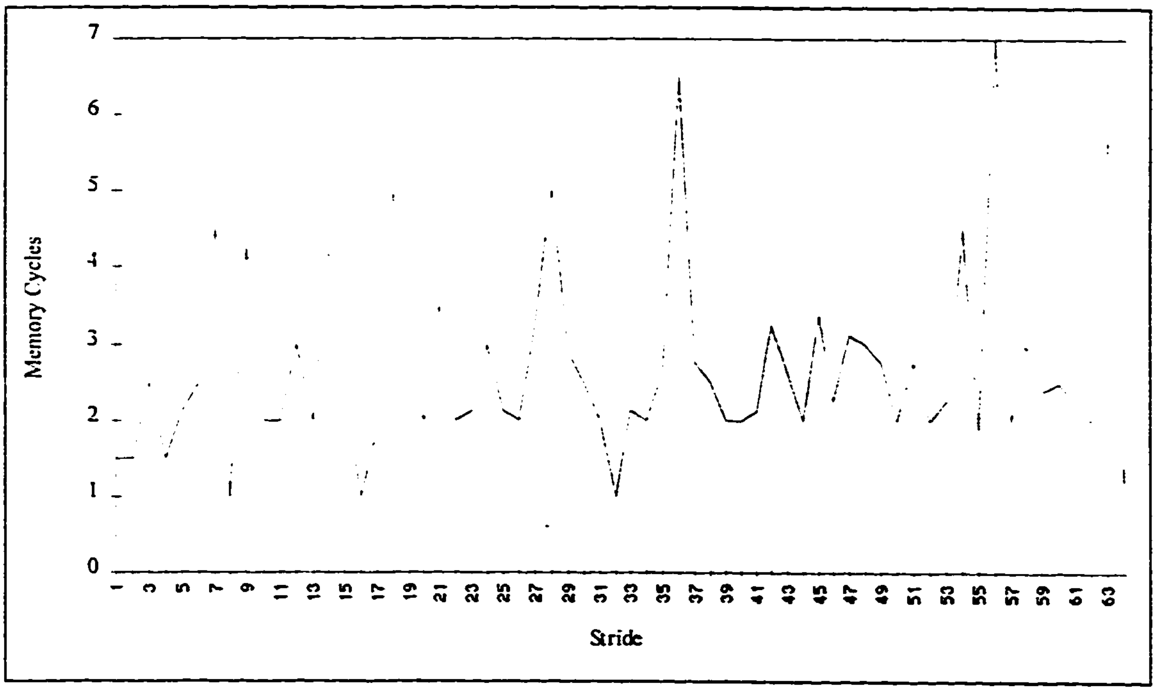


Figure 5-19 : Stride Access of AII for m=3

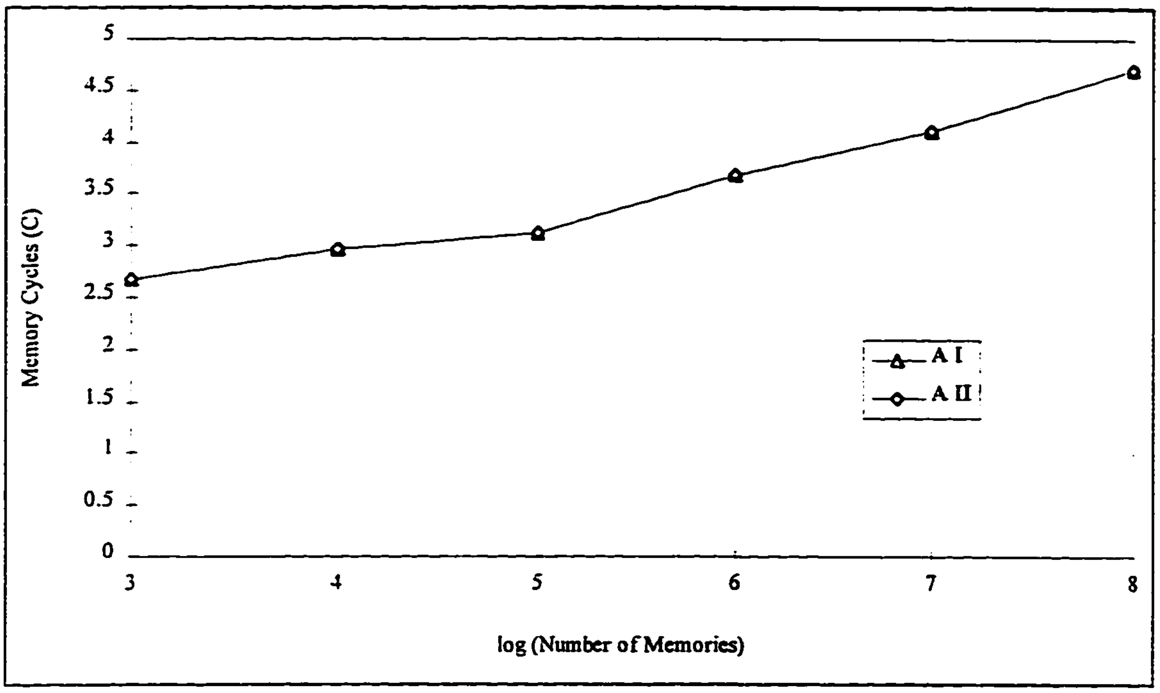


Figure 5-20 : Average Stride Access of NN

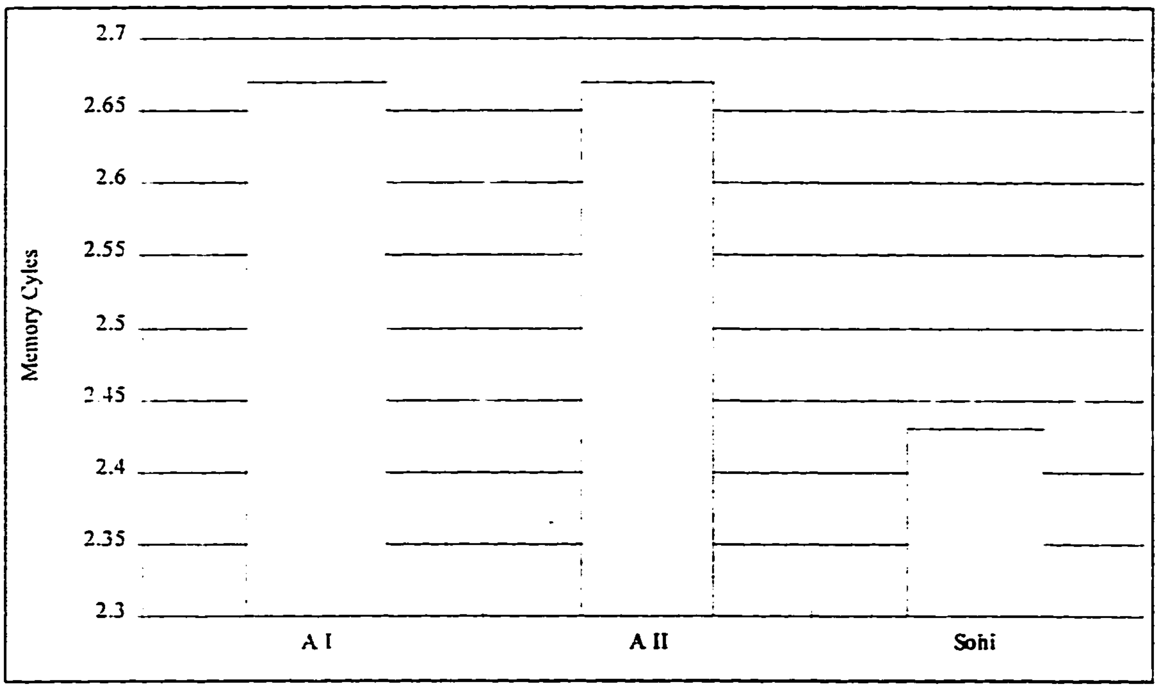


Figure 5-21 : Average Stride Access for m=3

	m					
	3	4	5	6	7	8
Worst	7.00	8.25	11.66	18.50	20.08	27.05
Avg	2.67	2.96	3.13	3.68	4.11	4.71
Var	1.46	2.53	4.74	9.28	5.09	20.20

Table 5-3 : Stride Access of AI

	m					
	3	4	5	6	7	8
Worst	7.00	8.25	11.66	18.50	20.08	27.05
Avg	2.67	2.96	3.13	3.68	4.11	4.71
Var	1.46	2.53	4.74	9.28	5.09	20.20

Table 5-4 : Stride Access of AII

Chapter 6

6. Genetic Algorithms

For the last decade, general optimization algorithms like simulated annealing, evolutionary algorithms, and genetic algorithms have been very widely and successfully used in diverse fields. Recently, a lot of attention and research efforts have been devoted to explore the theory and applications of genetic algorithms. This made genetic algorithms very popular in many fields like VLSI design automation, machine learning, and simulation of biological genetic systems.

In our problem, we have used genetic algorithms to synthesize storage matrices. The results were encouraging. Solutions found by GA were superior to all other approaches. However, the execution time of GA was the largest among the other approaches in the case of synthesizing matrices for storage of arbitrary stride patterns. In cases where storage synthesis is needed once at compilation time, GA are very attractive, since execution time is not a main issue.

6.1 Background

Simulated annealing, evolutionary algorithms, and genetic algorithms are directed random search strategies used in solving various types of optimization problems. These strategies are useful for problems with huge solution spaces. They mainly operate by testing sample solutions from diverse areas of the search space and then gradually intensifying the search in promising areas. This way, the search algorithm concentrates the effort in areas more likely to have the optimal solution instead of wasting time in searching hopeless areas. To achieve this, these algorithms have two main factors in common. First is the random search technique which is used to explore different areas of the solution space. Second is the technique used to direct the random search to promising areas. The second factor narrows the search scope as the algorithm progresses. Simulated annealing uses random perturbation to current solutions as the search technique and a cooling schedule as the technique to direct the search and narrow the search scope. Evolutionary and genetic algorithms are common in their natural origin from which they were inspired. Evolutionary algorithms use mutation as their main search strategy. On the other hand, genetic algorithms use the crossover operator as the main search technique and mutation as a secondary operator. Genetic algorithms use probabilistic selection as a way of directing the search process.

In nature, fit individuals survive and produce next generations while weak individuals die out. This way, good genes of fit individuals survive in next generations and generational evolution takes place. As new generations are produced, the average fitness of individuals is expected to improve. Features of individuals are carried by their genes. Two things are responsible for genetic evolution in nature : selection and reproduction. Giving survival to the fittest is actually selecting it and its genes to survive in next generations. Genetic materials of individuals are contained in what is called chromosomes. Reproduction happens when genetic materials of two parents recombine to form the genetic materials of new offsprings in the new generation. This process is called crossover. In crossover, segments of parents' chromosomes recombine and form the offsprings' chromosomes. Since crossover happens after selection takes place, only those fit individuals participate in producing new individuals for the new generation. This makes the expected fitness of the new generation high to drive the evolution wheel.

