Design and Analysis of a Gracefully Degrading Interleaved Memory System

KIFUNG C. CHEUNG, MEMBER, IEEE, GURINDAR S. SOHI, MEMBER, IEEE,
KEWAL K. SALUJA, MEMBER, IEEE, AND
DHIRAJ K. PRADHAN, FELLOW, IEEE

Abstract—**A hardware mechanism has been proposed to reconfigure an interleaved memory system. The reconfiguration scheme is such that, at any instant, all fault-free memory banks in the memory system can be utilized in an interleaved manner. The design of the hardware that enables the reconfiguration is discussed. The reconfiguration scheme proposed in this paper is analyzed for a number of distinct benchmark programs. It is shown that memory system performance degrades gracefully as the number of faulty banks increases if the memory system uses the proposed reconfiguration scheme.**

Index Terms—**Fault-tolerant memory, graceful degradation, interleaved memory, memory bandwidth, trace-driven simulation.**

I. Introduction

In A computer system that consists of a processing unit (CPU) connected to a memory system, the rate at which the CPU can process information is limited by the rate at which the information can be transmitted between the CPU and the memory. This is the well-known von Neumann bottleneck. Consequently, a decrease in the bandwidth of a memory system will directly affect the performance of the overall computer system.

There are two main approaches to attain a memory system with a high bandwidth. The first involves the use of a high-speed buffer or cache memory (an excellent survey can be found in [14]) and the second involves the use of several memory *banks* or *modules* connected in an interleaved fashion [5], [6]. Although the use of cache memories has become widespread, their utility is limited by their size. While cache memories are very effective for instructions and scalar data items [14], [15], they have not proven to be effective for numeric processing machines that utilize large data structures (such as arrays). For such systems, in order to achieve a high-bandwidth memory system, one is forced to use interleaved banks of memory. Of course, the best effect is achieved by using a cache memory for instructions and scalar data and an interleaved memory for noncacheable data.

In an interleaved memory system that consists of N independent memory banks (or modules), by associating address latches and data latches with each bank, N different memory accesses can be overlapped. In this C-access method [5], the memory system can accept a stream of memory requests from the processor and service each request, one at a time, thereby increasing the available bandwidth of the memory system to N times the bandwidth of a single bank. A processing system that utilizes a cache memory for instructions and a C-access interleaved memory system for data is shown in Fig. 1.[1]

The bandwidth of interleaved memories has been studied extensively using analytical and simulation techniques [1], [2], [12], [16]. Apart from the referencing behavior of programs, the main factor that influences the bandwidth of interleaved memory banks is the manner in which the addresses are distributed among the banks, i.e., the memory organization [12]. Generally the number of banks N that are used to build an interleaved memory is a power of 2, i.e., $N = 2^q$ where q is an integer. In such a system, q bits of the address suffice to select a bank and the remaining bits are used to select a word within a bank. If the q bits are the high-order bits of the address space, the scheme is a *high-order interleaving* scheme whereas a *low-order interleaving* scheme results if the low-order q bits are used to select the bank.

We should mention that an interleaving scheme is not restricted to using only a power of 2 number of banks. Interleaving schemes that utilize a prime number of memory banks have been investigated [8] and implemented [7]. However, the utility of such a scheme for high-performance machines is limited because of the complex logic that is needed to determine the appropriate bank/word from a given address.

In a high-order interleaved memory system, consecutive memory addresses in the linear address space lie in the same bank. Therefore, if the memory is referenced sequentially, consecutive memory references access the same bank and no increase in bandwidth is obtained. In a low-order interleaved memory system, consecutive addresses lie in different banks. Thus, if the memory is accessed sequentially, consecutive references will access different banks thereby increasing the bandwidth of the memory. Since the memory referencing

Manuscript received June 5, 1987; revised February 12, 1988 and December 16, 1988. This work was supported in part by the National Science Foundation under Grants DCR-8509194 and CCR-8706722, Air Force Grant AFOSR 84-0052 and NSF Grant MIT88-05586.

K. C. Cheung is with the Digital Equipment Corporation, Hong Kong.

G. S. Sohi and K. K. Saluja are with the University of Wisconsin, Madison, WI 53706.

D. K. Pradhan is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst.

IEEE Log Number 8931926.

[1] Throughout this paper the C-access configuration of the memory banks is assumed for the interleaved memory.

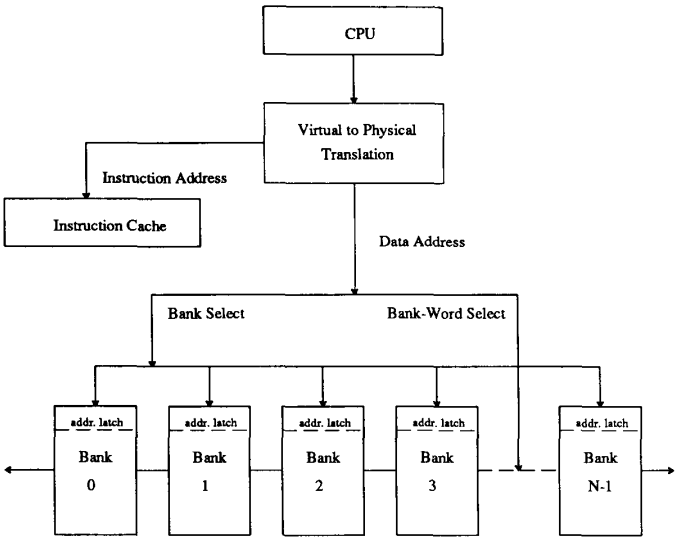


Fig. 1. A processor with an interleaved memory system.

pattern for most programs is generally sequential (because of sequential instructions and array structures with a constant stride of unity), a low-order interleaved memory system generally has a higher bandwidth than a high-order interleaved memory system.

