Software-Implemented EDAC Protection Against SEUs

Philip P. Shirvani, *Student Member IEEE*
Stanford University, Stanford

Nirmal Saxena, *Senior Member IEEE*
Stanford University, Stanford

Edward J. McCluskey, *Life Fellow IEEE*
Stanford University, Stanford

Key Words and Phrases: EDAC, ECC, software-implemented, memory protection, single-event upset, SEU, soft errors, error detection and correction, low-cost fault tolerance, transient error, COTS in space, memory bit-flips.

SUMMARY & CONCLUSIONS

In many computer systems, the contents of memory are protected by an *error detection and correction* (EDAC) code. Bit-flips caused by *single event upsets* (SEUs) are a well-known problem in memory chips and EDAC codes have been an effective solution to this problem. These codes are usually implemented in hardware using extra memory bits and encoding/decoding circuitry. In systems where EDAC hardware is not available, the reliability of the system can be improved by providing protection through software. Codes and techniques that can be used for software implementation of EDAC are discussed and compared.

We look at the implementation requirements and issues, and present some solutions. We discuss in detail how system-level and chip-level structures relate to multiple error correction. A simple solution is presented to make the EDAC scheme independent of these structures.

The technique presented in this paper was implemented and used effectively in an actual space experiment. We have observed that SEUs corrupt the operating system or programs of a computer system that does not have any EDAC for memory, forcing us to frequently reset the system. Protecting the entire memory (code and data) may not be practical in software. However, we have demonstrated that software-implemented EDAC is a low-cost solution that can provide protection for code segments and can significantly enhance the availability of a system in a low-radiation space environment. We also show this reliability improvement through analytical estimates. These estimates are based on parameter values that closely match the environment of our satellite experiment.

For applications where read and write operations are done in blocks of words, such as secondary storage systems made of solid-state memories (RAM discs), software-implemented EDAC could be a better choice than hardware EDAC, because it can be used with a simple memory system and it provides the flexibility of implementing more complex coding schemes.

1. INTRODUCTION

Acronyms:

ARGOS	Advanced Research and Global Observations Satellite
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Checking
ECC	Error-Correcting Codes
EDAC	Error Detection And Correction
MBU	Multiple-Bit Upset
OS	Operating System
SbEC-DbED	Single-byte-Error-Correcting, Double-byte-Error-Detecting
SEU	Single Event Upset
SEC-DED	Single-Error-Correcting, Double-Error-Detecting
SMU	Single-word Multiple-bit Upset

Transient errors and permanent faults in memory chips are well known reliability issues in computer systems. *Error detection and correction* (EDAC) codes — also called *error-correcting codes* (ECCs) — are the prevailing solution to this problem [1]. Typically, the memory bus architecture is extended to accommodate extra bits, and encoding and checking circuitry is added to detect and correct memory errors. This additional hardware is sometimes omitted due to cost considerations. If a computer is to be designed using *commercial-off-the-shelf* (COTS) components that do not have EDAC hardware for memory, the reliability problem has to be addressed with another form of redundancy. Hardware redundancy techniques, such as duplication or *triple modular redundancy* (TMR) [2], can be one solution, but they are very expensive. When hardware redundancy is not possible, we have to resort to software solutions.

In this paper, we discuss the implementation of EDAC in software and present a technique for a system that does not have hardware EDAC but requires protection for code and data that reside in the main memory. The goal is to provide protection against transient errors (soft errors) that manifest themselves as bit-flips in memory. These errors can be caused by *single event upsets* (SEUs) [3][4], power fluctuations or electromagnetic interference. Handling permanent faults (hard errors) in memory is discussed in the literature [1] [5] and is not the focus of this paper.

The motivation behind this work came from an actual space experiment called the Stanford ARGOS project [6]. ARGOS (Advanced Research and Global Observations Satellite) is an experimental satellite that carries several experiments, one of which is the USA experiment [7]. The USA (Unconventional Stellar Aspect) experiment includes a computing test-bed that has two processor boards. These boards are used for observing the behavior of computer systems in a radiation environment. One processor board uses a radiation-hardened processor chip set, has redundant processors (as a self-checking pair), and has EDAC hardware. The other board uses only COTS components and does not have EDAC hardware. The experiment involves collecting the errors that occur during the execution of programs in an actual space environment and comparing the performance of the two boards. We have observed that SEUs corrupt the operating system or the main control program, forcing a system reset. In order to effectively carry out our experiments, these critical programs have to be protected against SEUs. The objective of our experiment is to see whether software-implemented hardware fault-tolerance — which can include software-implemented EDAC — can provide sufficient reliability for COTS hardware to make it usable in low-radiation space applications.

Power fluctuation and electromagnetic interference may cause bit-flips in memories. It has been observed that radiation-induced transient errors also occur at ground level [8]. Therefore, the technique presented in this paper can be useful for terrestrial applications, too.

Previous discussions of software-implemented EDAC concentrate on communications and secondary storage systems [9-14]. In Sec. 2, we review some of these previous studies. In Sec. 3, we look at the problem in more detail and discuss the requirements of a scheme for the particular application presented above. Four different example EDAC coding schemes were implemented in software. These schemes are compared in Sec. 4. Issues that have to be considered for handling multiple errors and solutions to them are discussed in Sec. 5. Finally, the EDAC program has to be integrated into the whole system. We present our implementation in ARGOS in Sec. 6. The reliability improvement of an application in a space environment is estimated in Sec. 7. We conclude the paper with a discussion in Sec. 8.

2. PREVIOUS WORK

Error control coding is a well-developed field [5] [15]. EDAC codes are used to protect digital data against errors that may occur in storage media or transmission channels. The encoding and decoding of data can be done in hardware, software or a combination of both. For example, in the memory management unit (MMU) of a HaL microprocessor, error detection is done by hardware but correction is done by software, because hardware correction would increase the clock cycle time [16].

Since special hardware for a coding system can be expensive, researchers have studied the feasibility of using general-purpose microprocessors for software implementation of EDAC codes [9] [10]. Efficient software methods have been devised to do *Cyclic Redundancy Checking* (CRC) using table look-up [11] [12]. A comparison of fast implementation of different CRC codes is given in [13]. CRC codes are used for detecting multiple-bit errors in communication systems where correction can be done by retransmission. In storage systems, a coding scheme with correction capability is used. There are many different codes used in hard disks and tape backup systems. Some of these codes can be used for protecting data residing in memory chips. For example, a software implementation of a (255, 252) Reed-Solomon code that can do single-byte error correction is proposed in [14] for protecting RAM discs of satellite memories. However, there are differences between memory and secondary storage systems that need to be addressed in order to choose an appropriate EDAC scheme for memories.

The contributions of this paper are: identifying the issues in implementing EDAC in software, illustrating the options and differences in coding schemes by comparing four example codes that may be considered for EDAC, devising a technique that addresses all the requirements of software EDAC including multiple-bit error correction independent of system-level and chip-level structures, analyzing the reliability of a system with software EDAC for main memory, and finally presenting an actual implementation and demonstrating its effectiveness in an actual experiment.

3. GENERAL CONSIDERATIONS

In this section, the requirements for an EDAC scheme that is to be implemented in software are discussed. Software EDAC is an alternative to hardware-implemented EDAC. Our goal is to provide the protection capabilities of hardware EDAC in software.

3.1 Systematic Codes

A coding scheme provides a mapping of input data words to what are called *codewords*. A codeword contains extra check bits that are used for error detection and correction. Let us consider a 64-bit data word represented by the row matrix $D[d_0 d_1 ... d_{63}]$. A *single-error-correcting, double-error-detecting* (SEC-DED) Hamming code will add 8 check bits to these 64 bits and create 72-bit codewords $C[d_0 d_1 ... d_{63} c_0 c_1 ... c_7]$—denoted as a (72, 64) code. In this coding scheme, the data bits are not changed and are separable from the check bits. This type of code is called a *systematic* (or *separable*) code. In *non-systematic* codes, the data bits are not preserved and are mixed with check bits.

In a communication system, input data is given to the EDAC encoder and the check bits are calculated. The produced codewords are transmitted through the channel and given to the EDAC decoder at the receiving end. After checking for possible errors and correcting them, the decoded data is ready to be used. Similarly, in a secondary storage system such as a hard disk, the encoded data on the storage media is decoded when it is retrieved into a memory buffer for usage. Modifications are also made to the decoded data in the memory buffer and the data is re-encoded for storage. In these cases, the codewords are not accessed directly; they are always decoded before being used.