Genetic algorithms simulate the genetic evolution in nature. They treat solutions of a given problem as individuals of a population. The population in a genetic algorithm is all currently produced solutions. The fitness of a given solution is the objective function that we want to optimize in our problem. There should be an encoding scheme to encode solutions as chromosomes, in order to apply genetic operators like crossover and mutation. Typically, a chromosome is a string of integers representing a solution. The genetic algorithm starts by randomly generating an initial population. It then evaluates the fitness of all solutions in this population. After this comes the probabilistic selection stage in which a solution is selected with a probability proportional to its fitness value. Finally, solutions of the new generation are produced as offsprings of the selected parent solutions from the current population. Production of offsprings from parents is done by the two operators: crossover and mutation. Like in nature, in crossover, randomly selected segments of parents' chromosomes are intermixed to form chromosomes of offsprings. In mutation, a randomly selected gene of the offsprings chromosome is inverted. The cycle of producing a new generation from an existing one continues until some stopping criteria are met.

Any genetic algorithm is composed of the following main building blocks :

- Encoding Scheme.
- Fitness Function.
- Initial Population.
- Selection Mechanism.
- Crossover.
- Mutation.
- Control Parameters and stopping criteria.

A genetic algorithm is almost completely defined by defining the above building blocks. Although there are standard forms and settings of most of these items, many implementation details and variations differ from one genetic algorithm to another. The first two items are directly related to the problem being solved, while the other items are quite independent of the problem. So, for different problems, the first two items should be totally different while the others should not. In the following sections, each of these items will be described for the case of our problem.

6.2 Encoding Scheme

In many fields, finding a way of representing a solution as a chromosome is the main issue. Before using the genetic algorithms approach, there should be an encoding scheme. The encoding scheme is a way of representing a solution of the problem we have as a chromosome. Being successful in setting this encoding scheme is essential to the success of the genetic algorithm in finding a solution to the problem. In a good encoding scheme, the features of a solution are clearly manifested by the solutions genes. If we can attribute the high fitness of a given solution to a subset of its genes with a high probability, then we have a good encoding scheme. Typically, a solution is encoded as a string of integers. This string is the chromosome and the integers it contains are the solutions genes.

In our problem, a solution is simply a Boolean matrix. If in the problem there are 2^m memories and n basis vectors, then the storage matrix will be an $m \times n$ Boolean matrix. We want to encode or map this matrix into a string or an array of integers. Let us consider every vector or column in the matrix as the binary representation of an integer number, where the highest row in the matrix contains the most significant bits. This way, we can represent the matrix as an array of integers. These integers range from 0 to $2^m - 1$. In this encoding scheme, an $m \times n$ matrix is represented as a chromosome of n m-bit integers corresponding to genes.

Figure 6-1 shows how a solution (Boolean matrix) is encoded into a chromosome.

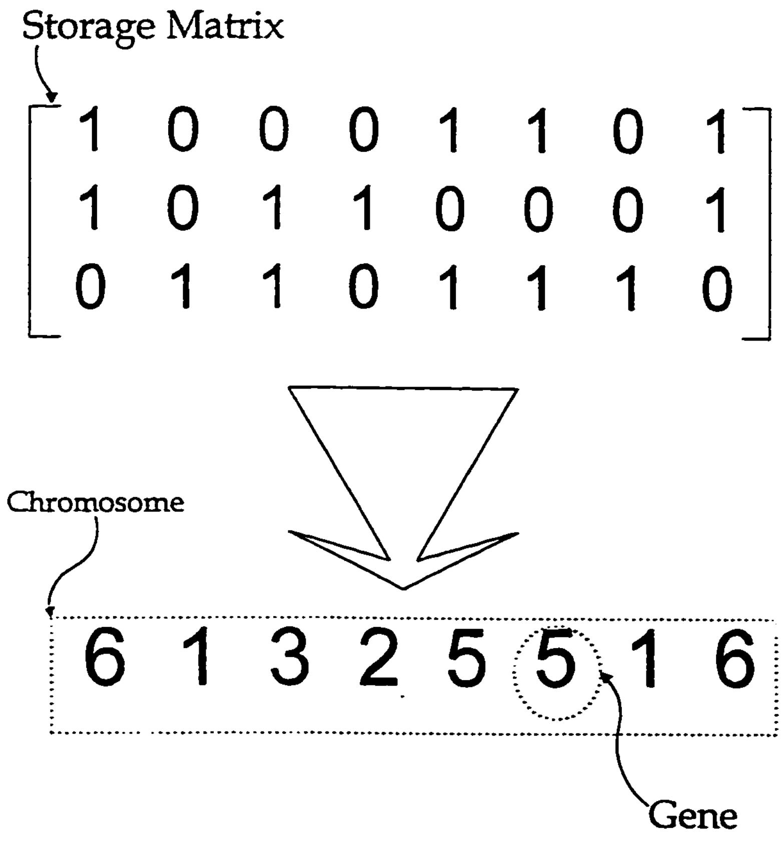


Figure 6-1 : Encoding Scheme

We know that a matrix is nonsingular if and only if its columns (vectors) are linearly independent. Now, in a given matrix, we can identify a subset of linearly independent vectors and the remaining vectors are linear combinations of these. So, in a solution with high fitness (a solution containing high- or full-rank restricted matrices), there is a high probability that those genes corresponding to linearly independent vectors are responsible for this high fitness. In other words, this encoding scheme is a good one in the sense that good features of solutions are clearly manifested by their genes. Another desirable feature of this encoding scheme is that it is simple and straight forward.

6.3 Fitness Function

The fitness function is a measure of how good a solution is. This function should be an accurate measure of our optimization objective so that good or fit solutions are selected to survive in next generations. After producing a new generation, the fitness function is invoked for every solution to compute its fitness in preparation for the selection step.

In the case of power-of-2 patterns, the cost function we used was the percent deviation from the lower bound of 1. This way, an optimal solution will have a cost of 1, while the worst solution will have a cost of 2^m in the case of 2^m memories. However, for the fitness function, we want the fitness value to be proportional to how good a solution is. In other words, it should be high for good solutions and low for bad solutions. So, we will choose the fitness function to be $\frac{1}{C}$, where C is the cost function used before. This fitness function will range from $\frac{1}{2^m}$ to 1, where 1 is at the fit end. The same thing applies for the case of stride access where C is the number of memory cycles.

6.4 Initial Population

The first step in the genetic algorithm is the generation of the initial population. The initial population is the first generation from which next generations will be formed by selection, crossover, and mutation. In our case, this population was randomly generated. Every gene is selected as a random integer between *0* and $2^m - 1$. It is essential to generate the initial population so that all possible genes are in the populations chromosomes. It also should be such that chromosomes in the population are of various and diverse combinations of genes. These two criteria depend to some extent on the random number generator used. It is also important to have a population size enough to cover all possible genes and diverse chromosome combinations of these genes.

6.5 Selection

After computing the fitness values of all solutions in the current population, the selection stage is invoked. "Survival for the fittest" is a main rule in nature and a corner stone in genetic algorithms. This rule ensures that good solutions will survive while weak solutions will extinct as new generations are produced. Survival of good solutions implies the survival of their good genes and hence survival of their good features. Improvement over generations is driven by two things: selection and crossover. Selection keeps good genes and features for new generations. In crossover, genes of good solutions are recombined hoping that an optimal solution is a combination of features or genes of good solutions. By having large number of offsprings in the new population taken from a good solution, we increase the probability of getting the right combination of an optimal solution in one of these offsprings. So, our selection mechanism should be chosen so that the number of offsprings taken from a given parent solution is proportional to its fitness value.

In our problem, we use the Roulette wheel selection method [12,42]. In this method, every solution is allocated a pie slice of a roulette. The area of this pie slice is proportional to its fitness. Then the roulette is spin so that it will rotate with a random angle between 0 and 2π. When the roulette stops, it will be pointing to one point belonging to one of the solutions' areas. Every time the roulette is spin; the solution corresponding to the pointed area is selected. Clearly, the probability that the roulette will point to a solutions area is proportional to that area and hence is proportional to its fitness value. So, this method guarantees that selecting a solution for reproduction is proportional to its fitness. This way, a fitter solution will be allocated a higher number of offsprings in the new generation and will have a higher probability of surviving in next generations. The roulette wheel selection method is demonstrated in Figure 6-2. For large populations, the expected number of offsprings of a given parent solution i will be $\frac{f_i}{f_t} \cdot P_{size}$, where f_i is the fitness of solution i, f_t is the total fitness of all solutions in the current population, and P_{size} is the population size.

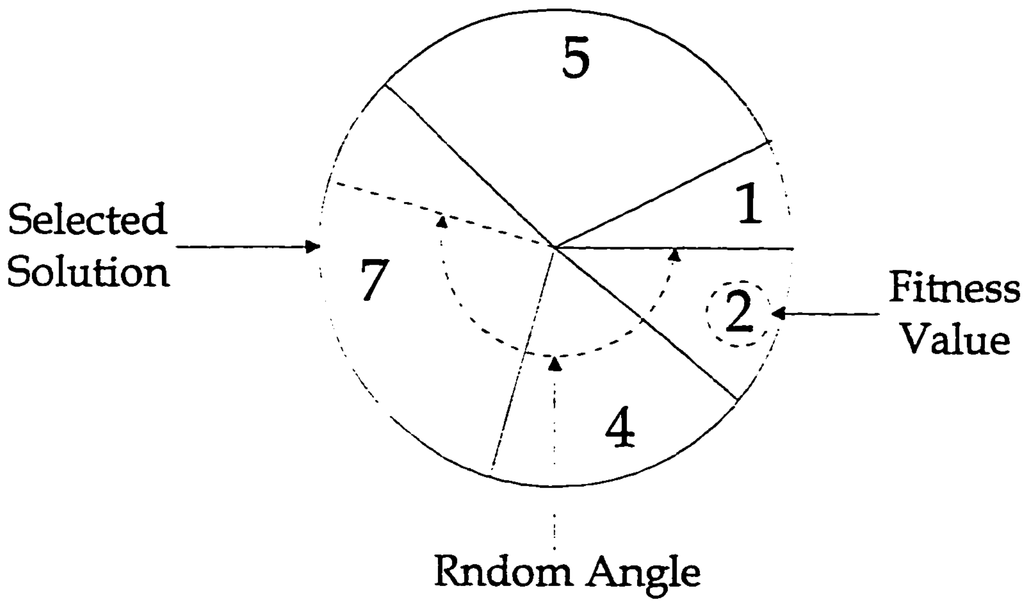


Figure 6-2 : Roulette Wheel Method

6.6 Crossover

Crossover is the main search operator in genetic algorithms. It has similar operator in nature. In nature, genes of parents are recombined to form genes of offsprings. In crossover, randomly selected substrings of parents' chromosomes are interchanged to form chromosomes of offsprings.