A low-order interleaving scheme has a major drawback—it is not modular, i.e., a failure in a single bank affects the entire address space [11]. If no precautions are taken to handle such a situation, the bandwidth of the memory and consequently the performance of the processor could be degraded to an intolerable extent.

In this paper, we study the organization of interleaved memories such that faults in the memory system degrade the performance in a graceful manner. We restrict our study to an interleaved memory system that starts out with 2^q memory banks and uses a low-order interleaving scheme. The ideas presented in this paper can easily be extended to other interleaved memory schemes.

Section II briefly describes the motivation and design objectives of the memory system. In Section III, a new reconfiguration scheme is presented. Section IV presents the design of the hardware needed to implement the reconfiguration scheme proposed in Section III. The reconfiguration scheme is evaluated using *trace-driven* simulation for a number of benchmarks in Section V. A final section presents concluding remarks.

II. Faults in Interleaved Memories

Consider a memory that consists of one or more *groups* of interleaved banks in the memory system. A group consists of 2^r banks (where r is an integer) that are fully interleaved using low-order interleaving. Thus, the banks within a group can be selected using an r-bit bank selection address field. Different groups can have a different number of banks in them. For example, group G_1 may consist of four banks while group G_2 may have only one bank. If the total number of banks in the memory system is 2^k where k is an integer, then there is only one group.

A conventional interleaved memory consists of a single group of 2^q banks. If each bank contains 2^p words, the total addressable main memory of the system is 2^n (where $n = p + q$) words. Using a standard low-order interleaving scheme, q bits, i.e., $A_{q-1}A_{q-2}\cdots A_0$ of the n-bit address $A_{n-1}A_{n-2}\cdots A_0$ (where A_{n-1} is the most significant bit), are used to select the bank and the remaining p bits, i.e., bits $A_{n-1}A_{n-2}\cdots A_q$ are used to select a word within a bank.

In this paper, we are interested in investigating the situation when one or more banks of memory fail. Therefore, the fault model that we shall use in this paper is that a fault(s) results in the loss of a complete bank(s) of memory. We assume that a mechanism that detects the presence of a faulty bank exists. Such a fault-detection scheme is not the subject matter of this paper. Our main thrust is to evaluate the loss in performance when a fault is reported and how might the memory system be organized so that the resulting degradation in performance is graceful.

Consider what happens when a bank is deleted from a memory system that consists of a single group of 2^q banks. This is exactly the situation when a bank fails in a standard interleaved memory system. The number of banks in memory is reduced to $2^q - 1$ and the total addressable physical memory is reduced to $(2^q - 1)2^p$ words. However, since $2^q - 1$ is not a power of 2, the banks no longer form a single group and the system loses its capacity to interleave memory requests in the low-order interleaving manner. Without interleaving, the bandwidth of the memory system reduces to the bandwidth of a single bank. Such a loss in memory bandwidth can be catastrophic to the performance of a high-speed CPU.

When a fault occurs, program execution must be halted and the address translation mechanism informed about the faulty

bank. Correct information is recovered from a backup store and program execution restarted (or restored from a checkpoint if a recovery scheme is used). Unfortunately, if the memory system is not able to perform near its fault-free performance level, overall processor performance will suffer. What could we possibly do to salvage some of this lost memory performance?

We could potentially maintain spare memory banks and, on identifying a faulty bank, a spare bank could be switched into its place. An organization of such a memory system is described in [3]. A limitation of this method is that as more banks become faulty, the system will eventually run out of spares if the spare banks cannot be replaced. Once all the spare banks have been exhausted, another fault-tolerance scheme must come into play.

An alternative approach is to reconfigure the remaining nonfaulty banks in order to salvage some of the lost performance. Clearly, such an approach could also be used if a system has spares but runs out of them eventually. The remaining banks have to be reconfigured so that interleaving is possible. Before proceeding further, let us see how the presence of a faulty bank affects memory system performance. A faulty bank degrades memory performance in two ways: 1) the number of fault-free banks is reduced thereby reducing the available bandwidth and 2) the amount of available physical memory is reduced thereby increasing the page fault rate.

How might we organize the fault-free banks so that the performance is not degraded to an intolerable extent? A simple solution that could be used to salvage some of the lost bandwidth is to reduce the number of addressable banks to the nearest power of two, i.e., 2^{q-1}, thereby achieving a maximum bandwidth of 2^{q-1} words per memory cycle. While the hardware that allows this translation and the resulting address translation and bank selection mechanism is quite straightforward, $2^{q-1} - 1$ banks of fault-free physical memory are not addressable and therefore are unutilized (note that even such a simple scheme requires additional hardware to implement the reconfiguration). The resulting memory configuration with 2^{q-1} banks is likely to result in a higher page fault rate than a memory system that uses all the nonfaulty banks. Furthermore, a memory system with X ($2^{q-1} < X \leq 2^q - 1$) banks could potentially result in a higher bandwidth than a memory system with 2^{q-1} banks (as we shall see in Section V). Therefore, we must use a reconfiguration scheme that uses as many fault-free banks as possible to salvage the memory bandwidth and, at the same time, minimize the degradation due to the smaller memory size. Also, the hardware needed to implement the reconfiguration scheme should be simple enough so that it does not degrade fault-free memory performance significantly.

III. The Reconfiguration Scheme

The scheme proposed in this paper reconfigures the remaining banks using a combination of high-order interleaving and low-order interleaving. Before explaining the reconfiguration scheme, we distinguish between three address spaces: 1) a *virtual* address space that is seen by the program, 2) a *logical* address space that consists of the fault-free memory banks (each with 2^p words), and 3) a *physical* address space that consists of actual addresses in the physical memory system. Further distinction between the logical and the physical address spaces will become clear in the remainder of this section and in an example (Fig. 5) that we shall use in Section IV.