Therefore, the coding scheme used in these applications does not have to be systematic. In contrast, for the application considered in this paper, we should use a systematic code.

As mentioned in the introduction, our objective is to devise a scheme to protect the data residing in main memory. For this application, the data that is protected by software EDAC is fetched and used by the processor in the same way as unprotected data is fetched and used. We want the EDAC program to run as a background task and be transparent to other programs running on the processor. The protected data bits have to remain in their original form if we want to make the scheme transparent to the rest of the system. This requires the use of a systematic code.

3.2 Checkpoints and Scrubbing

In memories with hardware EDAC, each word of memory is encoded separately[1]. The encoding is checked on each read operation and new codewords are generated on each write operation. In addition, the contents of memory are read periodically and all the correctable errors are corrected. This latter operation is called *periodic scrubbing* and avoids accumulation of errors, thereby reducing the probability of multiple errors that may not be correctable.

If the same protection that is provided by hardware, is to be provided by software, each read and write operation done by the processor has to be intercepted. However, this interception is infeasible because it imposes a large overhead in program execution time.

---

[1] In “chipkill-correct” EDAC protected memories that are mainly used in server computers, the codewords may expand over several words [17] and therefore, single-word write operations are done in a Read-Modify-Write fashion. This is all done in hardware using store buffers and is transparent to software.

Therefore, we chose to do only periodic scrubbing for software-implemented EDAC. If memory bit-flip errors are not corrected by the periodic scrubbing before a program is executed, we rely on other software-implemented error detection techniques (e.g., assertions, *Error-Detection by Duplicated Instructions* [18], or *Control-Flow Checking by Software Signatures* [19]) to detect the errors. When an error is detected, a scrub operation is enforced before the program is restarted.

The EDAC program is given the address and size of the memory block that needs to be protected. It requests another block from the OS to be used for the check bits. Then, it calculates the check bits (encoding) and stores them in the allocated block. On request, it checks the block for errors (decoding) and corrects them if possible. The content of the memory block may be fixed or variable. If it is fixed, the encoding is done once and the check bits remain constant. However, if the memory block is written to by the processor, the check bits have to be recalculated. There are two main types of information stored in a memory: code and data. Code segments contain instructions, and data segments contain the data that is used or produced in computations. After a program has been loaded and linked by the operating system, the contents of the code segment are not changed (with the exception of self-modifying codes that are not considered here). Therefore, a fixed set of check bits can be calculated for code segments.

Generally, the processor reads and writes to data segments and, as said earlier, it is not feasible to intercept all the write operations to update the check bits because the interceptions will incur significant performance overhead. However, for data that does not change, e.g., read-only data segments, or some calculation results that are stored for later use, EDAC protection can be provided in software. *Application Program Interfaces*

(APIs) can be defined so that the programmer can make function calls to the EDAC program and request protection for a specific data segment (an example API is given in Sec. 6). In this case, protection can also be provided for writable data segments. Read and write operations on these segments will be done through the APIs in blocks of words. However, this method is not transparent to the application programs and the programmer needs to take control of the reads and writes to the protected data and minimize the execution overhead.

3.3 Overhead

The space used for check bits reduces the amount of memory available for programs and data. Therefore, we need to keep the overhead introduced by the check bits as low as possible. The simplest code is a parity code that is formed by adding a single bit to data bits such that the total number of ones in the resulting codeword is even (or odd for odd parity). This code can detect only odd numbers of errors and cannot correct any errors. Correction can be done by keeping a second copy of the parity-protected data but EDAC codes can provide correction capability with fewer check bits. We also need to handle more than one error, because multiple errors may occur between scrub intervals. Codes that have more capability (correction and multiple detection), add more check bits (*check-bit overhead*) and tend to have more complex encoding and decoding algorithms, increasing both *performance overhead* and *program size overhead*. We need to select a code that can be implemented by a fast and small program and provides correction for multiple errors. If the program is fast, it imposes low overhead on system performance. More importantly, a fast program is less vulnerable to transient errors that may occur in the processor during execution of the program. Similarly, small program size is important not just because it takes less memory space that could be used for other programs, but more importantly because, it makes the EDAC program less vulnerable to SEUs that may corrupt its own program. In Sec. 6, we discuss how the EDAC program can be protected.

Note that, the check-bit overhead of hardware EDAC is the extra memory chips that are added to the memory system to contain the check bits. There is no program size overhead for hardware EDAC but there may be some performance overhead if the latency of EDAC circuitry increases the access time of the memory. With hardware EDAC, the check bits are fetched from memory at the same time the corresponding data bits are accessed. However, with software EDAC, extra memory accesses are needed to fetch the check bits. In addition, there will be some memory accesses for fetching the EDAC program into the processor cache. Therefore, the total memory bandwidth used by software EDAC is more than that of hardware EDAC.

4. CODE SELECTION

4.1 Vertical vs. Horizontal Codes

In memory systems with hardware EDAC, the memory width is extended to accommodate the check bits. Figure 4.1(a) shows a diagram for a 32-bit memory word that is augmented with seven check bits. Each set of check bits is calculated based on the bits of one word corresponding to one address. We refer to this type of coding as a *horizontal code*. When a horizontal code is implemented in software, each word is encoded separately and the check bits are concatenated to form a word. This check word is saved in a separate address (Fig. 4.1(b)).

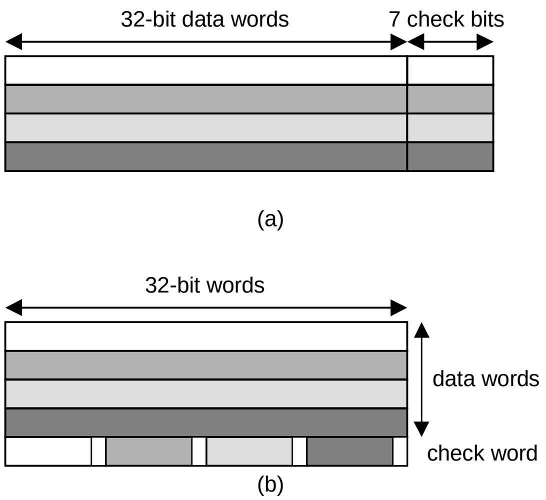

Figure 4.1 A horizontal code over bits of a word: (a) hardware implementation; (b) organization of bits when the code is implemented in software.

Another type of coding is shown in Fig. 4.2. Each set of check bits is calculated over the bits corresponding to one bit-slice of a block of words in consecutive addresses. This type of coding is used in some tape back-up systems [20] and we refer to it as a *vertical code*. This type of code matches well with the bitwise logical operations that are present in all common instruction set architectures (ISAs). When we discuss different codes in Sec. 4.2, we will see that the logical ‘`xor`’ operation is used in the implementation of most of the error detecting codes. Many shifts and logical operations are required for encoding each word in a horizontal code. In contrast, vertical codes lend themselves into very efficient algorithms that can encode all the bit-slices in parallel — similar to the parallelism in a *single-instruction multiple-data* (SIMD) machine. Therefore, a vertical code is preferred for a software-implemented EDAC scheme.

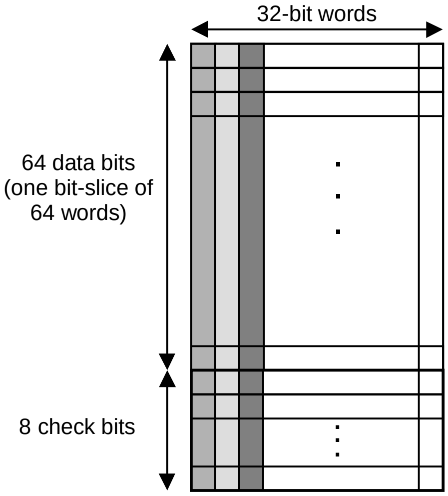

Figure 4.2 A vertical code over bit-slices of words.

Another aspect of these two types of codes is their handling of multiple errors. Let us assume that a SEC-DED code is used for both types of codes. If two bit- flips occur in one word, the horizontal code cannot correct it; but, since each bit-flip belongs to a different bit-slice, the vertical code will be able to correct both errors. On the other hand, if two bit-flips occur in one bit-slice of a block, a horizontal code will correct both, while a vertical code will fail. We discuss the occurrence and handling of multiple faults in Sec. 5.

There are coding schemes that are not quite horizontal or vertical. An advantage of implementing EDAC in software is that it is very flexible and the designer can mix different techniques and codes that would be expensive or infeasible in hardware.