A good solution will have a large number of offsprings as we have seen in the previous section and so a large number of substrings will be selected. This large number will increase the chance of taking the substring most responsible for the high fitness of the solution. When this substring is recombined with a similar substring of another good solution, it will likely lead to an optimal or suboptimal solution [12].

There are different ways of dividing a chromosome into substrings. The simplest form is shown in Figure 6-3, where a random point is selected between 0 and *L-1* (*L* is the length of the chromosome). This will divide the parent $P_1's$ chromosome into two substrings: S_{11} and S_{12}. Another parent P_2 is divided into two substrings: S_{21} and S_{22}. Now, in crossover, two offsprings will be formed: O_1 and O_2. O_1 will be composed of S_{11} and S_{22}. O_2 will be composed of S_{12} and S_{21} as shown in Figure 6-3. Another way of subdividing the chromosome is by taking two cut points instead of one as shown in Figure 6-4. In our experiments, we have tried both ways. We found that taking two cut points gives better results in the average.

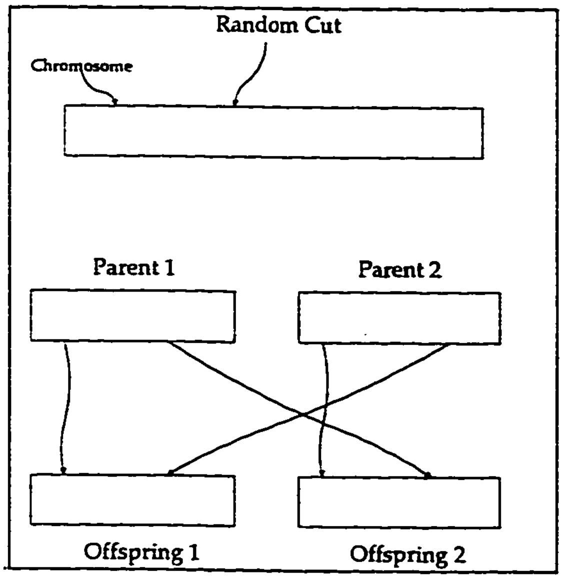


Figure 6-3 : 1-Cut Crossover

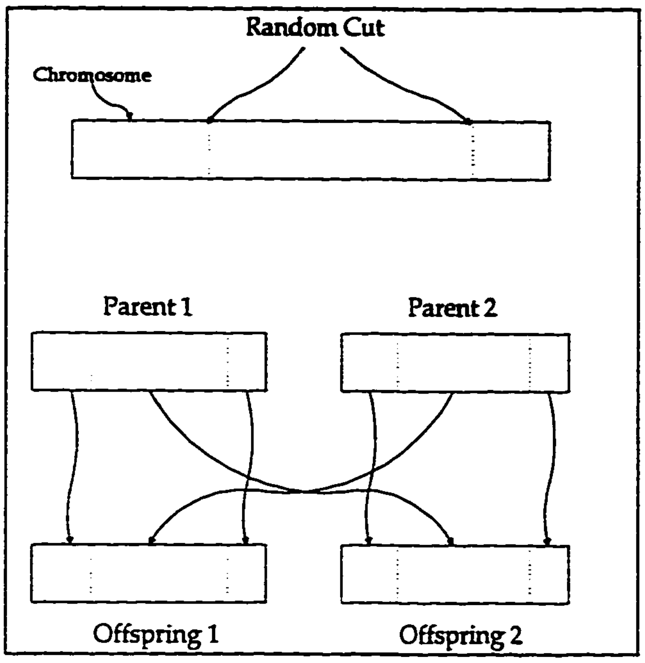


Figure 6-4 : 2-Cut Crossover

When two solutions are selected to be parents for next generation, they are not always subjected to the crossover operator. The crossover operator is applied with a probability P_C. So, there is a probability of $(1 - P_C)$ that the two parents will be taken as they are (without crossover) to be members of the new generation.

6.7 Mutation

Assume that a given gene (a vector in our case) is in the chromosome of only one solution in the population of a given generation. There is a probability of not selecting that solution for the next generation. In this case, that unique gene will be lost for ever. This is because the crossover operator will create new chromosomes by recombining existing ones, but will never create a new gene. So, with the crossover operator alone, a lost gene will be lost for ever which is undesirable [12]. To overcome this problem, the mutation operator was introduced. Again, there are various forms of this operator, the mostly used of which is the inversion. When this mutation is applied on a given chromosome, a randomly selected gene in that chromosome is inverted. This operator is demonstrated in Figure 6-5.

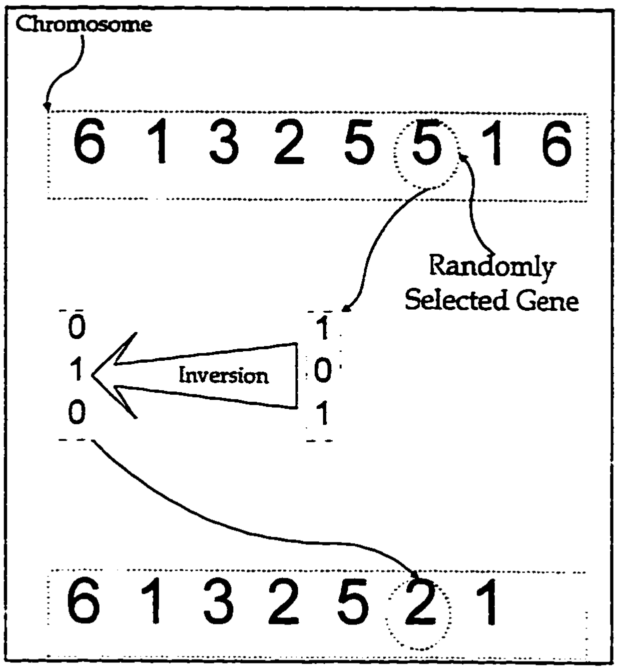


Figure 6-5 : Mutation by Inversion

In our implementation, we tried this form of mutation in addition to another form (Figure 6-6). In the other form, when a gene is randomly selected, it is replaced with a new random gene (a random integer between 0 and $2^m - 1$). Inversion is more suitable when genes are bits. In our case, however, when a gene consists of more than one bit, replacing a gene with another random independent gene (rather than inverting it) adds more diversity in the search process. The mutation operator is also used as a diversification operator. Our experiments showed that the other operator gives better results. This operator is also applied with a probability P_m which is a very small probability. Typically ranging from 0.005 to 0.05

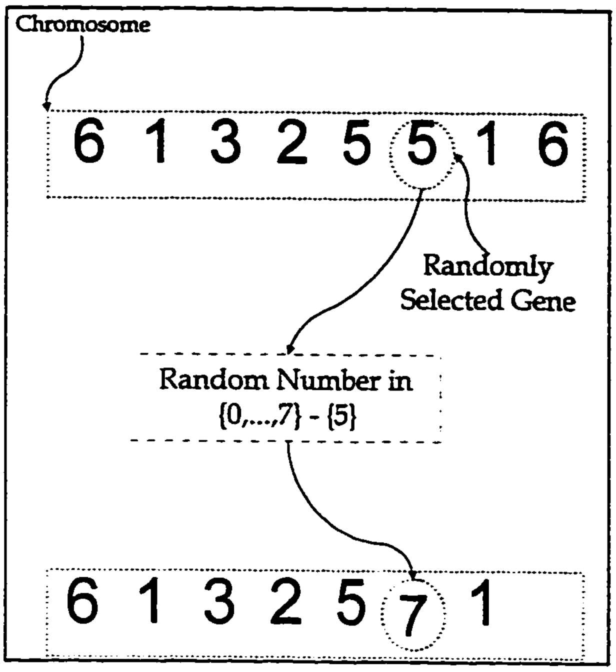


Figure 6-6 : Mutation by Random Number

6.8 Control Parameters

In previous sections, the building blocks of the genetic algorithm were discussed. There are, however, some control parameters that will affect the operation of these blocks and the performance of the whole genetic algorithm. These parameters are P_C, P_m, P_{size}, *MaxGen*, and the stopping criteria.

Typically, P_C ranges from 0.5 to 1.0. We have done experiments with varying P_C in that range. P_C=0.65 was the most suitable setting for this parameter. Our experiments showed that 0.05 was the best value of P_m in the range 0.01 to 0.09. For small problem sizes (e.g. m=4 and P=5), the stopping conditions (finding the optimal solution or not finding further improvement) were met after very few generations, while for large problems (e.g., $m = 8$ and $P = 15$), the stopping criteria were met after a number of generations in the order of 20 on the average. So, we decided to make *MaxGen* a function of m and P. The function chosen was *MaxGen*=$m + P$ because Increasing it beyond this number did not improve the results by significant amounts, but directly increased the execution time. By similar experiments, the best value for P_{size} was found to be equal to $m + P + 10$. As discussed in section 6.4, the population size should be set so that enough randomness and diversity in chromosomes and genes is incurred. Clearly, for large m, there is a larger number of possible genes that should exist some where in the population. Also, large P causes large variations in fitness of different combinations of genes, which needs large population to cover these different combinations. So, we see that P_{size} should also be a function of m and P.

We set the genetic algorithm to stop when one of the following three conditions happens. First, if the optimal solution is found. Second, if the current number of generations exceeds *MaxGen*. Third, if the improvement of the average fitness

($\frac{\bar{f}_{newpopulation}}{\bar{f}_{oldpopulation}}$, where $\bar{f}$ is the average fitness) is not worth continuing (<1.05 for 4 consecutive generations).

6.9 The Algorithm

The genetic algorithm we used is shown in Figure 6-7. The control parameters are set first. Then the algorithm starts by generating the initial population. The algorithm then enters the loop in step 3 which is the main part of the algorithm. In this loop, the fitness of the current population is computed by calling the fitness function for every solution. Next, for every member in the new population, a parent from the current population is selected by the function Select. Now, starting from $i = 0$ and for every i and $i+1$, two offsprings for the new generation are generated by applying the crossover operator on the two selected parents. The function Crossover makes this operator. The Mutation function is then called for every offspring. After generating all members of the new population, the loop is repeated and the fitness of this population is computed. The main loop is repeated until the stopping criteria is met. While producing the different generations the fittest solution is kept and returned at the end as the final solution.

The time complexity of this algorithm in the case of pattern access is $O(G \cdot P_{size} \cdot Pm^2)$, where G is the number of generations. m^2 is the complexity of computing the fitness value of a solution. In the case of stride access, we have to compute the average number of memory cycles C to find the fitness value. We compute C not for all possible origins, but for 4 random origins to reduce the execution time. At the end,

```
1. Initialize Population
2. Old_Avg_Fitness=0;Gen=0;
3. Do
        {
          Old_Avg_Fitness=Avg_Fitness;
          Evaluate_Population;
          for $(i = 0; i < P_{size} - 1; i += 2)$
                {
                  $O1=i$;
                  $O2=i+1$;
                  $P1$=Select ($O1$);
                  $P2$=Select ($O2$);
                  Mutation ($i$);
                  Mutation ($i+1$);
                }
          ++Gen;
        }While (! Stopping Condition)
```

Figure 6-7 : The GA

however, we have to compute C of the final solution for all possible origins. So, the complexity in the case of stride access is $O(G \cdot P_{size} \cdot S \cdot 2^m \cdot mn)$, where 2^m is the number of addresses to be transformed by a solution S is the number of strides, and mn is the complexity of multiplying the $m \times n$ storage matrix by the address vectors.