Addresses in the logical and physical spaces are specified as a bank number and an address within the bank. In the absence of faults, there is a single group of banks and the logical and physical address spaces are the same. On occurrence of a fault, the faulty bank is switched out and the memory reconfigured. We assume that program execution can be restarted (or restored if a recovery scheme is used) from the backup memory. Addresses generated by the user program are still complete virtual addresses; the program does not know about the loss of a memory bank. The logical address space is reduced in a systematic manner. The virtual memory management process is informed about the loss of banks and the new logical configuration of the memory; it views the loss of a memory bank as the loss of a few page frames (equal to one bank) of memory. The virtual to logical translation process makes sure that no information is placed in the unavailable logical space and, for interleaved access to the elements of a page, it places a page entirely within the banks of a single group. As the program executes, pages are brought in from the backup store into the remapped main memory. The logical addresses are translated into physical addresses by the reconfiguration hardware (described in Section IV) depending upon the actual banks that have failed.

A logical address specifies a *logical* bank number and a word within the logical bank. For a single faulty bank, there are $2^q - 1$ nonfaulty logical banks and one faulty logical bank. The number of faulty logical banks is the same as the number of faulty banks. The faulty logical bank is numbered $2^q - 1$ and the nonfaulty logical banks are numbered from 0 through $2^q - 2$. Nonfaulty logical banks are partitioned into sets. Thus, if $2^q - 1$ logical banks were available, they would be partitioned into q sets. These q sets form two subsets; subset $S_0(2^{q-1})$ consisting of a single group of 2^{q-1} logical banks and subset $S_1(2^{q-1} - 1)$ defined recursively as consisting of two subsets $S_0(2^{q-2})$ and $S_1(2^{q-2} - 1)$. Therefore, $S_0(2^{q-1})$ has one group $G_0(2^{q-1})$ that has 2^{q-1} logical banks and $S_1(2^{q-1} - 1)$ is made up of group $G_{10}(2^{q-2})$ which has 2^{q-2} logical banks and the subset $S_1(2^{q-2} - 1)$ which has $2^{q-2} - 1$ banks. This recursive partition stops when S_1 has only one logical bank. An alternate way (suggested to us by an anonymous referee) of looking at this partioning of logical banks into groups is as follows. Write down the number of nonfaulty logical banks as a binary number $b_{q-1}b_{q-2}\cdots b_i \cdots b_0$. There is a group with 2^i banks if bit i of the binary number is 1. As we shall see in the following paragraph, the concept of sets is useful in understanding the logical address decoding process. An example of the partitioning of seven logical memory banks into groups is given in Fig. 2. With seven fault-free banks, there are three groups G_0, G_{10} and G_{11} with four, two, and one banks, respectively.

The 2^k banks within a group $G_i(2^k)$ are organized for low-order interleaving; high-order address bits are used to deter-

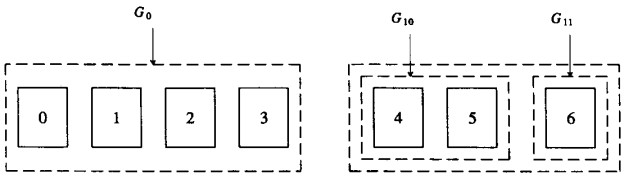


Fig. 2. Partitioning logical banks into groups.

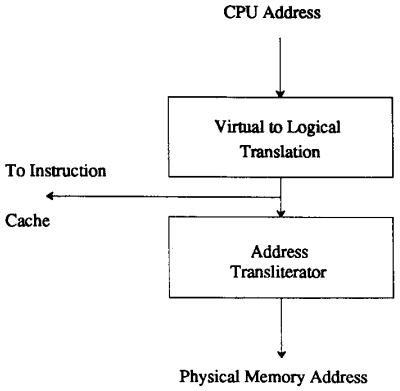


Fig. 3. Two-level address translation with an address transliterator.

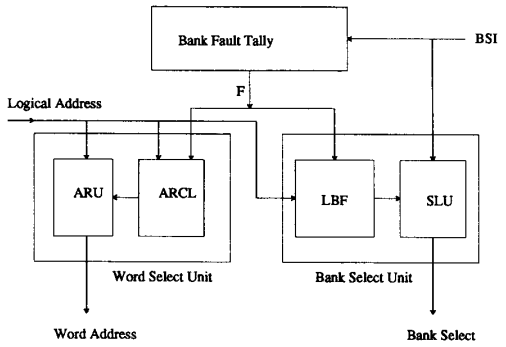


Fig. 4. The address transliterator.

mine the group. If there is only one group (for example, in the fault-free case), no group selection needs to be done. The low-order q bits of the address select the logical bank and the high-order p bits of the address select the word within the logical bank. With one fault, the number of groups becomes q with the number of logical banks $2^q - 1$ and q bits suffice to uniquely identify $2^q - 1$ nonfaulty logical banks and one faulty logical bank. An address is decoded as follows: the most significant bit of the address, A_{n-1}, is used to select either subset $S_0(2^{q-1})$, i.e., group $G_0(2^{q-1})$ or subset $S_1(2^{q-1} - 1)$. If group $G_0(2^{q-1})$ is selected, then bits $A_{q-2} \cdots A_0$ are used to select one of 2^{q-1} logical banks within this group and bits $A_{n-2} \cdots A_{q-1}$ are used to address the word within the logical bank. If $S_1(2^{q-1} - 1)$ is selected, then bit A_{n-2} of the address is used to select either $G_{10}(2^{q-2})$ (with bits $A_{q-3} \cdots A_0$ used to select a logical bank within this group) or $S_1(2^{q-2} - 1)$ and so on. Note that this group identification scheme resembles the decoding scheme used to decode Huffman-encoded information. Once the group number and the bank has been determined from the address using q bits, the remaining p bits are used to select the word within the logical bank.