4.2 Coding Schemes

In this section, we look at four different codes and compare them. These codes were chosen to illustrate the options, the differences, and the facts that need to be considered in choosing a coding scheme. The designer of a software-implemented EDAC scheme may choose a code depending on the application.

1) Scheme 1 is a (72, 64) Hamming code implemented as a vertical code over a block of 64 data words with eight check-bit words. The parity generation matrix was optimized to have minimum-weight columns. For example, the equation for the first check bit c_0 is:

$$\begin{aligned} c_0 = d_0 \oplus d_6 \oplus d_7 \oplus d_{12} \oplus d_{13} \oplus d_{15} \oplus d_{16} \oplus d_{19} \oplus d_{20} \oplus d_{26} \oplus d_{29} \oplus d_{31} \oplus d_{34} \oplus d_{35} \oplus \\ d_{38} \oplus d_{43} \oplus d_{45} \oplus d_{47} \oplus d_{48} \oplus d_{50} \oplus d_{51} \oplus d_{56} \oplus d_{60} \oplus d_{61} \oplus d_{62} \oplus d_{63} \end{aligned} \quad (1)$$

where ⊕ denotes the `xor` operation. This equation can be used directly in the C program that implements the EDAC algorithm. By defining each c_i and d_i in (1) as a 32-bit word, we can implement a vertical code as shown in Fig. 4.2. Using the bitwise '`xor`' instruction, 32 `xor`'s will be done in parallel. In other words, the encoding of all the 32 bit-slices can be done in parallel. The decoding process is done in a similar way.

This Hamming code can correct single errors and detect double errors. Therefore, in this scheme, a single bit error can be independently corrected in each bit-slice. In other words, as many as 32 bit-flips can be corrected as long as each of them is in a different bit-slice (this includes a single word correction).

2) Scheme 2 is also a vertical code with the same size but uses a cyclic code instead of a Hamming code. The (72, 64) cyclic code is based on the primitive polynomial: $P(X) = X^8 + X^7 + X^2 + 1$. The polynomial division used in this code was done by implementing the Linear Feedback Shift Register (LFSR) shown in Fig. 4.3, in software. Similar to Scheme 1, the encoding/decoding process of the 32 bit-slices is done in parallel. The correction capability of this scheme is the same as that of Scheme 1.

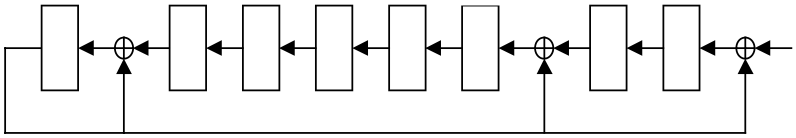

Figure 4.3 The LFSR corresponding to polynomial $P(X) = X^8 + X^7 + X^2 + 1$.

3) For Scheme 3, we chose a (1088, 1024) 2-dimentional parity code similar to a rectangular code. For simplicity, let us consider a block of four 4-bit words, d_{0-4}. Figure 4.4(a) shows a rectangular code where parity bits are calculated over each word (horizontal parity) and each bit-slice (vertical parity). A single error in the block will cause one horizontal and one vertical parity error which will indicate the location of the error. As mentioned earlier, calculating the horizontal parities in software is not as fast as calculating the vertical parities. Therefore, in Scheme 3, the horizontal parity is replaced with diagonal parity (similar to the scheme in [21]) which is essentially the same but translates to a more efficient software implementation (Fig. 4.4.(b)). The block size in our implementation is 32 words, because of 32 bits in each word (hence a (n=k+64, k=32×32) code).

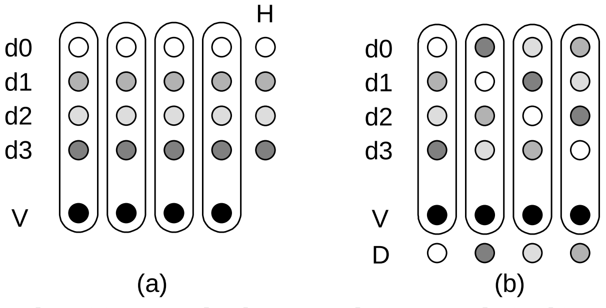

Figure 4.4 Parity codes: (a) vertical + horizontal (rectangular); (b) vertical + diagonal.

4) Scheme 4 uses a (66, 64) Reed-Solomon (RS) code in $GF(2^{32})$. The polynomial used for this code is: $P(X) = X^{32} + X^{22} + X^2 + X + 1$. The equations for the check-bit words are: $c_0 = \sum d_i, c_1 = \sum d_i\alpha, c_2 = \sum d_i\alpha^2$, where α is the field generator (the $\sum$ translates to the bitwise `xor` operation in a C program). c_0 is simply the vertical parity. c_1 and c_2 are calculated by a software implementation of a Multiple-Input Signature Register (MISR) [20]. The efficiency of software implementation of this scheme is similar to Schemes 2 and 3. With c_0 and c_1, the distance of the code (*d*) is 3 and a single word error can be corrected (SbEC). With c_0, c_1 and c_2, the distance is 4 and in addition to SbEC, double word errors are also detected (DbED). However, this extra coverage will be at the expense of a larger EDAC code and longer execution time. The block size for this code can be up to $2^{32}-1$ words, including the check-bit words. Therefore, this code can have a very low check-bit overhead. However, the probability of multiple errors increases as the block becomes larger. We keep the block size for this scheme at 64 words; the same as those of schemes 1 and 2.

4.3 Overhead Comparison

We implemented the four schemes described in the previous section in software and measured their performance on a 200MHz UltraSPARC-I microprocessor. Table 4.1 shows the results. Column 2 shows the size of the code segment of the program that does the encoding and the error detection and correction. Column 3 shows the overhead of the check bits. Notice that for Scheme 4, the block size can be larger and the overhead can be reduced as long as the probability of multiple errors in the block remains below the specifications. The decoding (error detection) speed mainly determines the performance overhead of each scheme because decoding is done more often than encoding or correction. The decoding speed (*DS*) of each scheme in terms of megabytes per second is shown in column 4. Given the size of memory that is being protected (S_{mem}) and the scrubbing interval (T_{scrub}), performance overhead (OH_{perf}) can be calculated using the following formula:

$$OH_{perf} = \frac{S_{mem}/DS}{T_{scrub} - S_{mem}/DS}.$$

Column 5 summarizes the error detection and correction capability of each scheme.

Table 4.1 Comparison of program size, check-bit overhead and decoding (error detection) speed of the four coding schemes.

Scheme	Program Size (bytes)	Check-bit Overhead = check-bit/data (words)	Decoding Speed (MB/s)	Detection/Correction Capability
Hamming	14,307	8/64=12.5%	187.80	bit-slice SEC-DED per block
Cyclic	6,731	8/64=12.5%	29.24	bit-slice SEC-DED per block
Parity	6,747	2/32=6.25%	34.68	SEC-DED per block
RS (d=3)	6,723	2/64=3.125%	24.41	SbEC per block

Notice that column 2 shows only the size of the core part of the EDAC program. There are other parts of the program that: maintain the list of memory segments that are scrubbed, implement the interleaving technique (discussed in Sec. 5.3), communicate with other programs, etc. The size of these parts, which is not included in column 2, depends on the features of the EDAC program and is the same for all the coding schemes. In our implementation, these parts were about 15,000 bytes in size. The differences in the core size are small compared to the size of the whole EDAC program. Therefore, when comparing the coding schemes, the core program size is a minor factor.

Scheme 1 has the highest decoding speed but also has the largest program size. The latter is a minor disadvantage as discussed in the previous paragraph. Scheme 2 has the same check-bit overhead and detection/correction capability as Scheme 1, but has a much lower decoding speed (this speed may be acceptable depending on the application).

Schemes 3 and 4 have lower check-bit overhead at the expense of less detection/correction capability.

There are many other EDAC codes and the proper code has to be chosen depending on application specifications. A scheme that has smaller program size, lower check-bit overhead and higher decoding speed is the preferred one. The last decision factor is the capability of the codes in handling multiple errors.

5. MULTIPLE ERROR CORRECTION

Multiple errors occur in two ways. Multiple SEUs can occur before the memory is scrubbed for errors, or a single SEU causes *multiple-bit upsets* (MBUs). In the former case, the scrubbing frequency needs to be adjusted according to the SEU rate to avoid exceeding the correction capability of the utilized EDAC code with a high level of confidence. The latter case has to be approached differently.