6.10 Evaluation

The same data used before will be used in this section to evaluate the two Genetic Algorithms (1-cut and 2-cut). The plots in this section are the same as those presented in previous chapters.

Figures 6-8 and 6-9 show how the deviation increases with increasing the number of patterns. We can see how the deviation increases as the number of memories increases. Figure 6-10 shows the difference in performance between the 1-cut and 2-cut GA's. The 2-Cut GA outperforms the 1-Cut GA and behaves more smoothly. The 2-Cut GA is expected to be better because it adds more disruption and variety to population's chromosomes which is required for small populations [12]

In stride access also the 2-Cut is better, not only in the average, but also in the variance and the worst case. Figure 6-16 shows that the two GA versions outperform the Sohi's solution. This is a major result of our work since no body in the literature was able to outperform Sohi's solution. Moreover, our approach is dynamic. In other words, it can be optimized for any number of memories and any set of strides. Sohi's solution, however, is fixed to 8 memories and to arbitrary strides with 12-bit array addresses.

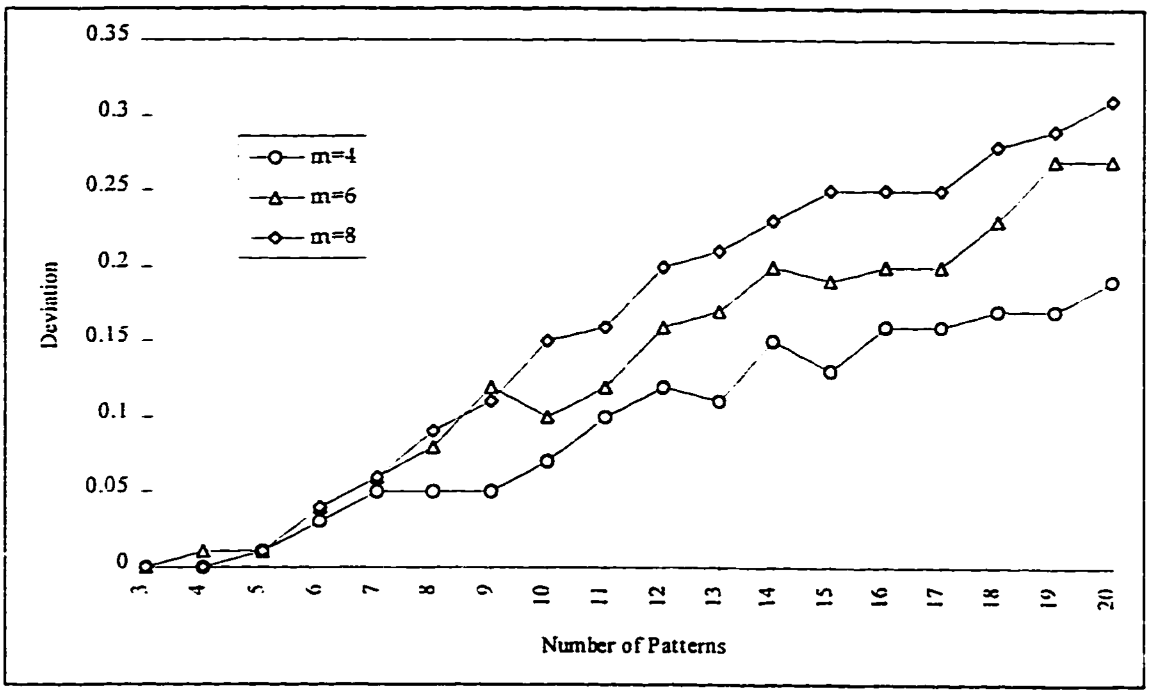


Figure 6-8 : Pattern Access of 2-Cut GA

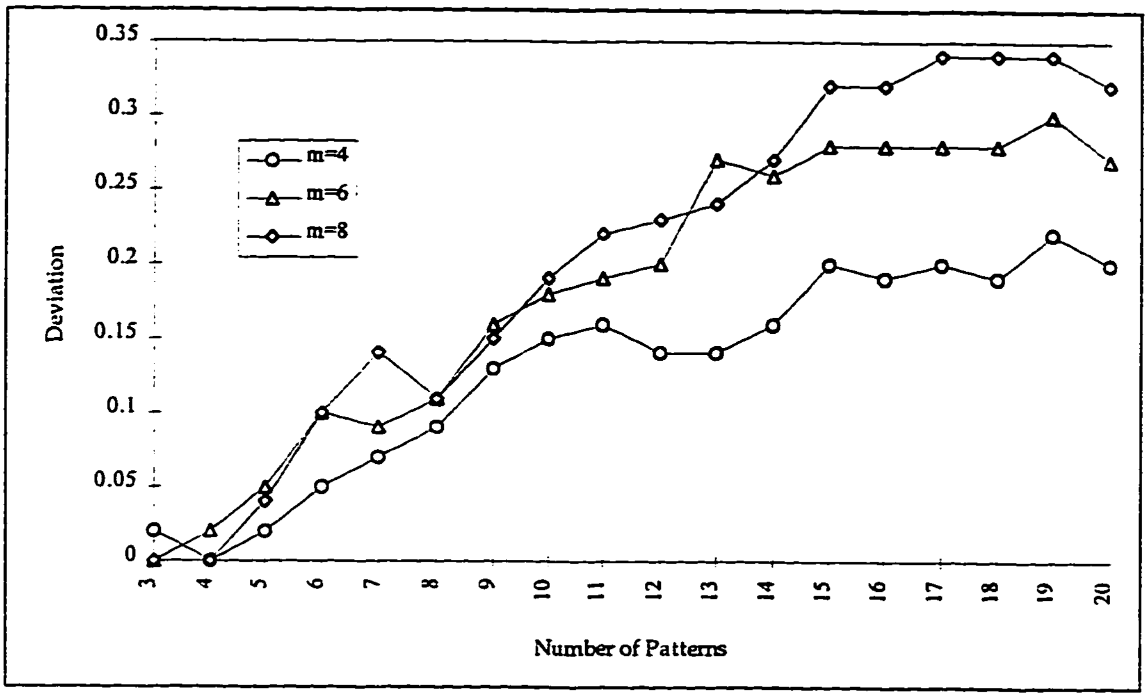


Figure 6-9 : Pattern Access of 1-Cut GA

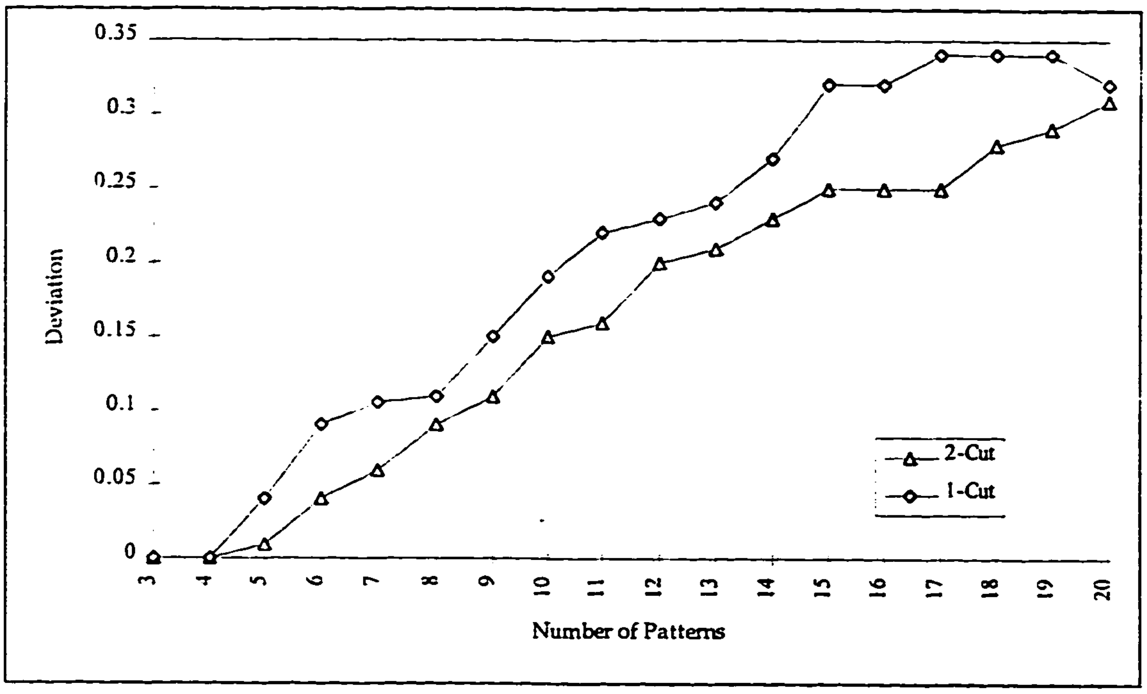


Figure 6-10 : Pattern Access for m=8

		m					
		3	**4**	**5**	**6**	**7**	**8**
P	**3**	0.00	0.00	0.00	0.00	0.00	0.00
	8	0.16	0.16	0.23	0.24	0.25	0.24
	15	0.21	0.25	0.25	0.37	0.36	0.34
	20	0.30	0.25	0.33	0.34	0.48	0.41

Table 6-1 : Worst Case Pattern Access of 2-Cut GA

		m					
		3	**4**	**5**	**6**	**7**	**8**
P	**3**	0.00	0.17	0.13	0.00	0.00	0.00
	8	0.20	0.28	0.27	0.23	0.28	0.29
	15	0.29	0.28	0.33	0.45	0.37	0.41
	20	0.28	0.35	0.33	0.47	0.50	0.44