The logical banks must now be mapped onto the *physical* banks of the memory system. For example, in a system with eight banks if physical bank 3 is faulty, logical bank 7 (the unavailable logical bank) must be mapped onto physical bank 3 and logical banks 0–6 must be mapped onto the remaining physical banks. The logic that decodes the address and generates the appropriate bank select and word select signals is now more complex than a simple decoder. We call this logic the *address transliterator (AT)*. Each memory request that is serviced by the physical memory now passes through the AT before it is forwarded to the physical memory system (Fig. 3). The design of the AT is discussed in detail in the next section. The inputs to the AT are an n-bit logical memory address and a 2^q bit vector, the *bank status indicator (BSI)*, that indicates the status of each bank. The BSI vector consists of a single bit flag for each bank. The flag is set to 0 if the bank is fault-free (available) and 1 if the bank is faulty (unavailable). The BSI vector is updated as faults occur and are detected. The output from the AT is the appropriate physical bank address and the address of the word within the selected bank.

On the occurrence of another fault, the program is stopped and the memory is remapped again. Thus, in the presence of a second fault, the smallest logical group that contains only one bank is eliminated. The AT hardware is informed (through the BSI vector) and it responds by modifying its group numbering accordingly. The program is then restarted and continues to execute, albeit with degraded memory performance.

IV. The Address Transliterator

The AT hardware consists of three parts as shown in Fig. 4: 1) a *bank fault tally (BFT)*, 2) a *bank select unit (BSU)*, and 3) a *word select unit (WSU)*. The inputs to the AT consist of the n-bit logical address and the BSI vector. The BFT determines the number of faulty banks using the BSI vector. The BSU is responsible for generating the physical bank select signals for a given logical address and the WSU is responsible for supplying the address within the selected bank. Before proceeding with the details of the AT, we would like to mention that the partitioning of the AT into distinct components is done solely to facilitate the understanding of its operation. For implementations of the AT, the above demarcation is not necessary. However, the overall operation of the logic will remain unchanged. Depending upon the technology used to realize the AT, such a demarcation may or may not offer optimum performance. In fact, it may be possible to implement

TABLE I
LARGEST VALID LOGICAL ADDRESS WITH k FAULTY BANKS

k	A_{n-1}	A_{n-2}	A_{n-3}	A_{n-4}	$\cdots$	A_0
7	0	0	0	1	$\cdots$	1
6	0	0	1	1	$\cdots$	1
5	0	1	0	1	$\cdots$	1
4	0	1	1	1	$\cdots$	1
3	1	0	0	1	$\cdots$	1
2	1	0	1	1	$\cdots$	1
1	1	1	0	1	$\cdots$	1
0	1	1	1	1	$\cdots$	1

portions of the AT within the address decoders of the memory banks.

A. The Bank Fault Tally

The bank fault tally (BFT) determines the number of faulty banks in the memory system from the information given in the 2^q bit BSI vector. The BFT provides a $2^q + 1$ bit output in the form of decoded fault count indicators $F = F_{2^q}, \cdots, F_0$, where $F_i = 1$ if and only if there are exactly i faulty banks in the memory system. Thus, for a fault-free memory system, F_0 is 1 and all the remaining fault count indicators are 0. Note that the number of faulty banks in the memory system is given by the number of 1's in the BSI vector and that the BFT is a combinational circuit.

It is possible to have the outputs of the BFT in an encoded form. This will reduce the number of interconnecting lines between BFT and other subunits of the AT, but then the coded information may need to be decoded by other subunits. Note that the BFT realizes a tally circuit [10]. Throughout this paper, we use the decoded form of the outputs of the BFT for ease of understanding.

B. Bank Select Unit

The bank select unit consists of two subunits, the *logical bank finder (LBF)* and the *switching logic unit (SLU)*.

1) Logical Bank Finder (LBF): The LBF determines the logical bank number of the bank being accessed for a given logical address. Recall that in the presence of a single faulty bank, the faulty bank is represented by a string of 1's in the high-order q bits of the logical address. This result can be generalized for multiple faults. Table I shows the largest valid logical address assuming that the memory system consists of eight banks and there are exactly k, $0 \leq k < 8$, faulty banks. In the general case, given a logical address, the LBF determines the logical bank number of the bank being addressed based on the following inputs:

1) $2q$ bits (i.e., high-order q bits and low-order q bits) of the address and

2) $2^q + 1$ bits from the BFT indicating the fault count.

The q bits of the logical address that are used to select the logical bank depend upon the number of faults. The LBF has 2^q outputs, denoted as $B_{2^q-1}, \cdots, B_0$, that represent the decoded form of the logical bank numbers.

The operation of the LBF is best illustrated by an example. Table II gives the 3-bit logical bank number $L_2L_1L_0$ of the addressed bank as a function of 6 bits from the complete logical address, i.e., $(A_{n-1}A_{n-2}A_{n-3}A_2A_1A_0)$, and the number of faults k for the case of eight ($q = 3$) banks. In the fault-free case, the low-order bits $A_2A_1A_0$ are used to select the logical bank. In the presence of a single fault, the seven fault-free logical banks are divided into three groups of four, two, and one banks, respectively. If bit A_{n-1} is 0, the group of four logical banks is selected and bits A_1A_0 are use to select the logical bank within the group (see bold row in Table II). If bit A_{n-1} is 1, then the set of three logical banks is selected and bit A_{n-2} is used to distinguish between the two groups of banks within the set. Other entries in the table can be interpreted in a similar fashion.

TABLE II
LOGICAL BANK NUMBER SELECTION IN CASE OF k FAULTY BANKS

k	A_{n-1}	A_{n-2}	A_{n-3}	L_2	L_1	L_0
7	X	X	X	0	0	0
6	0	0	X	0	0	A_0
5	0	0	X	0	0	A_0
	0	1	0	0	1	0
4	0	X	X	0	A_1	A_0
3	0	X	X	0	A_1	A_0
	1	0	0	1	0	0
2	0	X	X	0	A_1	A_0
	1	X	X	1	0	A_0
1	**0**	**X**	**X**	**0**	**A_1**	**A_0**
	1	0	X	1	0	A_0
	1	1	0	1	1	0
0	X	X	X	A_2	A_1	A_0

The LBF is a combinational circuit that implements a set of independent Boolean equations of the inputs A_i and F_i. For reasons of brevity, we do not present the exact Boolean equations for the LBF in this paper. The interested reader is referred to [3].