The probability of bit-flips in memories is increasing in new integrated circuit (IC) technologies due to reduction of feature sizes and lowered supply voltage, which make the stored charge in a cell comparable to the energy of a single particle. The reduced proximity between cells aggravates the problem. It has been observed that a single particle can affect multiple adjacent memory cells and cause multiple bit-flips [22] [23] [24] [25]. MBUs occurred in 1-5% of SEUs in a set of satellite experiments [26]. The fact that these multiple errors correspond to memory cells that are physically adjacent should be considered when designing an EDAC scheme. If the design is such that the physically adjacent bits belong to separate codewords, these errors can be corrected. To achieve this, the designer of the EDAC scheme needs to know the mapping of physical bits of the memory structure, to the logical bits in memory address space (location of the bits in a programmer's view of the memory). This mapping is determined by the system-level structure and the chip-level structure. We look at each of these separately.

5.1 System-Level Structure

Let us consider a system with 2MB of memory and a 32-bit data bus. Each memory chip that is used to build this memory can have 1, 4 or 8 data outputs, usually denoted as a ×1, ×4 or ×8 chip, respectively. For example, if 512K×1 chips are used, each chip will provide one data bit of the bus and 32 chips will make 2MB of memory. If 512K×8 chips are used, each chip will provide 8 data bits of the bus and four chips are enough to make 2MB of memory. In systems with hardware EDAC, the ×1 chips have the advantage that if one whole chip becomes faulty, a SEC-DED code can compensate for this failure. To tolerate chip failures of the wider chips, more advanced EDAC designs have to be used — these codes are beyond the scope of this paper; for a good discussion of this subject the reader is referred to [17].

The errors caused by SEUs are independent in each memory chip. This fact can be used when designing EDAC for chips with a specified output width. For example, if ×8 chips are used with Scheme 4, it will be beneficial to implement the Reed-Solomon code in $GF(2^8)$ and have the check-bits over each 8-bit byte portion of a 32-bit word (byte-slices as shown in Fig. 5.1). This code will be capable of correcting multiple errors that do not necessarily align in one word of the address space. Therefore, the fact that multiple dependent errors (caused by one SEU) do not cross the byte borders, can be used to enhance to capability of the code in correcting multiple independent errors. Notice that this is achieved with the same check-bit overhead but a more complicated code for doing the encoding and decoding. In addition, the size of the block can now be increased only up to $2^8 - 1$ words.

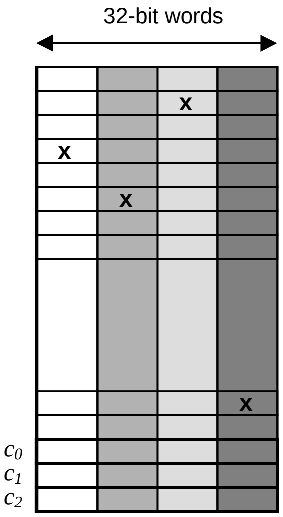

Figure 5.1 A byte-slice implementation of an EDAC code. An example of a multiple error that can be corrected is shown with the marked bytes.

Similarly, the vertical code shown in Fig. 4.2 can handle multiple independent errors caused by multiple SEUs in different ×1 chips. With ×1 chips, multiple errors that are caused by a single SEU (MBUs) will all be in one bit-slice (not necessarily in consecutive word addresses). A horizontal code that is capable of SEC-DED for each 32-bit word can easily handle these MBUs. However, a vertical code will fail if these errors map to the words of the same block. This mapping depends on the internal structure of the memory chip. Even with wider chips such as ×4 or ×8 chips, one needs to look at the structure inside the memory chips to know where these physically adjacent errors will be in the logical memory address space.

5.2 Chip-Level Structure

In this section, several possible implementations of a 512K×8 memory chip are analyzed. Figure 5.2 shows three different implementations of such a memory taken from the data sheets of Cypress Semiconductor Corporation [27].

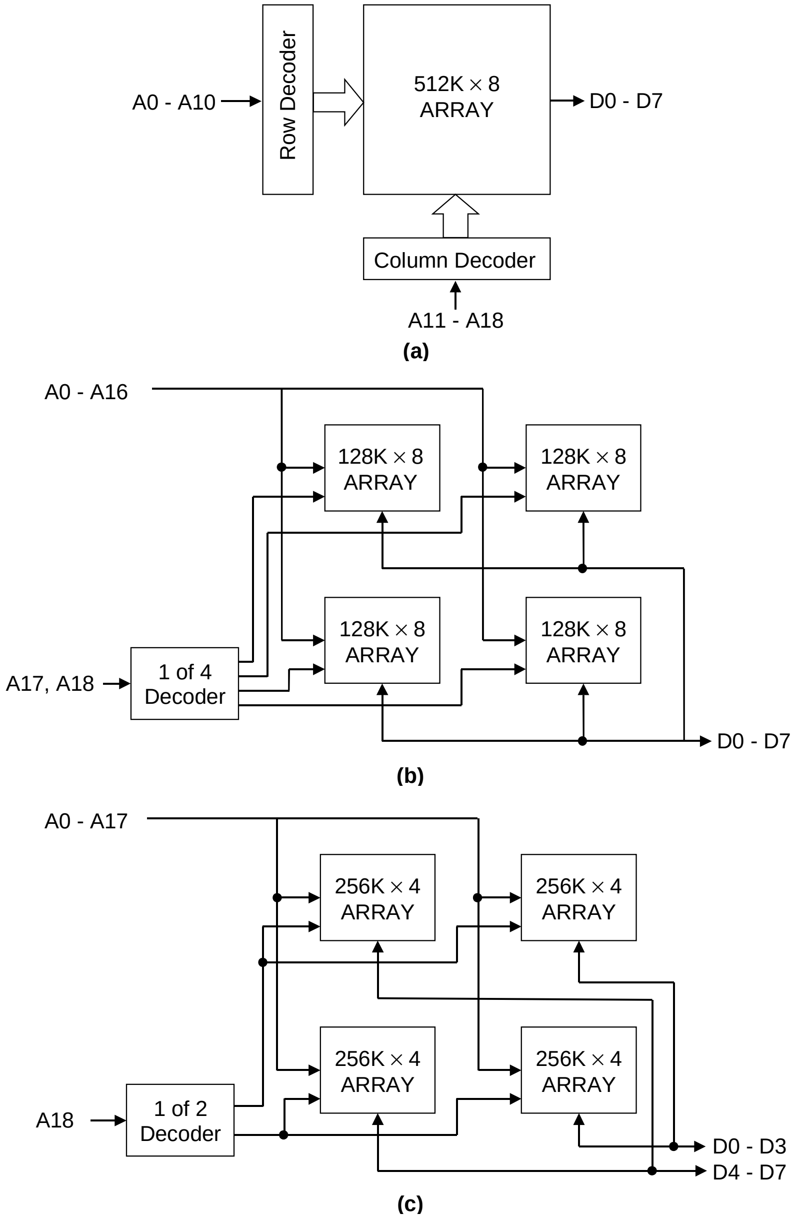

Figure 5.2 Three different implementation of a 512K×8 memory chip: (a) one ×8 array (Cypress CY62128), (b) four ×8 arrays (CYM1465), (c) four ×4 arrays (CYM1464).

With the structure in Fig. 5.2(b), errors in the four arrays will be independent. If the structure in Fig. 5.2(c) is used, then errors in each nibble (4 bits) of a word will be independent of errors in the other nibbles. An EDAC design can take advantage of this fact and enhance its correction capability in the same way as discussed for the example of Fig. 5.1.

One important thing that the data sheets do not show is the mapping (physical connection) of external address bits to internal address bits. For example, it is not necessarily the case that in Fig. 5.2(a), address bits A0 to A10 are connected to the row decoder and in that order.

To completely derive the physical to logical mapping of the bits inside a memory chip, we looked at the actual physical implementation of the four arrays in Fig. 5.2(b). Each 128K×8 module is divided into 8 subarrays (groups). Each subarray has 1024 rows and 128 columns — not counting the redundant rows and columns that are used for yield enhancement (defect tolerance). Let us indicate the address bits connected to the group, row and column decoders with AG, AR and AC, respectively. The mapping of external address bits (A0-A16) to these bits is shown in Table 5.1.

Table 5.1 Mapping of external to internal address bits in the 128K×8 array.