Table 6-2 : Worst Case Pattern Access of 1-Cut GA

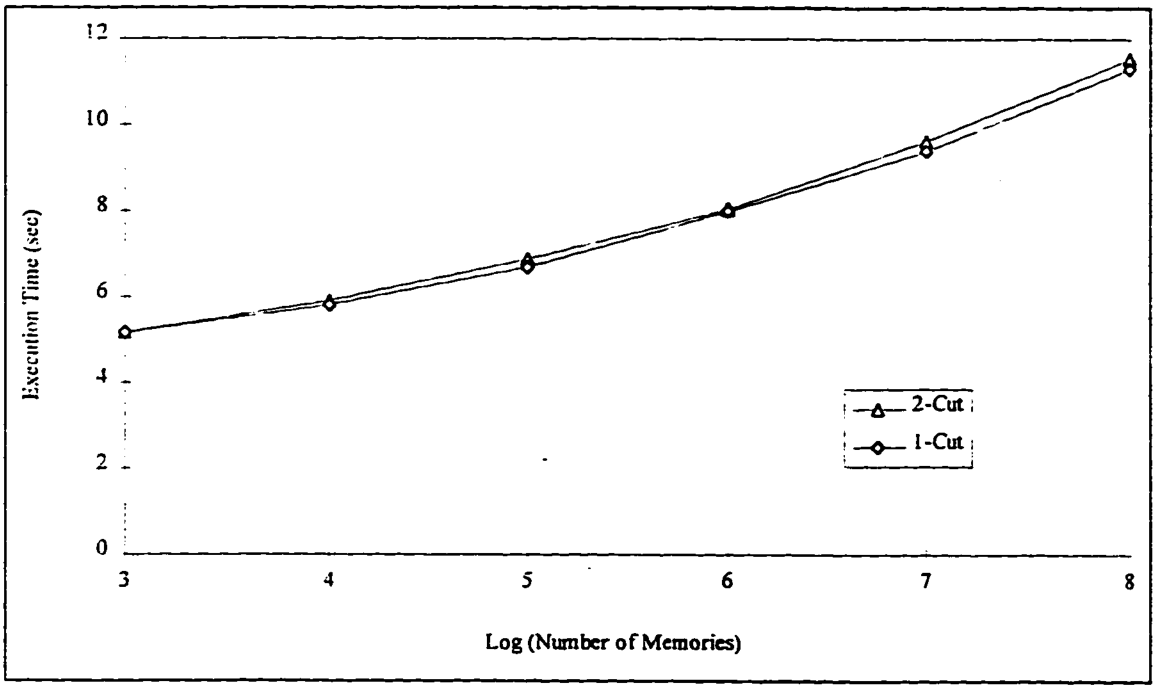


Figure 6-11 : Execution Time Vs. P of GA

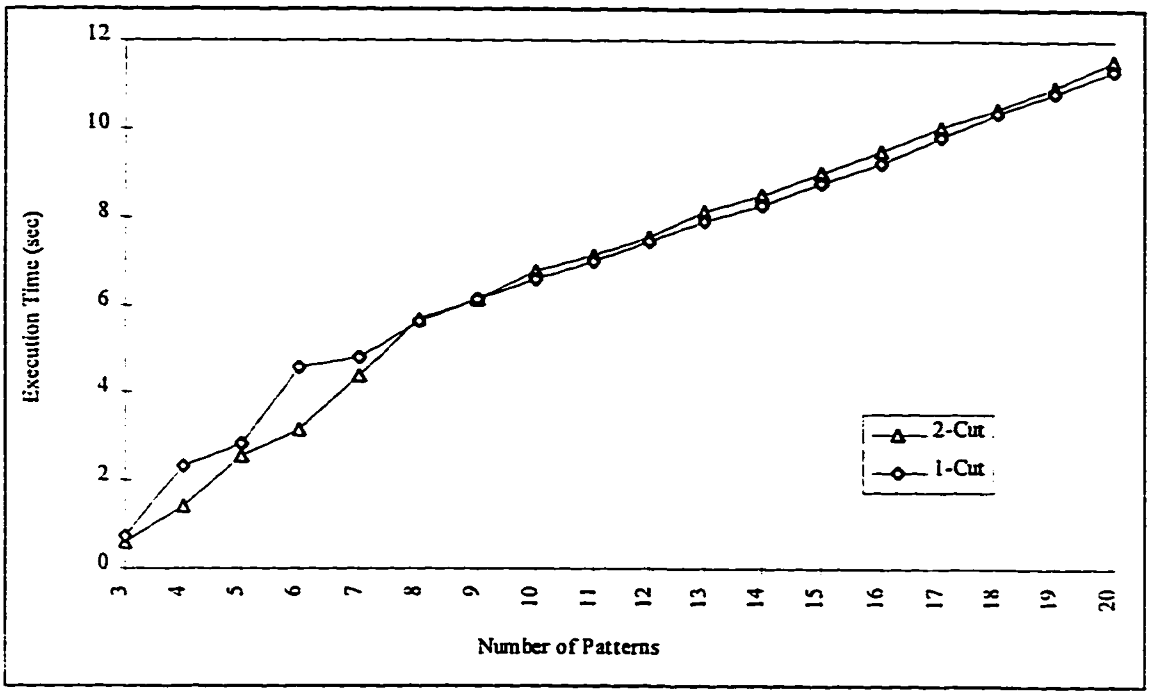


Figure 6-12 : Execution Time Vs. m of GA

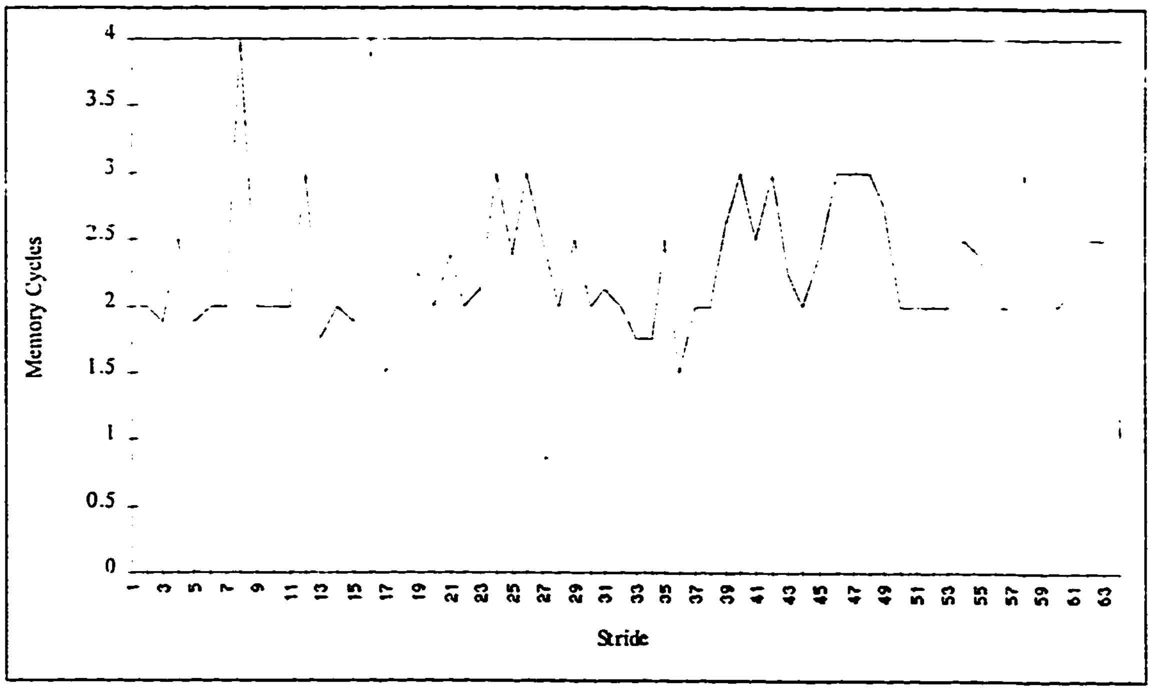


Figure 6-13 : Stride Access of 2-Cut GA

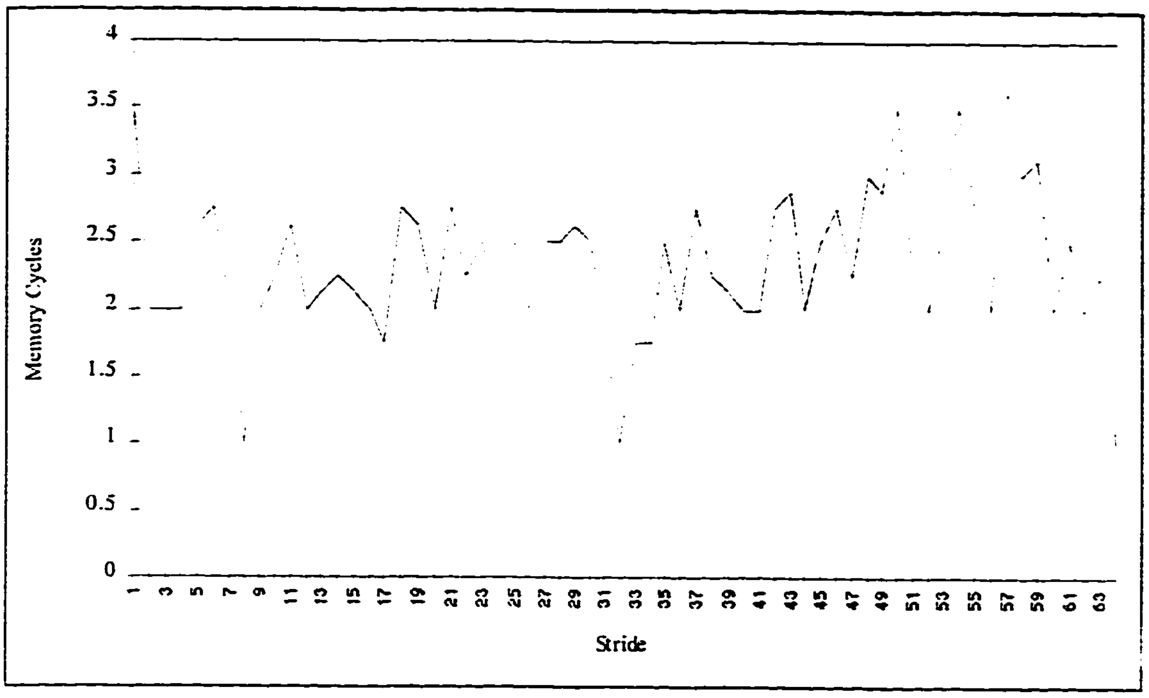


Figure 6-14 : Stride Access of 1-Cut GA

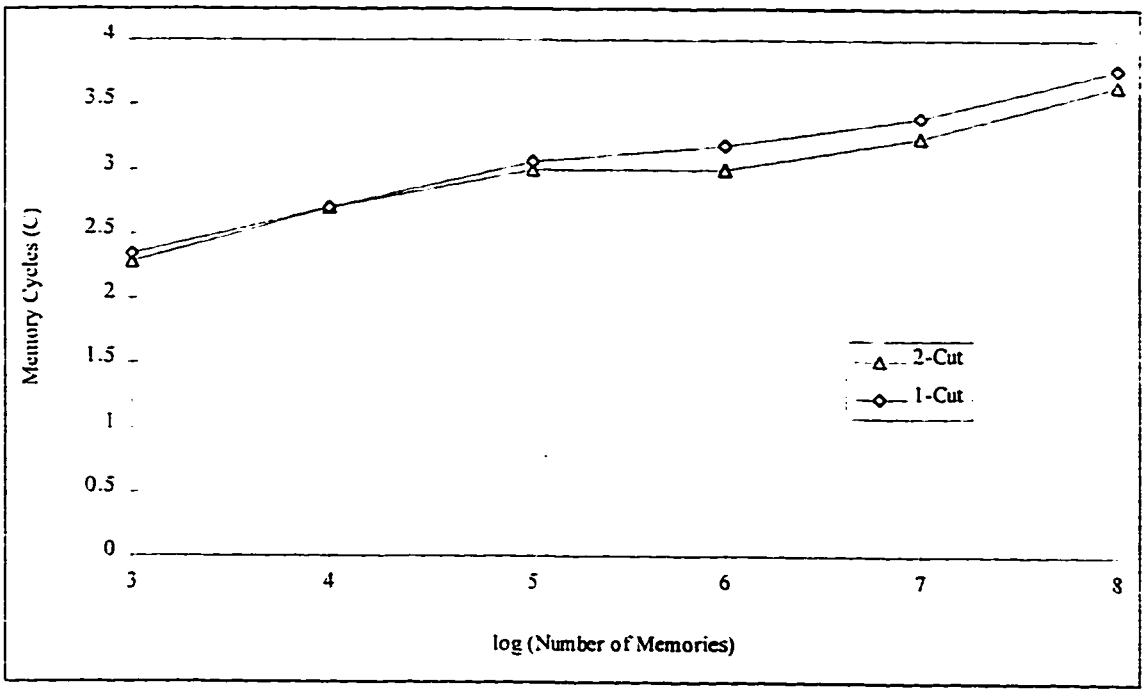


Figure 6-15 : Average Stride Access of GA

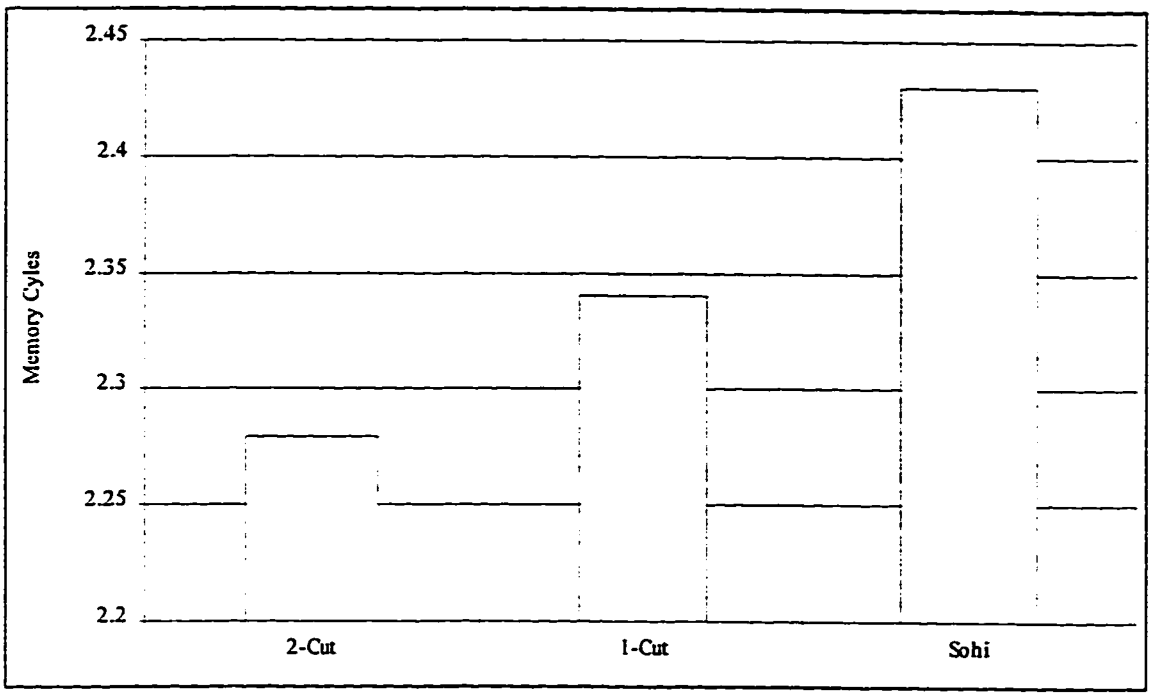


Figure 6-16 : Average Stride Access for m=3

	m					
	3	4	5	6	7	8
Worst	4.00	5.00	4.25	5.62	4.77	5.38
Avg	2.28	2.69	2.99	2.99	3.24	3.65
Var	0.28	0.40	0.31	0.52	0.58	0.58

Table 6-3 : Stride Access of 2-Cut GA

	m					
	3	4	5	6	7	8
Worst	3.62	4.31	4.59	5.77	4.69	5.58
Avg	2.34	2.70	3.06	3.18	3.39	3.68
Var	0.29	0.29	0.38	0.57	0.58	0.82

Table 6-4 : Stride Access of 1-Cut GA

Chapter 7

7. Comparisons and Conclusions

7.1 Comparisons

In this section, we present brief comparisons between the different approaches presented in this thesis. The comparisons are based on the plots below. Table 7-1 shows the ranks from 1 to 5 of the different techniques based on the different criteria we have seen before.

The Genetic approach is the best in terms of solution quality for both patterns and strides. The only problem of this approach is the execution time. This problem, however can be ignored since these are compile time procedures. It will not be an overhead to add few seconds to the compilation time. Other faster approaches (like Clustering or NN) can also be used for intermediate or test compilations and use the GA only at the final compilation. In the case of stride access, some times a storage scheme for a range of strides (e.g., strides 1,2,...,64) is generated once and used as a static scheme without the need to rebuild it at every compilation. In such cases, execution time is not an issue at all and the GA approach will be the best choice.

The MS heuristic can be modified to improve its performance. For example, one can use the Merge procedure only at the beginning to convert the graph into a clique and then proceed just as the GC. This way we should find solutions that are at least as good as those found by the GC.

The NN though was not successful in finding solutions as those found by other techniques, is still attractive in the sense that it is a totally different approach. The very little dependence of AI NN on the number of patterns makes it a good choice for problems of large size. As, can be seen from Figure 7-1, the NN is approaching the performance of other techniques for large number of patterns. The speed of the NN is also a major advantage of this approach.