2) Switching Logic Unit (SLU): The SLU maps a logical bank number obtained from the LBF onto a physical bank number, i.e., it provides the select signals for the physical memory banks. A faulty bank is never selected. If there are no faulty banks in the memory system, the physical bank number of every bank is same as the logical bank number as shown in Fig. 5(a). However, in the presence of faulty banks in the memory system, the logical bank numbers are remapped by the SLU onto physical bank numbers as follows. If there is one faulty bank in the memory system, say physical bank number m, then a logical bank number $m+i$, $i \geq 0$, is remapped to the physical bank number $m+i+1$. Thus, if the faulty bank is bank number 1, physical bank number 2 will be selected for logical bank number 1 as shown in Fig. 5(b). For multiple faults, the remapping mechanism is extended in a natural manner and always selects the next available fault-free physical bank. The inputs to the SLU consist of 2^q-bit BSI vector, containing the location of faulty banks in the memory system, and the 2^q outputs of the LBF. The outputs of the SLU are 2^q physical bank select signals $(BS_0, \cdots, BS_{2^q-1})$ where BS_i is used to select the physical bank number i. This logic can, therefore, be realized using switches or multiplexors [3].

C. Word Select Unit

In a fault-free memory system, the high-order p bits of the address determine the address of the word within a bank.

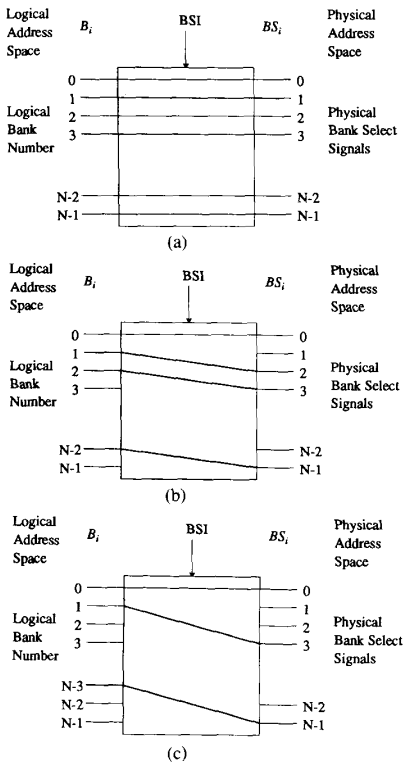


Fig. 5. (a) Logical to physical bank mapping with no faults. (b) Logical to physical bank mapping with physical bank 1 faulty. (c) Logical to physical bank mapping with physical banks 1 and 2 faulty.

However, if faulty banks exist in the memory system, the address of the word within a bank depends on the number of faulty banks in the system as well as the logical bank number of the selected bank. For example, for a memory system with eight banks ($q = 3$) and with one faulty bank, if a memory reference selects logical bank 5 then the word address within logical bank 5 is given by bits $A_{n-3}, A_{n-4}, \cdots, A_1$. In general, if for a given address some high-order j bits are used to determine the logical bank number then the word address within the bank is given by $(A_{n-j}, \cdots, A_{q-j+1})$. The function of the WSU, therefore, is to extract the appropriate p bits that represent the address of the word within the selected bank from the n-bit logical address. The WSU accomplishes this by using two subunits, the *address reformulation control logic (ARCL)* and the *address reformulation unit (ARU)*.

1) The Address Reformulation Control Logic (ARCL): This subunit determines the p bits that represent the word address. The p bits are then extracted in the ARU by using shift-left by i bit or SL_i operations on the input address and retaining the high-order p bits. The ARCL determines the value of i, i.e., the SL_i signals from the information provided in the high-order q bits of the logical address and the number of faults obtained from the BFT. The SL_i signals are then provided to the ARU.

Rather than present the detailed Boolean equations, we again illustrate the operation of the ARCL with the help of an example. Table III presents the SL_i signals for $q = 3$ and k faults. In the fault-free case, the high-order p bits themselves represent the word address, i.e., $SL_0 = 1$. In the case of two faulty banks, we have two groups of four and two banks, respectively. If the group of four banks is addressed, i.e., $A_{n-1} = 0$ (see the bold row in Table III), the logical address needs to be shifted left by 1 ($SL_1 = 1$) and the high-order p bits retained for the word address. If the group of two banks is selected ($A_{n-1} = 1$), then the logical address needs to be shifted left by 2 ($SL_2 = 1$) and the high-order p bits extracted.

TABLE III
CONTROL SIGNALS OF THE ARCL IN CASE OF k FAULTY BANKS

k	A_{n-1}	A_{n-2}	A_{n-3}	SL_3	SL_2	SL_1	SL_0
7	X	X	X	1	0	0	0
6	0	0	X	0	1	0	0
5	0	0	X	0	1	0	0
	0	1	0	1	0	0	0
4	0	X	X	0	0	1	0
3	0	X	X	0	0	1	0
	1	0	0	1	0	0	0
2	**0**	**X**	**X**	**0**	**0**	**1**	**0**
	1	X	X	0	1	0	0
1	0	X	X	0	0	1	0
	1	0	X	0	1	0	0
	1	1	0	1	0	0	0
0	X	X	X	0	0	0	1

2) The Address Reformulation Unit (ARU): The ARU accepts the n-bit address and the shift control signals SL_i as inputs and provides the p-bit word address. The ARU is, therefore, a simple shifter and can be realized using multiplexors.