Internal Address Bits	External Address Bits
AG 0,1,2	A 15,16,10
AR 0-9	A 4,5,6,7,8,9,11,12,13,14
AC 0-3	A 0,1,2,3

The order of the data bits that come out of each group also differs for each group. If we look at a small portion of one subarray, it will look like Fig. 5.3(b). Bit 2 and bit 6 are physically adjacent. The number in each cell corresponds to the logical address of the word that contains that bit. This correspondence is illustrated in Fig. 5.3(a) where the same bits are numbered in a logical view of the memory. Let us consider bit 2 of address 18. This bit is physically adjacent to bit 2 of addresses 01, 02, 03, 17, 19, 33, 34 and 35 (we refer to this as *type 1 adjacency*)— if the geometries are small enough, we may have to consider adjacency with a larger radius [25]. For a more interesting example, consider bit 6 of address 16. This bit is physically adjacent with bit 6 of addresses 00, 01, 17, 32 and 33 (type 1), and with bit 2 of addresses 15, 31 and 47 (we refer to this as *type 2 adjacency*); which is something not quite expected. Adjacencies of type 2 are in different bit-slices and vertical codes can correct MBUs of this type. However, type 1 adjacencies are in the same bit-slice and vertical codes may fail to correct the corresponding MBUs. To handle type 1 adjacencies with a vertical code, a technique called *interleaving* can be used (Sec. 5.3).

Notice that a horizontal code can correct the MBUs corresponding to both types of adjacencies. In other words, the internal structure of some memories (like the above example) is such that hardware EDAC works well for all MBUs. However, this is not always true. For example, the internal structure of a ×8 memory chip from Texas Instruments is such that MBUs can occur within individual words [28]. Such *single-word multiple-bit upsets* (SMUs) will defeat a SEC-DED horizontal code. Therefore, in this case, a well-designed software EDAC can be more effective than a hardware EDAC.

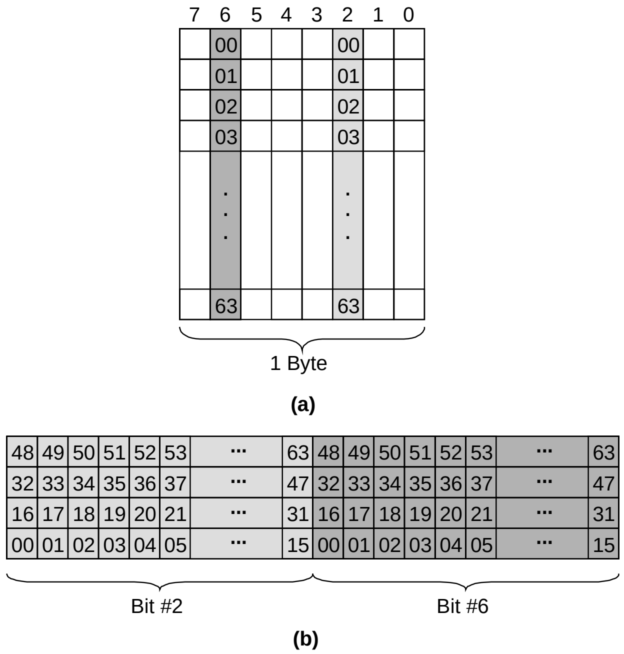

Figure 5.3 Bit positions for a small portion of the memory array of Fig. 5.2(b): (a) logical positions, (b) physical positions.

5.3 Interleaving

It was illustrated in the previous section that multiple errors can occur in one bit-slice of a block of words protected by a vertical EDAC code. If a SEC-DED code is used, these errors cannot be corrected. One solution is to use a code that can correct more errors in a codeword. However, codes with higher correction capability have higher check-bit, performance and program size overhead. Another solution is to logically separate the adjacent bits so that each error occurs in a different codeword. This can be done by *interleaving* the words that belong to the protected blocks. Interleaving is a technique where logically adjacent bits are mapped to bits of the communication channel or storage media that are not physically adjacent. This technique is used for handling burst errors. For example, audio CDs employ the *Cross-Interleaved Reed-Solomon Code* (CIRC) to overcome burst errors due to scratches and dust particles. CD-ROMs use a two-dimensional version of CIRC. Figure 5.4 shows a 4-way interleaved EDAC scheme that has 64 data words and 8 check-bit words. Starting from address 0, the words of a protected block belong to memory addresses 0, 4, 8, 12, 16,…, 252. Looking at Fig. 5.3(b), we see that having address 0 and 16 in the same block is not desirable. Any *i-way* interleaving scheme, where *i* is of the form $i = 2^k$ (a power of 2), $i = 2^k - 1$ or $i = 2^k + 1$, has the same problem. Therefore, when choosing an interleaving factor, it is best to avoid these numbers. By doing so, the scheme will be independent of the internal structure of the memory chips because for any internal structure, the adjacencies will have a relation that has these three forms (with different *k*'s).

The geometrical model for multiple upsets presented in [25], assumes that a memory cell will be upset if an ion comes within a distance *R* from its assumed center. We refer to this distance as the *sensitive radius of adjacency*. Notice that if *R* is larger than one, the interleaving factor has to be chosen more carefully. In our project, we assumed this radius is one and we used $i = 6$ for our application.

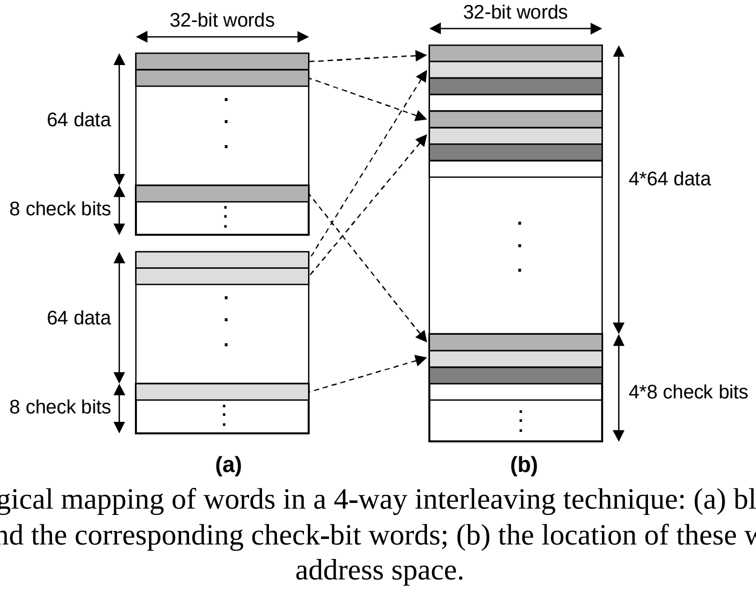

Figure 5.4 Logical mapping of words in a 4-way interleaving technique: (a) blocks of EDAC protected data and the corresponding check-bit words; (b) the location of these words in memory address space.

6. IMPLEMENTATION

We assume that the target system has a multi-tasking OS. As mentioned earlier, the EDAC program is an independent task that is executed periodically. Timers can be used to wake up the EDAC task periodically. The task should also have higher priority than normal programs so that it is executed at its fixed frequency, independent of the load on the system. Because this task has high priority, it will run to completion (one sweep of memory) before normal programs are resumed.

The EDAC program needs to access the data and code segments of other tasks. Direct access to the address space of another task is not always granted to a normal task. The operating system in our system is VxWorks with a flat address space and no protection option activated. However, in many operating systems, for example, Unix, the address space of a task is protected from being accessed by other tasks using hardware and software mechanisms. Only the operating system has unrestricted access to the whole memory. Therefore, in this case, the EDAC program has to be run at kernel level or given proper access rights.

We used APIs to interface application programs to the EDAC program. An example set of APIs is shown in Table 6.1. Since the EDAC program is a separate task, the function parameters are sent to it through message passing. The first time each application program is loaded into memory, it sends the address of its first and last instruction to the EDAC program using the `EDAC_add_block` function. Using the same mechanism, a program can also ask for protection of a data segment. The read and write functions are used for data segments.

Table 6.1 An example set of APIs for software EDAC.

Function Name and Parameters	Description
EDAC_add_block(*StartAddr*, *EndAddr*)	Add the block between *'StartAddr'* and *'EndAddr'* to the list of blocks to be scrubbed periodically.
EDAC_delete_block(*StartAddr*)	Delete the protected block that starts at *'StartAddr'*.
EDAC_read(*ReadAddr*, *Size*, &*Buffer*)	Read *'Size'* words into *'Buffer'* starting at *'ReadAddr'* from the corresponding protected block. The data are checked for errors before copying into *'Buffer'*.
EDAC_write(*WriteAddr*, *Size*, *Buffer*)	Write *'Size'* words from *'Buffer'* to locations starting at *'WriteAddr'*. New check bits are calculated for the corresponding protected block.