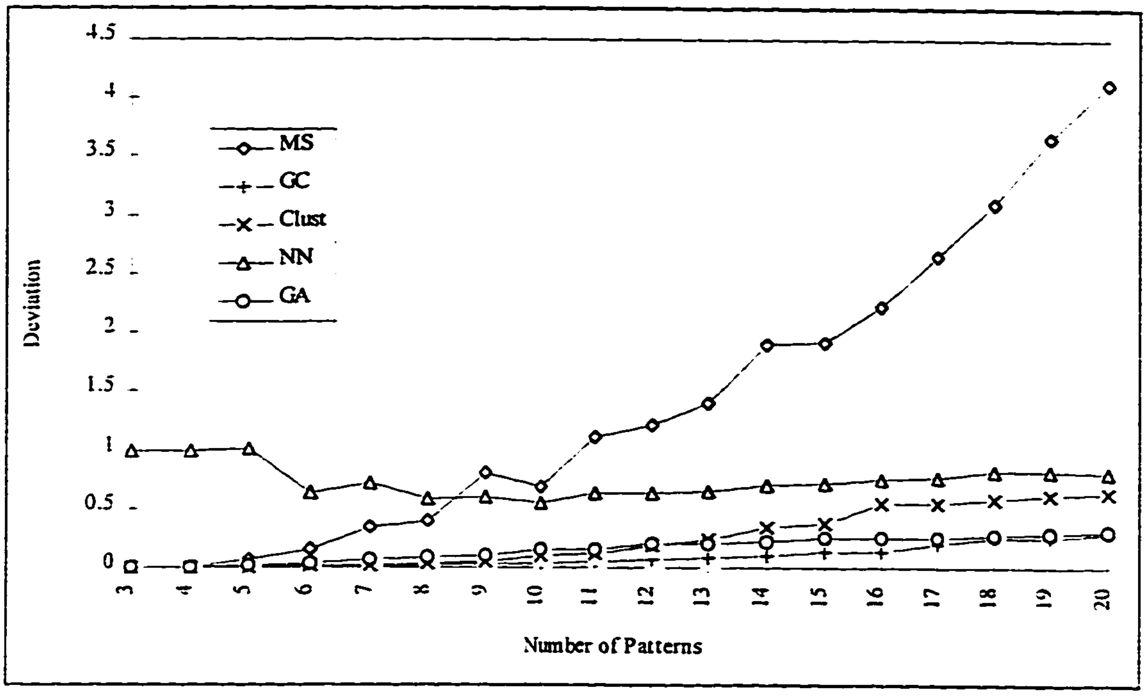


Figure 7-1 : Pattern Access for m=8

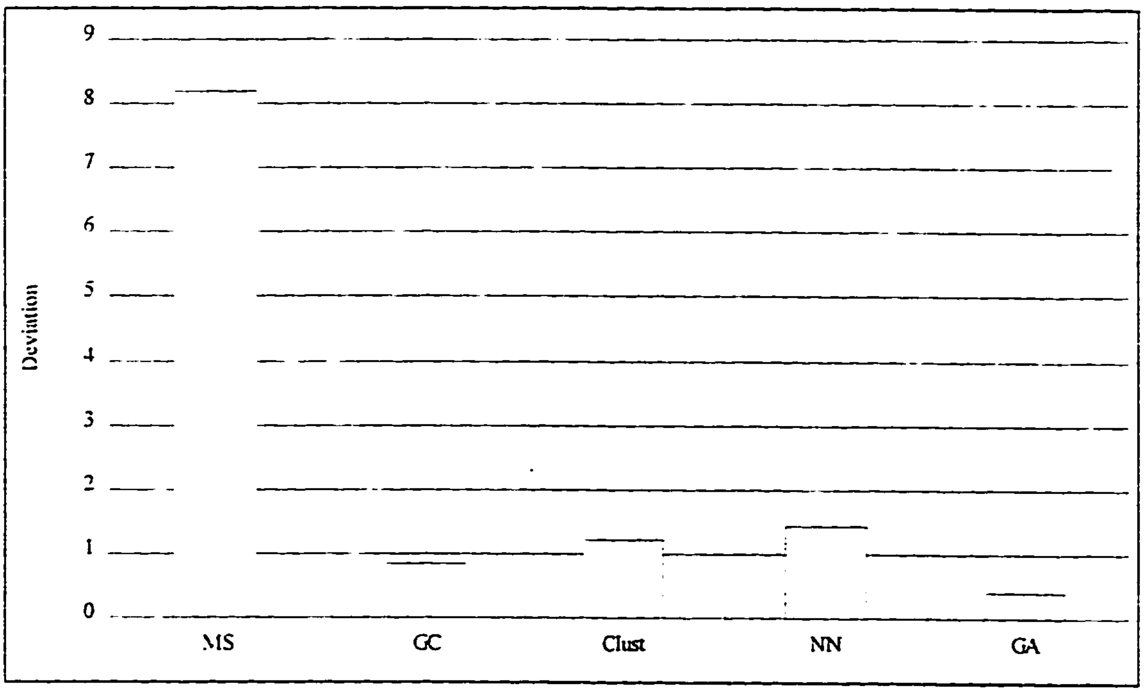


Figure 7-2 : Worst Case in Pattern Access for P=20,m=8

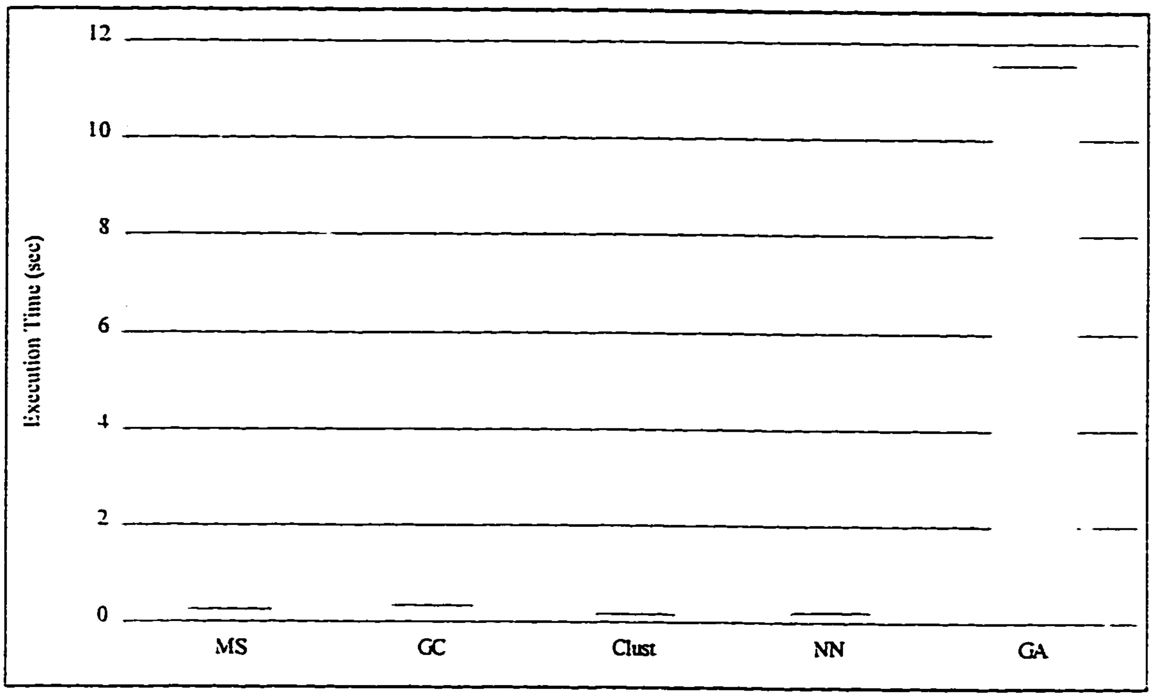


Figure 7-3 : Execution Time in Pattern Access for P=20, m=8

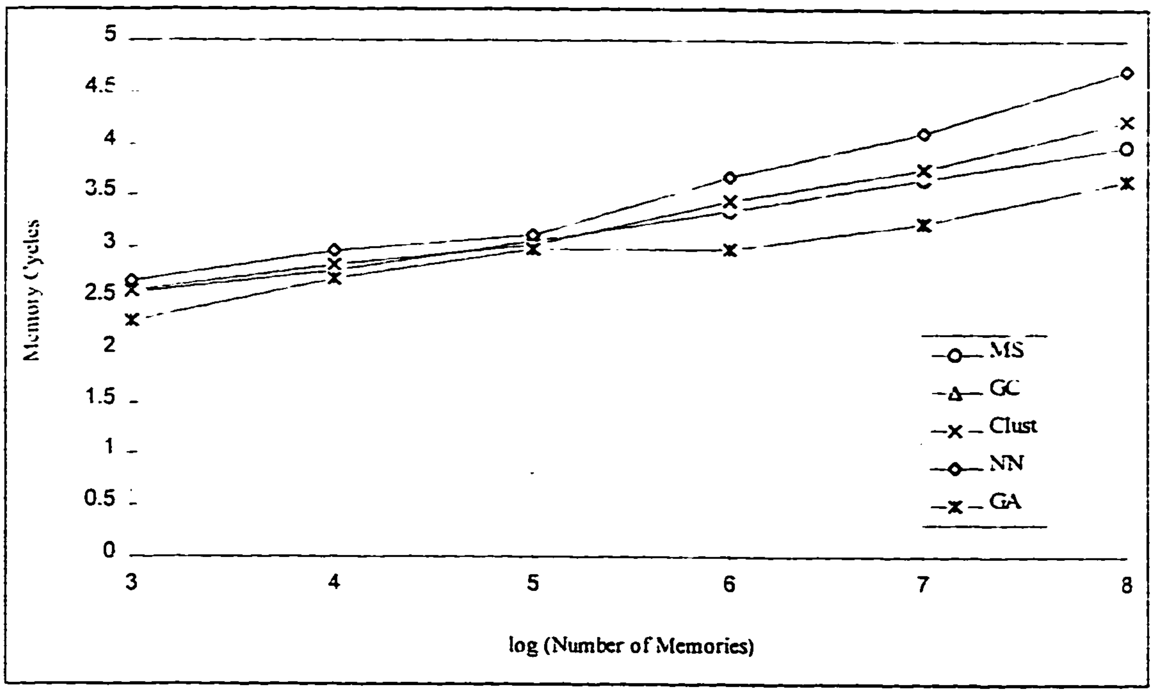


Figure 7-4 : Average Stride Access

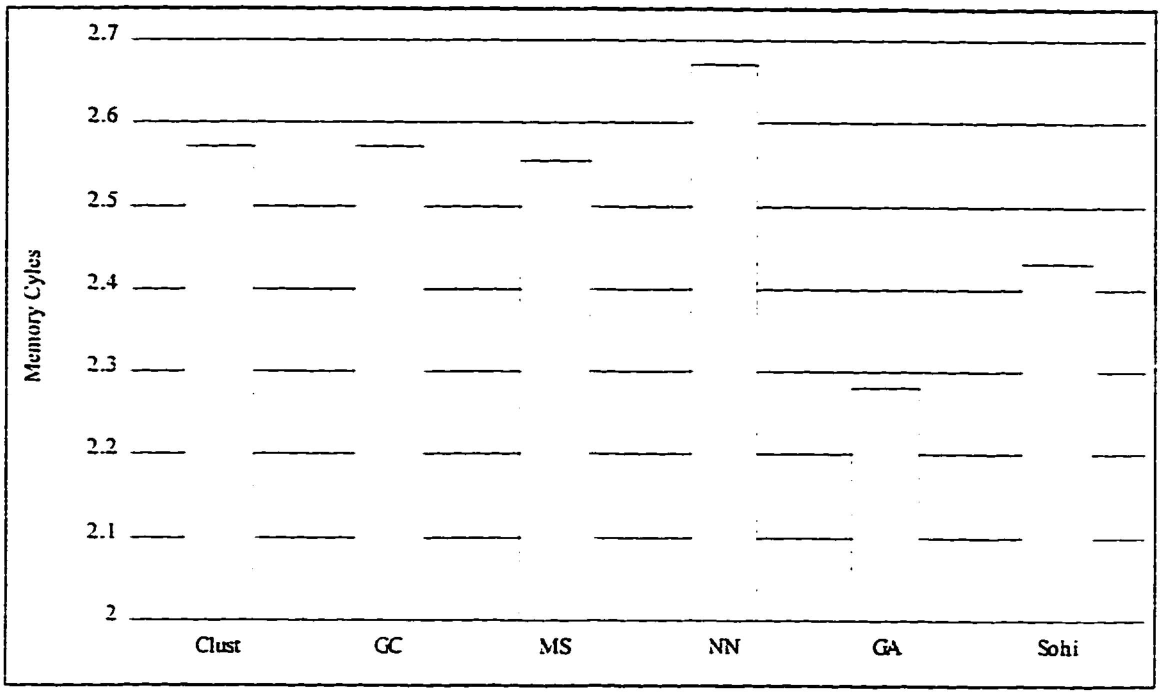


Figure 7-5 : Average Stride Access for m=3

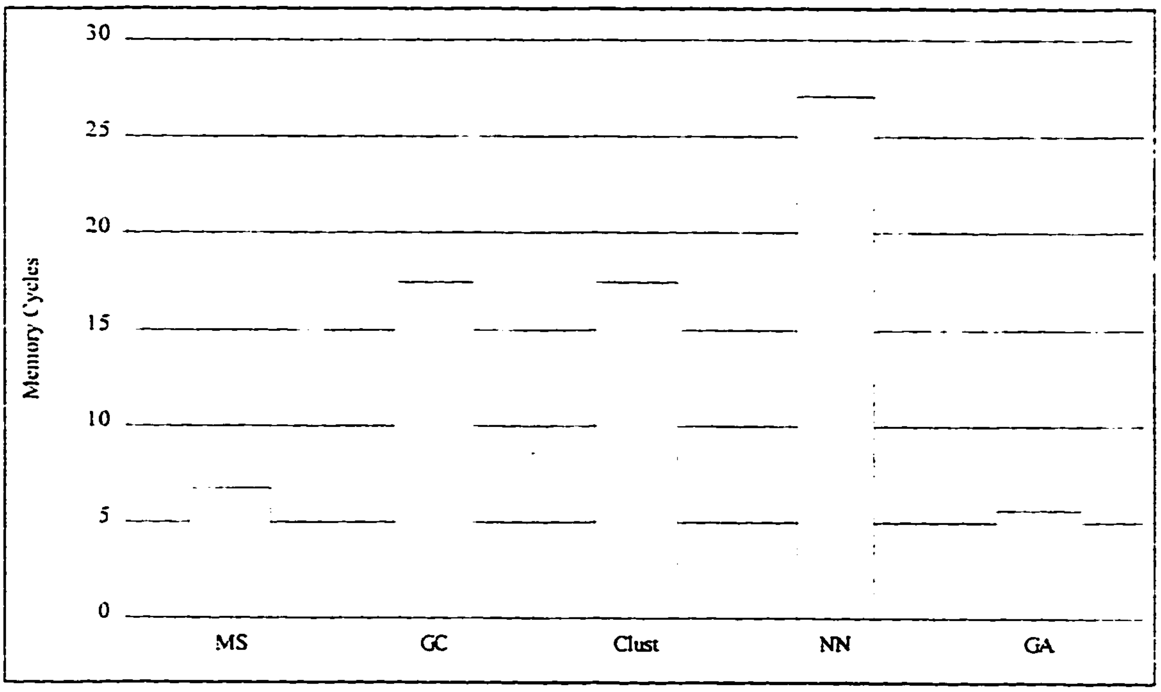


Figure 7-6 : Worst Case in Stride Access for m=8

	Pattern Access		Stride Access		
	Avg. Deviation	Worst Case	Memory Cycles	Worst Case	Execution Time
MS	5	5	2	2	4
GC	2	2	3	3	5
Clust	3	3	3	3	1
NN	4	4	5	5	2
GA	1	1	1	1	5

Table 7-1 : Ranking of the different techniques

7.2 Conclusion

In this thesis, we have addressed the problem of serialization of parallel memory access. In specific, we have addressed the problems of accessing power of 2 patterns as well as accessing arbitrary strides in parallel lock-step memories. We have developed different techniques based on graph coloring, Neural Networks, and Genetic Algorithms.