D. An Example

Through the following example we illustrate the operation of the complete AT. Let us consider a memory system consisting of eight memory banks ($q = 3$) with a 16 bit address ($n = 16$). If physical banks 1 and 2 are faulty, the BSI vector will be 00000110, where the most significant bit of the BSI vector indicates the status of the physical bank 7. If the input address is 1001111111111100, from Table II we see that logical bank 4 of group G_{10} is selected. Therefore, 3 bits of the address $(A_{n-1}A_{n-2}A_0)$ are used to identify the logical bank and the remaining bits of the address $(A_{n-3}A_{n-4}\cdots A_2A_1)$ are used as the word address within the bank. The remapping of the logical bank numbers, i.e., outputs of the LBF, to the physical bank select signals by the SLU is shown in Fig. 5(c). As the address given above generates the bank address for the logical bank 4, the physical bank 6 is selected. The outputs of the subunits of the AT are summarized below in Table IV.

E. Logic Delays in the AT

For a conventional low-order interleaved memory system, shown in Fig. 1, the address bits are transferred on two paths between the processor and the memory. The two paths are the bank select path and the word select path. The logic present in either path is a simple decoder. By using an AT, additional logic is inserted in both these paths. Since the delays introduced by the extra logic will be of a different nature for different technologies and different implementations of the AT,

TABLE IV
OUTPUTS OF THE SUBUNITS OF THE AT; INPUT ADDRESS = 1001111111111100 AND $k = 2$

Unit	Inputs	Outputs
Bank Fault Tally (BFT)	BSI=00000110	$F_2 = 1$, $F_i = 0$ for $0 \le i \le 7$ and $i \ne 2$.
Logical Bank Finder (LBF)	F_i 's and $A_{n-1}A_{n-2}A_{n-3}$=100 $A_2A_1A_0$ =100	$B_4 = 1$, $B_i = 0$ for $0 \le i \le 7$ and $i \ne 4$.
Switching Logic Unit (SLU)	B_is and BSI vector	$BS_6 = 1$, $BS_i = 0$ for $0 \le i \le 7$ and $i \ne 6$.
Address Reformulation Control Logic (ARCL)	F_i 's and $A_{n-1}A_{n-2}A_{n-3}$=100	$SL_2 = 1$, SL 's = 0 for $0 \le i \le 3$ and $i \ne 2$
Address Reformulation Unit (ARU)	16 bit address and SL_i's	0111111111110

we shall not attempt to quantify the delays in general. Rather, we give the reader a feel for the additional delays introduced by the AT and present the results for a conservative CMOS design.

The BFT does not contribute to the delay through the AT since its output does not change between faults. The longest path within the BSU is from the address inputs to the outputs of the LBF plus the delay through the SLU. The complexity of the LBF is such that it can be realized by a two-level logic circuit (gates or a PLA); the SLU can be realized as a simple switch (or multiplexor). Each of these units will, therefore, contribute a small, fixed delay. Similarly, in the WSU, the critical path is from the address inputs to the outputs of the ARCL plus the delay through the ARU. The ARCL logic is simple enough that it can be realized as a two-level logic circuit and, as commented in Section IV-C-2, the ARU can be realized using multiplexors. The operations of the BSU and the WSU are carried out in parallel and, therefore, the critical path for the AT is the longer of the critical paths of the two units. Thus, the delay through the AT is equivalent to the delay through a few levels of logic.

In order to get a better feel of the delays in the AT, we implemented the AT logic in CMOS VLSI using Magic. Each unit of the AT was designed separately as described in this paper. The AT was designed for a memory system consisting of 16 memory banks and a 32-bit logical address. The details of complete design can be found in [3]. A timing simulation using the Crystal simulator indicated a delay of 44 ns through the AT. We believe that the delay can be reduced with aggressive design; relative delays within the WSU and BSU and the methods to reduce these delays are also discussed in [3]. Even for our conservative design of the AT, for a given processor and memory technology, the delay through the AT can be kept within a single CPU clock cycle [3]. Therefore, we believe that the AT logic will not degrade fault-free memory latency to a significant extent.

V. PERFORMANCE EVALUATION

We evaluated the performance of the proposed memory reconfiguration scheme using a *trace-driven* simulation analysis. A trace of instruction and data references was obtained for several benchmark programs. Data from the trace files was then fed into a program that simulated the memory system. Since the simulator is driven by actual memory reference traces that have no timing information, the simulator assumes that consecutive memory references in the trace file occur in consecutive CPU cycles.

The simulator written by us takes into account the memory structure and the virtual memory management process [3]. The simulation model consists of a pipelined processor capable of issuing a memory request at each CPU cycle. The memory references are divided into instruction references and data references. For each memory reference a word is transferred between the memory system and the processor. We assume an instruction cache with a cycle time identical to the CPU cycle time to service instruction references. Thus, only data references go to the interleaved memory. The use of the bus alternates between the instruction and the data cycles. During an instruction cycle, instructions are fetched from the cache and the data requests are buffered in a queue. We assume that all instruction requests are satisfied by the instruction cache. During a data cycle, data requests in the data queue are allowed to access memory if no bank conflict occurs. If an access to a busy bank is detected, subsequent requests are suspended until the next available memory cycle.

A program is allocated a fixed number of data pages (maximum of 32) for its use. A least recently used (LRU) replacement policy is employed to replace a page when a page fault occurs and no free page frame is available. The pages are loaded into memory on demand. The page size is 2K bytes. Initially, we assume that only one data page is present in memory. In case of bank failures, fewer data pages are allocated to the program. The reduction in the number of pages is proportional to the number of faulty banks. The pages are distributed among the groups of the reconfigured memory in proportion to the number of banks in the group. A page lies completely within a group of banks. Pages are first loaded into the group with the largest number of banks and are then loaded into the groups with fewer banks. For example, if there are two groups consisting of eight and four banks, respectively, a process will place 67 percent of its data pages in the group of eight banks first and the remaining pages in the group of four banks next. The time to process a page fault is 2000 memory cycles [9].