Almost all modern microprocessors use caches to compensate for the slow access to the main memory. In a split cache architecture, the data and instruction caches are separate. When the EDAC program checks the code segment of another program for errors, it reads the instructions of that program. These instructions go through the data cache because they are data for the EDAC program. If any correction is done on these instructions, the correction is written into the data cache. Therefore, the EDAC program should invalidate the instruction cache (if the corrected address exists in cache) and flush the data cache after a correction is done. This will force the correct instruction to be fetched from memory and into the instruction cache next time that address is accessed.

During a normal sweep of the memory by the EDAC program (no errors detected), all the checked addresses are accessed only once. Therefore, there is no benefit in caching these addresses. Moreover, they will replace all the active lines of the caches which will degrade the performance of the system by causing many cache misses after the EDAC program finishes one scrub operation. Therefore, it is better to treat the data accesses of the EDAC program as non-cacheable addresses so that they do not pollute the data cache. Notice that, even in this case, the cache has to be invalidated if a correction is done on an address that exists in the cache.

The EDAC program resides in memory and therefore it is vulnerable to errors itself. Read-only memories (ROMs) are less susceptible to SEUs hence, running the EDAC program out of ROM is one way of protecting it against bit-flips in its code segment. However, ROMs are not immune to SEUs and they are slower than RAMs. In some cases (for example, our ARGOS project) adding EDAC may be an after-thought in project design, so it is not possible to put the EDAC program in ROM. Notice that SEUs that occur in the processor can also result in miscalculations in the EDAC program. Therefore, in any case, some sort of redundancy is needed to ensure the correctness of this process. Time-redundancy (multiple executions) can be used to check for SEUs that occur in the processor. However, if an SEU corrupts the code segment of the EDAC program, it needs to be corrected so that it does not produce a wrong result repeatedly. This code segment can be protected by EDAC, similar to other programs that are being protected. However, a corrupt code cannot be trusted to correct itself. Therefore, a second copy of the EDAC program should exist. Each copy can do checking and correction on the other one (cross-checking). Another possibility is to have a second copy in ROM, and correct the errors simply by copying the image. At any time, there should be a healthy copy of the EDAC program that can be trusted to correct a possibly corrupted one.

For the ARGOS project, we use the two copies scheme with cross-checking. This scheme is currently running on the satellite and has proved to be effective in enhancing the availability of the system. Our experiment is being carried on at the time of this writing and we do not have enough data to calculate the improvement in availability. However, here are two examples where we have observed the effects.

After a system reset, we upload some programs and run them. Then, a few days later, we upload some new programs. Without software EDAC, the second uploads fail, sometimes causing exceptions or system reset. We attribute the failure to the accumulated bit-flips in the OS code that handles the uploading and linking, and in the global symbol table — the global symbol table holds the addresses of global variables and functions and is used in the linking process. When the EDAC program is uploaded and run with the first set of programs (shortly after a reset), the bit-flips in code segments of the OS are scrubbed periodically. In this case, most of the second uploads are successful — the failures can be due to bit-flips in the global symbol table (which is a data segment and is not protected against SEUs), or bit-flips in the OS code that have occurred since the last scrub operation.

Another issue is the time it takes for the system to halt after a fresh start (total reset). Without software EDAC, the errors accumulate in the code segments of the OS and our programs; after some time, the system gets an exception, for example, due to an illegal instruction (caused by a bit-flip in an instruction). This is a code that was running correctly and stops because of a transient error in hardware and not a software bug. When software EDAC is added, the frequency of these errors is significantly reduced and the system can operate correctly for a longer period before it halts.

The observations explained above show that, in the absence of hardware EDAC, system availability can be improved by software EDAC. In the next section, we quantify the reliability obtained by software EDAC for programs running in SEU prone environments. As a reference for comparison, reliability estimates are derived for programs running with both no EDAC and hardware EDAC support. We also quantify the sensitivity of program reliability to scrubbing interval.

7. SCRUBBING INTERVAL AND RELIABILITY ANALYSIS

Several papers [29-32] present reliability analysis for memory systems using hardware EDAC and scrubbing. Building on this prior work, the analysis presented in this section provides a framework for comparing hardware and software EDAC methods from the standpoint of program reliability. The environment assumed for this analysis closely matches the environment for the ARGOS experiment.

Any program alternates between two states: run and dormant. In the *run state*, the program instructions are fetched and executed. In the *dormant state*, the program instructions are resident in memory and the program waits in this state until it is scheduled by the OS to the run state. With EDAC and scrubbing, the program will have an additional scrub state. During the *scrub state*, the program instructions are read and rewritten (upon detection of correctable errors) with corrected data using hardware or software EDAC methods. The time between two successive scrub states is the *scrubbing interval*. By definition, the scrubbing interval will be the sum of run, dormant and scrub state times. The lifetime of a program, with EDAC and scrubbing, will be a renewal event comprising a repetition of the sequence of run, dormant, and scrub states. Table 7.1 lists the parameters used in the reliability analysis.

Table 7.1 Program environment parameter definitions and typical values.

Parameter	Description
u	Upset rate (probability of single-bit upset in a cycle). The units are upset/bit-cycle. Typical value assumed is 5.52 x 10^{-19}/bit-cycle. This is derived from 10 upsets/Mbyte-day using a clock rate of 25 MHz.
T_r	Number of cycles the program is in the run state. Typical value assumed is 10^9 cycles.
T_d	Number of cycles the program is in the dormant state. Typical value assumed is 6.5 x 10^9 cycles.
T_s	Number of cycles it takes to scrub the program data. For hardware EDAC, the typical value is 1.25x10^5 cycles. For software EDAC, the typical value is 2.5 x 10^7. The scrubbing interval is $T_r+T_d+T_s$ (5 minutes in 25MHz clock rate cycles).
n	Horizontal bit width of program data. Typical values assumed are n=32 for no EDAC or software EDAC and n=39 for hardware EDAC.
m	Number of words in a block of vertical code (for software EDAC). Typical value assumed is 72 (64 data words + 8 check-bit words).
S	Number of program words protected by EDAC. Typical value assumed is 131,072 (program size = 0.5 Mbyte).
S'	S plus number of check-bit words needed for S ($S' = S$ x 72/64).

Let us begin by estimating the reliability of a program running with no EDAC protection. Without EDAC protection, the lifetime of a program is a renewal event comprising repetition of the sequence of run and dormant states. The probability that a program will survive one sequence of run and dormant states is given by:

$$(1-u)^{nS(T_r+T_d)}.$$

Using the values in Table 7.1, the reliability (probability that a program will survive multiple sequences of run and dormant states) as a function of time (in minutes) is given in Table 7.2.

Table 7.2 Survival probability of a program with no EDAC support.

Time (in Minutes)	Reliability
10	0.97
20	0.93
30	0.90
40	0.87
1 day	0.0067

Table 7.2 shows that without EDAC support, the probability of a program surviving even for a day is very small. Therefore, the necessity of protecting programs with EDAC is demonstrated for these assumptions. With SEC-DED hardware EDAC support, the probability that a program survives one sequence of run, dormant, and scrub states is the same as the probability that there is no more than a single-bit error in any program word (horizontal codeword). Simple combinatorial analysis shows this probability as:

$$(n(1-u)^{(T_r+T_d+T_S)(n-1)} - (n-1)(1-u)^{n(T_r+T_d+T_S)})^S \, .$$

With software EDAC support, the program is unprotected against SEUs during the run state. Therefore for a program to survive, with a software SEC-DED EDAC support, the program must not encounter any error during the run state and no more than a single-bit error in any vertical codeword during the dormant and scrub states. For the sake of simplicity in analysis, we assume that the EDAC program is error-free. We also assume that due to physical locality of memory references, only 10% of the program code is used in each scrubbing interval. Using simple combinatorial analysis, the following expression quantifies the reliability of a program for one sequence of run, dormant, and scrub states:

$$(1-u)^{n\frac{S}{10}T_r}(m(1-u)^{(T_d+T_S)(m-1)} - (m-1)(1-u)^{m(T_d+T_S)})^{(nS'/m)} \, .$$

Again using the values in Table 7.1, Table 7.3 shows the reliability values for a program with software EDAC and hardware EDAC. This table shows that with software EDAC, the reliability of a program is several orders of magnitude better than a program without any EDAC protection. Reliability of a program with no EDAC, hardware EDAC, and software EDAC is shown in Fig. 7.1 for a period of 48 hours. Note that the time axis is the sum of T_r and T_d and therefore the graph shows the reliability comparison for equal amount of work. Since T_s is less than one percent of $T_r + T_d$ in both hardware and software EDAC, total times ($T_r + T_d + T_s$) are almost the same.