In Chapter 1, the problem of bandwidth mismatch between fast processors and slow memories was introduced. We have seen the different categories of techniques developed to bridge this gap. Among Technology, Software, and Architecture. Architecture and specifically parallel memory architecture was the most promising technique to speed up memories.

In Chapter 2, we have presented a brief survey of the techniques used to overcome the problem of conflicts in parallel memory access. We have seen that improper storage of arrays into memories can waste up to 56% of the system throughput in accessing arbitrary strides. In the case of other patterns, like power of two patterns, memory throughput can drop by several orders. Three main techniques of storage were presented: Skewing, Prime Memory Systems, and Bitwise Address Transformations. Bitwise Address Transformations are desirable due to their cost, efficiency, simplicity, and other reasons.

In Chapter 3, we have presented a detailed analysis of linear bitwise schemes. The necessary and sufficient conditions for parallel access of power of 2 patterns were stated and proved. The NP-Completeness of the problem of constructing combined storage schemes was also proved.

In Chapter 4, the problem of constructing bitwise storage schemes was reduced to a graph coloring problem. Three heuristics for constructing combined storage schemes were presented and evaluated.

In Chapters 5 and 6, Neural Networks and Genetic Algorithms respectively were used to find storage schemes for both power of 2 patterns and arbitrary strides. The two approaches were tested with in the same way the other heuristics were tested.

In the case of power of 2 patterns, we were able to find optimal solutions in many cases and near optimal in most of the cases. Our solutions are at least 20 times better than those found by Al-Mouhamed and Seiden [1]. In [1] they used simple heuristics to show the feasibility of solutions. In the case of arbitrary strides, our schemes gave better solutions than all schemes in the literature and were more dynamic than some of the best schemes like the one proposed by Sohi [41].

Bibliography

[1] Al-Mouhamed, M. and Steven Seiden, "A Cost-Effective Heuristic Storage For Minimizing Access Time of Arbitrary Data Templates", *Technical Report, ICS-UCI*, 1993.

[2] Al-Mouhamed, M. and Steven Sieden, "Minimization of Memory and Network Contention for accessing Arbitrary Data Templates in SIMD Systems", *IEEE Transactions On Computers*, pp. 757-762, June 1996.

[3] Al-Mouhamed, M. and Steven Sieden, "A Heuristic Storage for Minimizing Access Time of Arbitrary Data PAtterns", *IEEE Transactions On Parallel and Distributed Systems*, Vol. 8, No. 4, April 1997.

[4] Balakrishnan, M., R. Jain, and C. S. Raghavendra, "On Array Storage for Conflict-Free Access for Parallel Memories", *International Conference on Parallel Processing*, pp. 103-107, 1988.

[5] Batcher, K., "The Multidimensional Memory Access in STARAN", *IEEE Transactions On Computers*, C-26..pp. 174-177, Feb 1977.

[6] Boppana, R. V. and C. S. Raghavendra, "Efficient Storage Schemes for Arbitrary Size Matrices in Parallel Processors with Shuffle-Exchange Networks", *International Conference on Parallel Processing*, pp. I:365-368, 1991.

[7] Budnik, P. and David J. Kuck, "The Organization and Use of Parallel Memories", *IEEE Transactions on Computers*, pp. 1566-1569, December 1971.

[8] Charny, C., "Matrix Partitioning on a Virtual Shared Memory Parallel Machine", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 4, pp. 343-355, April 1996.

[9] Cormen, T. H., C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms", *The MIT Press*, 1993.

[10] Deb, A. A., "Conflict-Free Access of Arrays - a Counter Example", *Information Processing Letters*, Vol. 10, p. 20, Feb. 1980.

[11] Deb, A. A., "Multiskewing--A Novel Technique for Optimal Parallel Memory Access", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 6, pp. 595-604, June 1996.

[12] Filho, J. L., P. C. Treleaven, and C. Alippi, "Genetic-Algorithm Programming Environments", *IEEE Computer*, pp. 27-43, June 1994.

[13] Frailong, J. M., W. Jalby, and J. Lenfant, "XOR-Schemes : A Flexible Data Organization in Parallel Memories", *International Conference on Parallel Processing*, pp. 276-283, 1985.

[14] Gao, Q. S., "The Chinese Remainder Theorem and the Prime Memory System", *Proceedings Of the 20th International Symposium on Computer Architecture*, pp. 337-340, 1993.

[15] Goldberg, D. E., “Genetic Algorithms in Search, Optimization, and Machine Learning”, *Addison-Wesley Publishing Company, Inc.*, 1989.

[16] Gupta, R. and M. L Soffa, "Compile-Time Techniques for Improving Scalar Access Performance in Parallel Memories", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, pp. 138-148, April 1991.

[17] Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky, "Computer Organization", *McGraw-Hill International Editions*, 1990.

[18] Harper, D. T. and D. A. Linebarger, "A Dynamic Storage Scheme for Conflict-Free Vector Access", *Proceedings of the International Symposium on Computer Architecture*, pp. 72-77, 1989.

[19] Harper, D. T., "Address Transformations to Increase Memory Performance", *International Conference on Parallel Processing*, pp. I:237-241, 1989.

[20] Harper, D. T., "Increased Memory Performance During Vector Accesses Through the use of Linear Address Transformations", *IEEE Transactions on Computers*, Vol. 41, No. 2, pp. 227-230, Feb. 1992.

[21] Hwang, K. and F. A. Briggs, "Computer Architecture and Parallel Processing", *McGraw-Hill*, 1984.

[22] Jain, A. K. and J. Mao, "Artificial Neural Networks: A Tutorial", *IEEE Computer*, pp. 31-44, March 1996.

[23] Kittel, C., "Introduction to Solid State Physics", *John Wiley & Sons, Inc., New York*, 1986.

[24] Kung, S. Y., "Digital Neural Nrtworks", *Prentice Hall*, 1993.

[25] Lawrie, D. H. and C. R. Vora, "The Prime Memory System for Array Access", *IEEE Transactions on Computers*, Vol. c-31, No. 5, pp. 435-441, May 1982.

[26] Lawrie, D., "Access and Alignment of Data in an Array Processor", *IEEE Transactions on Computers*, Vol. c-24, pp. 1145, Dec. 1975.

[27] Lee, D-L. and Y. H. Wang, "Conflict-Free Access of Arrays In A Parallel Processor", *Proceedings Of the ACM International Symposium On Supercomputing*, pp. 313-317, 1989.

[28] Lee, D-L., "Scrambled Storage for Parallel Memory Systems", *Proceedings Of the 15th International. Symposium On Computer Architecture*, pp. 232-239, 1988.

[29] Lee, D-L., "Solution to an Architectural Problem in Parallel Computing", *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pp. 434-442, 1990.

[30] McHugh, J. A, "Algorithmic Graph Theory", *Prentice Hall*, 1990.

[31] Norton, A. and E. Melton, "A Class of Boolean Linear Transformations for Conflict-Free Power-of-Two Stride Access", *Proceedings Of the International Conference On Parallel Processing*, pp. 247-254, 1987.

[32] Prince, B., "Memory in the Fast Lane", *IEEE Spectrum* , pp. 38-41, Feb. 1994.

[33] Raghavan, R. and J. P. Hayes, "Scalar-Vector Memory Interference in Vector Machines", *International Conference on Parallel Processing*, pp. I:180-187, 1991.

[34] Raghavendra, C. S. and R. Boppana, "On Methods for Fast and Efficient Parallel Memory Access", *International Conference on Parallel Processing*, pp I:76-83, 1990.

[35] Ramanujam, J. and P. Sadyappan, "Optimization by Neural Neyworks", *IEEE International Conference on Neural Networks*, pp. 325-332, July 1988.

[36] Saavedra, R. H. and D. Park, "Improving the Effectiveness of Software Prefetching With Adaptive Execution", *Parallel Architectures and Compilation Techniques*, pp. 68-78, 1996.

[37] Seznec, A. and J. Lenfant, "Interleaved Parallel Schemes", *IEEE Transactions on Parallel and Distributed* Systems, Vol. 5, No. 12, pp. 1329-1334, Dec. 1994.

[38] Seznec, A. and J. Lenfant, "Odd Memory Systems may be quite interesting", *Proceedings Of the 20th International Symposium on Computer Architecture*, pp. 341-350, 1993.

[39] Shapiro, H. D., "Theoretical Limitations on the Efficient Use of Parallel Memories", *IEEE Transactions on Computers*, Vol. c-27, pp. 421-428, May 1978.

[40] Shing, H. and L. Ni, "A Conflict-Free Memory Design for Multiprocessors", *Proceedings Of the ACM International Symposium On Supercomputing*, pp. 46-55, 1991.

[41] Sohi, G. S., "High-Bandwidth Interleaved Memories for Vector Processors--A Simulation Study", *IEEE Transactions on Computers*, Vol. 42, No. 1, pp. 34-44. Jan. 1993.

[42] Srinivas, M. and Patnaik L. M, "Genetic Algorithms : A Survey", *IEEE Computer*, pp. 17-26, June 1994.

[43] Sun, K. T. and H. C. Fu, "A Hybrid Neural Network Model for Solving Optimization Problems", *IEEE Transactions On Computers*, pp. 218-226, Feb. 1993.

[44] Swamy, M. N. and K. Thulasiraman, "Graphs, Networks, and Algorithms", *John Wiley & Sons Inc.*, 1981.

[45] Tanenbaum, A. S., "Modern Operating Systems", *Prentice Hall International Editions*, 1992.

[46] Valero, M., T. Lang, M Peiron, and E. Ayguade, "Conflict-Free Access for Streams in Multimodule Memories", *IEEE Transactions On Computers*, Vol. 44, No. 5, pp. 634-646, May 1995.

[47] Wijshoff, H. A. and J. V. Leeuwen, "The Structure of Periodic Storage Schemes for Parallel Memories", *IEEE Transactions on Computers*, Vol. c-34, No. 6, pp. 501-505, June 1985.

[48] Wilkinson, B., "Computer Architecture Design and Performance", *Prentice Hall 2nd edition*, 1996.

[49] Wong, C., et. al., "Binary-Exchange Algorithms on a Packed Exponential Connections Network", *International Conference on Parallel Processing*, pp. III:176-183, 1995.

Vita

- Husam Saad Abu-Haimed
- Born in 1974, Ranyah, Saudi Arabia
- Received a Bachelor of Science degree in Computer Engineering in 1995 from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.
- Received a Master of Science degree in Computer Engineering in 1997 from King Fahd University of Petroleum and Minerals, Dhahrn, Saudi Arabia.