Recall that faults degrade memory system performance in two ways: 1) the available memory bandwidth is reduced and 2) a reduction in the available physical memory increases the probability of a page fault. We believe that a performance metric must take into account both factors of memory performance. Therefore, we combine the effects of reduced bandwidth and increase in page faults into a single metric T similar to the metric used by Smith [13]. The performance metric T is defined as

$$T = \text{total data trace length} + \text{time to process a page fault} \times \text{number of data page faults}$$

where the data trace length is the number of data references divided by the data bandwidth. T is an indicator of the memory access time for the trace when both bandwidth and page faults are taken into account.

We realize, however, that in some situations the bandwidth

TABLE V
STATISTICS OF THE BENCHMARK PROGRAMS

Trace	Trace Records		Data Pages
	Instruction	Data	Touched
nroff	284966	175488	52
compact	234110	205468	22
boyer	217147	229871	216
tak	236628	250384	151
spice	258563	250996	55
mpla	255708	173266	94
cripta	147663	150386	221
csh	220367	211592	72

TABLE VI
PERFORMANCE RESULTS FOR BENCHMARK PROGRAMS

Benchmark	Metric	Number of Fault-Free Banks in Memory (N)								
		16	15	14	13	12	11	10	9	8
nroff	Bandwidth	3.417	3.078	3.079	3.083	3.098	3.021	3.009	3.025	3.011
	Page Faults	53	53	53	63	122	131	165	204	246
	Pages	32	30	28	26	24	22	20	18	16
	T	1	1.04	1.04	1.16	1.91	2.03	2.47	2.96	3.50
compact	Bandwidth	2.895	2.450	2.345	2.421	2.396	2.583	2.627	2.568	2.544
	Page Faults	22	23	23	44	112	326	827	1908	3760
	Pages	16	15	14	13	12	11	10	9	8
	T	1	1.13	1.16	1.50	2.69	6.36	15.07	33.88	66.10
boyer	Bandwidth	3.283	2.809	3.110	3.043	3.143	2.957	2.998	2.223	2.998
	Page Faults	1465	1696	1962	2328	2754	3209	4454	6696	8895
	Pages	32	30	28	26	24	22	20	18	16
	T	1	1.16	1.33	1.58	1.86	2.17	2.99	4.50	5.96
tak	Bandwidth	4.920	4.600	2.900	3.322	4.343	2.114	4.083	4.193	3.986
	Page Faults	319	335	365	393	449	513	585	678	1286
	Pages	32	30	28	26	24	22	20	18	16
	T	1	1.05	1.18	1.25	1.39	1.66	1.79	2.06	3.82
spice	Bandwidth	3.577	3.017	3.045	2.981	3.109	2.813	2.948	2.894	2.894
	Page Faults	80	116	160	189	231	247	274	351	407
	Pages	32	30	28	26	24	22	20	18	16
	T	1	1.37	1.75	2.01	2.36	2.53	2.75	3.42	3.91
mpla	Bandwidth	2.119	2.094	2.095	2.090	2.072	2.029	2.027	2.059	2.046
	Page Faults	129	132	136	374	2200	2543	2670	2712	2738
	Pages	16	15	14	13	12	11	10	9	8
	T	1	1.02	1.04	2.45	13.20	15.22	15.97	16.21	16.35
cripta	Bandwidth	2.569	2.466	2.477	2.399	2.417	2.190	2.366	2.201	2.321
	Page Faults	564	629	808	1044	1139	1261	1490	1684	1869
	Pages	32	30	28	26	24	22	20	18	16
	T	1	1.11	1.41	1.81	1.97	2.18	2.57	2.90	3.20
csh	Bandwidth	3.097	2.483	2.645	2.426	2.759	2.071	2.505	2.434	2.697
	Page Faults	443	507	562	627	740	909	1143	1528	5029
	Pages	16	15	14	13	12	11	10	9	8
	T	1	1.15	1.26	1.41	1.63	2.01	2.48	3.29	10.62

may be the more important metric and the number of page faults may be of secondary importance. In other cases, the page faults may be of major concern. Therefore, along with the metric T, we also present the bandwidth and the number of page faults for each one of our experiments.

A. Experiments and Results

To evaluate the performance of the reconfigurable memory system, we used the following benchmark programs: 1) *nroff*, a text formatting program, 2) *compact*, a program for file compaction using adaptive Huffman encoding, 3) *boyer*, a theorem proving program [4], 4) *tak*, an execution of the Takeuchi function [4], 5) *spice*, a circuit simulation program, 6) *mpla*, a PLA generation program for the Magic layout editor, 7) *cripta*, an encryption program written in Lisp, and 8) *csh*, a command interpreter for the UNIX operating system.

The number of instruction and data references traced for the above programs are given in Table V. The total number of data pages in each trace are also given in this table.

The traces for each of these programs were fed to the trace-driven simulator and the bandwidth, the page faults, and the value of the performance metric computed. To account for the presence of faulty banks, each trace is simulated for reduced number of addressable banks and the number of pages allocated to each program reduced in proportion to the amount of memory lost. The T metric is normalized with respect to the fault-free case. The results of our experiments are presented in Table VI. The bandwidth in this table is defined as the number of busy banks per memory cycle. From the table, one observation is quite obvious—a decrease in the available physical memory increases the number of page faults. We also observe that, in most cases, the bandwidth when α banks ($9 \leq \alpha \leq 15$) are used is better than the bandwidth that could be achieved with eight banks. If bandwidth alone is the major performance-determining factor, then the reconfiguration scheme could be used to reconfigure part or all of the remaining fault-free banks (for example, we could choose to reconfigure only 12 banks in the case of one, two, three,

or four faults). If the memory capacity is a limiting factor, then page faults play an important role in the overall memory access time. Our reconfiguration scheme allows the use of all the fault-free memory thereby minimizing the number of page faults. The resulting degradation in memory system performance (as measured by T) is quite graceful.