Table 7.3 Software and hardware EDAC reliability comparison.

Time (in Days)	Reliability with Software EDAC	Reliability with Hardware EDAC
1	0.9355	~1.0
2	0.8752	0.999999
3	0.8187	0.999999
4	0.7659	0.999998

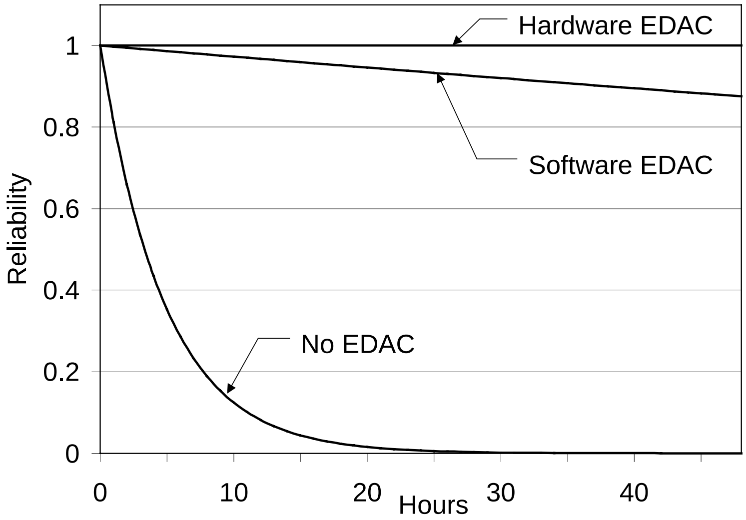

Figure 7.1 Reliability comparison using the parameters in Table 7.1.

The sensitivity of program reliability to the scrubbing interval and the upset rate is shown in Table 7.4. This analysis demonstrates that for low-radiation environments, the scrubbing interval can be increased (thereby reducing the performance overhead) without appreciably affecting reliability.

Table 7.4 Software and hardware EDAC reliability sensitivity to scrubbing interval.

Scrubbing Interval	Reliability for a Period of One Day			
	Software EDAC		Hardware EDAC	
	$u = 5.52 \times 10^{-19}$	$u = 5.52 \times 10^{-18}$	$u = 5.52 \times 10^{-19}$	$u = 5.52 \times 10^{-18}$
10 minutes	0.935506	0.513345	0.999999	0.999904
20 minutes	0.935504	0.513274	0.999998	0.999808
30 minutes	0.935503	0.513202	0.999997	0.999712
40 minutes	0.935502	0.513130	0.999996	0.999617
1 day	0.935319	0.503198	0.999862	0.986297

8. DISCUSSION

Solid-state memories, such as RAMs, are used for the main memory, secondary storage, or processor caches in a computer system. In this paper, we present a software-implemented EDAC technique for protecting memories. Let us consider EDAC protection for main memory. With software EDAC, the data that is read from main memory may be erroneous, if the error occurs after the last scrub operation and before the time of reading. In other words, single-bit errors may cause failures. In contrast, hardware EDAC checks all the data that is read from memory, and corrects single-bit errors. Therefore, hardware EDAC provides better reliability and, when possible, should be the first choice for protecting the main memory. When hardware EDAC is not available or affordable, software EDAC can be used as a low-cost solution for enhancing the reliability of systems.

For cases where data is read and written in blocks of words rather than individual words, software EDAC may be a better choice than hardware EDAC. For example, when solid-state memories are used for secondary storage (such as in satellites), the processor can access the data through a buffer in main memory rather than directly from the secondary storage (in this case, there is no need to restrict ourselves to systematic codes).

For this secondary storage memory, EDAC protection can be provided in software by periodic scrubbing and APIs (as explained in Sec. 3.2). All read operations are checked for errors, and single-bit errors in this memory will not cause failures. Therefore, assuming that the execution of the EDAC program is error-free, software EDAC can provide the same reliability as hardware EDAC if they use the same coding scheme. Considering the flexibility of software EDAC, it is possible to implement more capable coding schemes that are infeasible in hardware, and thereby provide better reliability through software. Moreover, as mentioned in Sec. 5.2, there are cases of MBUs where hardware EDAC will fail but software EDAC is able to correct the errors. Therefore, software EDAC could be a better choice for this application.

An extension of this work can be studying the feasibility of implementing software-implemented EDAC for cache memories.

ACKNOWLEDGMENTS

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047. The authors wish to thank Dr. Bella Bose for his valuable suggestions, SunilKumar Koduru from Cypress Semiconductor for the information on Cypress memory chips, and also Samy Makar, Santiago Fernandez-Gomez, Subhasish Mitra and Nahmsuk Oh for their helpful comments. The ARGOS experiment is a collaborative project with the Naval Research Laboratory (NRL) USA experiment group lead by Dr. Kent Wood as the principal investigator.

REFERENCES

[1] C.L. Chen and M.Y. Hsiao, "Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review," *IBM J. Res. Develop.*, Vol. 28, pp. 124-134, Mar. 1984.

[2] D.P. Siewiorek and R.S. Swarz, *Reliable Computer Systems*, Burlington, Digital Press, 1992.

[3] R. Koga and W.A. Kolasinski, "Heavy Ion-Induced Single Event Upsets of Microcircuits: A Summary of the Aerospace Corporation Test Data," *IEEE Trans. on Nuclear Science*, Vol. 31, No. 6, pp. 1190-1195, Dec. 1984.

[4] E. Worley, R. Williams, A. Waskiewicz, and J. Groninger, "Experimental and Simulation Study of the Effects of Cosmic Particles on CMOS/SOS RAMs," *IEEE Trans. on Nuclear Science*, Vol. 37, No. 6, pp. 1855-1860, Dec. 1990.

[5] T.R.N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*, Prentice Hall, 1989.

[6] P.P. Shirvani and E.J. McCluskey, "Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project, " *CRC-TR 98-2*, Stanford University, December 1998.

[7] K.S. Wood, et al., "The USA Experiment on the ARGOS Satellite: A Low Cost Instrument for Timing X-Ray Binaries," Published in *EUV, X-Ray, and Gamma-Ray Instrumentation for Astronomy V*, ed. O.H. Siegmund & J.V. Vellerga, SPIE Proc., Vol. 2280, pp. 19-30, 1994.

[8] J.F. Ziegler, et al., *IBM J. Res. Develop.*, Vol. 40, No. 1, (all articles), Jan. 1996.

[9] R.H. Paschburg, “Software Implementation of Error-Correcting Codes,” Univ. Illinois, Urbana, (AD-786 542), Aug. 1974.

[10] J.W. Whelan, “Error Correction with a Microprocessor,” *Proc. IEEE National Aerospace and Electronics Conf.*, pp. 1317-23, 1977.

[11] J.S. Whiting, “An Efficient Software Method for Implementing Polynomial Error Detection Codes,” *Computer Design*, Vol. 14, No. 3, pp. 73-7, Mar. 1975.

[12] D.V. Sarwate, “Computation of Cyclic Redundancy Checks via Table Look-up,” *Communications of the ACM*, Vol. 31, No. 8, pp. 1008-13, Aug. 1988.

[13] D.C. Feldmeier, “Fast Software Implementation of Error Detection Codes,” *IEEE/ACM Trans. Networking*, Vol. 3, No. 6, pp. 640-651, Dec. 1995.

[14] M.S. Hodgart, “Efficient Coding and Error Monitoring for Spacecraft Digital Memory,” *Int’l J. Electronics*, Vol. 73, No. 1, pp. 1-36, 1992.

[15] S.B. Wicker, *Error Control Systems for Digital Communications and Storage*, Englewood Cliffs, N.J., Prentice Hall, 1995.

[16] N.R. Saxena, et al., “Fault-Tolerant Features in the HaL Memory Management Unit,” *IEEE Trans. Comp.*, Vol. 44, No. 2, pp.170-179, February 1995.

[17] T.J. Dell, “A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory,” http://www.chips.ibm.com:80/products/memory/chipkill/chipkill.html, IBM Microelectronics Division, Rev. 11/19/97.

[18] N. Oh, P.P. Shirvani and E.J. McCluskey, "Error Detection by Duplicated Instruction in Superscalar Microprocessors," *CRC-TR*, in preparation.

[19] N. Oh, P.P. Shirvani and E.J. McCluskey, "Control-Flow Checking by Software Signatures," *CRC-TR*, in preparation.