Based on the experimental results we can conclude that the proposed reconfiguration scheme allows for the graceful degradation of interleaved memory systems. In situations where the memory capacity is unimportant, the reconfiguration scheme is able to reconfigure the fault-free banks so that the resulting memory configuration has a better bandwidth than a memory configuration with the next lower power-of-2 number of banks. In situations where the memory system capacity is a limiting factor, the reconfiguration scheme is able to reconfigure the memory to minimize the number of page faults and, at the same time, recover part of the lost memory bandwidth.

VI. Conclusions

In this paper, we have presented the design of an interleaved memory system whose performance degrades gracefully in the presence of faulty banks. We discussed the details of such a design and evaluated its performance using a trace-driven simulation. Our simulation results show that the performance of an interleaved memory system that employs the design proposed in this paper does indeed degrade gracefully in the presence of faults. Furthermore, the address translation mechanism needed for graceful degradation does not increase the memory latency significantly.

Acknowledgment

The authors are thankful to Prof. C. R. Kime and to the anonymous referees for their helpful comments and suggestions.

References

[1] D. P. Bhandarkar, "Analysis of memory interference in multiprocessors," *IEEE Trans. Comput.*, vol. C-24, pp. 897–908, Sept. 1975.

[2] G. Burnett and E. G. Coffman, "A study of interleaved memory systems," in *Proc. AFIPS 1970 Spring Joint Comput. Conf.*, 1970, pp. 467–474.

[3] K. C. Cheung, "Organization and analysis of interleaved memory systems," M.S. Thesis, Dep. Elec. Comput. Eng., Univ. of Wisconsin-Madison, Madison, WI, 1987.

[4] R. P. Gabriel, *Performance and Evaluation of Lisp Systems*. Cambridge, MA: MIT Press, 1985.

[5] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

[6] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.

[7] D. J. Kuck and R. A. Stokes, "The Burroughs scientific processor (BSP)," *IEEE Trans. Comput.*, vol. C-31, pp. 363–376, May 1982.

[8] D. H. Lawrie and C. R. Vora, "The prime memory system for array access," *IEEE Trans. Comput.*, vol. C-31, pp. 435–442, May 1982.

[9] M. Malkawi and J. H. Patel, "Performance measurement of paging behavior in multiprogrammed systems," in *Proc. 13th Annu. Symp. Comput. Architecture,* June 1986, pp. 111–118.

[10] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[11] D. K. Pradhan, *Fault Tolerant Computing: Theory and Techniques*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[12] B. R. Rau, "Program behavior and the performance of interleaved memories," *IEEE Trans. Comput.*, vol. C-28, pp. 191–199, Mar. 1979.

[13] A. J. Smith, "A modified working set paging algorithm," *IEEE Trans. Comput.*, vol. C-25, pp. 907–914, Sept. 1976.

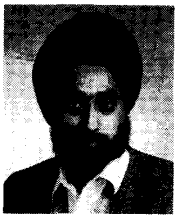
[14] ——, "Cache memories," *ACM Comput. Surveys,* vol. 14, pp. 473–530, Sept. 1982.

[15] J. E. Smith and J. R. Goodman, "A study of instruction cache organizations and replacement policies," in *Proc. 10th Annu. Symp. Comput. Architecture,* June 1983, pp. 117–123.

[16] F. W. Terman, "A study of interleaved memory systems by trace driven simulation," in *Proc. Symp. Simulation Comput. Syst.*, 1976, pp. 3–9.

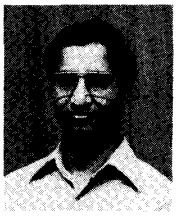
Kaifung C. Cheung (M'88) received the B.S.E.E. degree from the University of Southern California, Los Angeles, in 1985 and the M.S.E.E. degree from the University of Wisconsin-Madison, Madison, in 1987.

He is now with the Far East Local Engineering Group of the Digital Equipment Corporation, Hong Kong, where he is working on local language software products. His current interests include computer architecture, VLSI designs, and computer networks.



Gurindar S. Sohi (S'85–M'85) received the B.E.(Hons.) degree in electrical engineering from the Birla Institute of Science and Technology, Pilani, India in 1981 and the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois, Urbana-Champaign, in 1983 and 1985, respectively.

Since September 1985, he has been with the Computer Sciences Department of the University of Wisconsin-Madison where he is currently an Assistant Professor. His interests are in computer architecture, parallel and distributed processing, and fault-tolerant computing.



Kewal K. Saluja (S'70–M'73) received the B.E. degree in electrical engineering from the University of Roorkee, Roorkee, India, in 1967, and the M.S. and Ph.D. degrees from the University of Iowa, Iowa City, in 1972 and 1973, respectively.

He is currently in Associate Professor in the Department of Electrical and Computer Engineering at the University of Wisconsin–Madison where he teaches logic design, computer architecture, microprocessor-based systems, VLSI design and testing. Previously he was at the University of Newcastle, Australia. He has also held visiting and consulting positions at number of institutions such as the University of Southern California, the University of Iowa, and Hiroshima University. His research interests include design for testability, fault-tolerant computing, VLSI design, and computer architecture.

Dhiraj K. Pradhan (S'70–M'72–SM'80–F'88) is currently a Professor in the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst. Previously, he has held positions with the University of Regina, Sask., Canada, Oakland University, Rochester, MI, Stanford University, Stanford, CA and IBM Corporation. Also, he has served as a consultant to various industries. He has been actively involved with research in fault-tolerant computing, testing, computer architecture, and parallel processing since receiving the Ph.D. degree in 1972. He has published numerous papers in these areas. He is editor and coauthor of the book entitled, *Fault-Tolerant Computing: Theory and Techniques, Vols. I and II* (Englewood Cliff, NJ: Prentice-Hall).

Dr. Pradhan has edited the Special Issue on Fault-Tolerant Computing of the IEEE Transactions on Computers (March 1980). Also, he has served as session chairman and Program Committee Member for various conferences. He was the co-chairman for the 1988 IEEE Workshop on Fault-Tolerant Parallel Distributed Systems.