[20] A.M. Patel and S.J. Hong, “Optimal Rectangular Code for High Density Magnetic Tapes,” *IBM J. Res. Develop.*, Vol. 18, pp. 579-88, November 1974.

[21] A.M. Patel, “Adaptive Cross-Parity (AXP) Code for a High-Density Magnetic Tape Subsystem,” *IBM J. Res. Develop.*, Vol. 29, pp. 546-62, November 1985.

[22] J.F. Ziegler, et al., “IBM Experiments in Soft Fails in Computer Electronics (1978-1994),” *IBM J. Res. Develop.*, Vol. 40, No. 1, pp. 4, Jan. 1996.

[23] J. Liu, et al., “Heavy Ion Induced Single Event Effects in Semiconductor Device,” *17th Int’l. Conf. on Atomic Collisions in Solids (in Nucl. Instrum. & Methods in Physics Res.)*, Sec. B, Vol. 135, No. 1-4, pp. 239-243, 1997.

[24] R. Reed, et al., “Heavy Ion and Proton-Induced Single Event Multiple Upset,” *IEEE Trans. Nucl. Sci.*, Vol. 44, No. 6, pp. 2224-9, July 1997.

[25] R. Hosken, et al., “Investigation of Non-Independent Single Event Upsets in the TAOS GVSC Static RAM,” *IEEE Radiation Effects Data Workshop*, pp. 53-60, July 1997.

[26] C.I. Underwood, “The Single-Event-Effect Behavior of Commercial-Off-The-Shelf Memory Devices – A Decade in Low-Earth Orbit,” *Proc. 4th European Conf. on Radiation and Its Effects on Components and Systems*, pp. 251-8, Palm Beach, Cannes, France, Sep. 1997.

[27] Data Sheets of Cypress Semiconductor Corp. SRAMs, http://www.cypress.com/design/datasheets/index.html, Cypress Semiconductor Corp., 1999.

[28] C.I. Underwood, et al., "Observation of Single-Event Upsets in Non-Hardened High-Density SRAMs in Sun Synchronous Orbit," *IEEE Trans. Nucl. Sci.*, Vol. 39, No. 6, Dec. 1992.

[29] J.A. Abraham, E.S. Davidson and J.H. Patel, "Memory System Design for Tolerating Single Event Upsets," *IEEE Trans. Nucl. Sci.*, Vol. 30, No. 6, pp. 4339-44, Dec. 1983.

[30] A.M. Saleh, et al., "Reliability of Scrubbing Recovery-Techniques for Memory Systems," *IEEE Trans. on Reliability*, Vol. 39, No. 1, pp. 114-22, April 1990.

[31] R.M. Goodman, et al., "The Reliability of Semiconductor RAM Memories with On-Chip Error-Correction Coding," *IEEE Trans. on Information Theory*, Vol. 37, No. 3, pp. 884-96, May 1991.

[32] G.-C. Yang, et al., "Reliability of Semiconductor RAMs with Soft-Error Scrubbing Techniques," *IEE Proc. – Computers and Digital Techniques*, Vol. 142, No. 5, pp. 337-44, Sept. 1995.

AUTHORS

Philip P. Shirvani; Center for Reliable Computing, Departments of Electrical Engineering and Computer Science, Stanford University, Room #236, MC9020, Gates Building 2A, 353 Serra Mall, Stanford, California 94305 USA.

Internet (e-mail): shirvani@crc.stanford.edu

Philip P. Shirvani received his BS degree in electrical engineering from Sharif University of Technology, Tehran, Iran, in 1991 and his MS degree in electrical engineering from Stanford University in 1996. He is currently a PhD candidate in electrical engineering at Stanford University and assistant director for the Stanford ARGOS project at Center for Reliable Computing, Stanford University. His research interests include computer architecture, fault-tolerant computing and VLSI design and test. He is a student member of the IEEE Computer Society.

Dr. Nirmal Saxena; Center for Reliable Computing, Departments of Electrical Engineering and Computer Science, Stanford University, Room #236, MC9020, Gates Building 2A, 353 Serra Mall, Stanford, California 94305 USA.

Internet (e-mail): saxena@crc.stanford.edu

Nirmal R. Saxena received the BE degree in electronics and communication engineering from Osmania University, India, in 1982; the MS degree in electrical engineering from the University of Iowa in 1984; and the PhD degree in electrical engineering from Stanford University in 1991. He is currently with Tiara Networks and is also a co-investigator for the ROAR project at Center for Reliable Computing, Stanford

University. His research interests include computer architecture, fault-tolerant computing, combinatorial mathematics, probability theory and VLSI design/test. He is a senior member of the IEEE.

Dr. Edward J. McCluskey; Center for Reliable Computing, Departments of Electrical Engineering and Computer Science, Stanford University, Room #236, MC9020, Gates Building 2A, 353 Serra Mall, Stanford, California 94305 USA.
Internet (e-mail): ejm@crc.stanford.edu

Edward J. McCluskey received the A.B. degree (summa cum laude, 1953) in mathematics and physics from Bowdoin College, and the B.S. (1953), M.S. (1953), and Sc.D. (1956) degrees in electrical engineering from M.I.T. The degree of Doctor Honoris Causa was awarded in 1994 by the Institut National Polytechnique de Grenoble.

He worked on electronic switching systems at the Bell Telephone Laboratories from 1955 to 1959. In 1959, he moved to Princeton University, where he was Professor of Electrical Engineering and Director of the University Computer Center. In 1966, he joined Stanford University, where he is Professor of Electrical Engineering and Computer Science, as well as Director of the Center for Reliable Computing. He founded the Stanford Digital Systems Laboratory (now the Computer Systems Laboratory) in 1969 and the Stanford Computer Engineering Program (now the Computer Science MS Degree Program) in 1970. The Stanford Computer Forum (an Industrial Affiliates Program) was started by Dr. McCluskey and two colleagues in 1970 and he was its Director until 1978.

McCluskey developed the first algorithm for designing combinational circuits — the Quine-McCluskey logic minimization procedure as a doctoral student at MIT. At

Bell Labs and Princeton, he developed the modern theory of transients (hazards) in logic networks and formulated the concept of operating modes of sequential circuits. His Stanford research focuses on logic testing, synthesis, design for testability, and fault-tolerant computing. Prof. McCluskey and his students at CRC worked out many key ideas for fault-equivalence, probabilistic modeling of logic networks, pseudo-exhaustive testing, and watchdog processors. He collaborated with Signetics researchers in developing one of the first practical multivalued logic implementations and then worked out a design technique for such circuitry.

Dr. McCluskey served as the first President of the IEEE Computer Society. He is the recipient of the 1996 IEEE Emanuel R. Piore Award. He is a Fellow of the IEEE, AAAS, and ACM; and a member of the NAE. He has published several books including two widely used texts.

Dr. McCluskey served as the first President of the IEEE Computer Society. His most recent honors include election to the National Academy of Engineering, 1998, the 1996 IEEE Emanuel R. Piore Award, IEEE Computer Society Golden Core Member. In 1984, he received the IEEE Centennial Medal and the IEEE Computer Society Technical Achievement Award in Testing. In 1990, he received the EURO ASIC 90 Prize for Fundamental Outstanding Contribution to Logic Synthesis. The IEEE Computer Society honored him with the 1991 Taylor L. Booth Education Award. He is a Fellow of the IEEE, AAAS, and ACM.

He has published several books and book chapters as well as hundreds of papers. His most recent book is Logic Design Principles with Emphasis on Testable Semicustom Circuits, published in 1986 by Prentice-Hall.

FIGURES

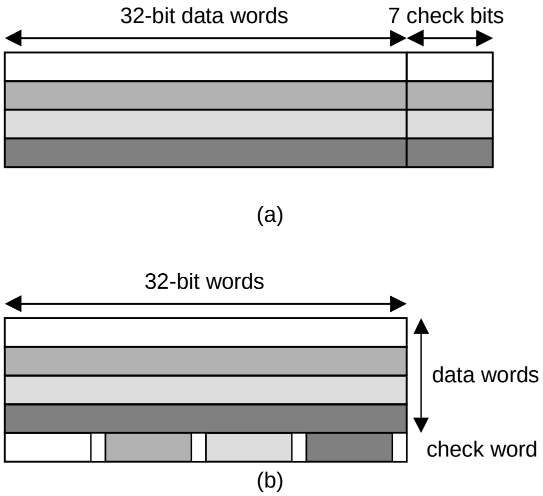

Figure 4.1 A horizontal code over bits of a word: (a) hardware implementation; (b) organization of bits when the code is implemented in software.

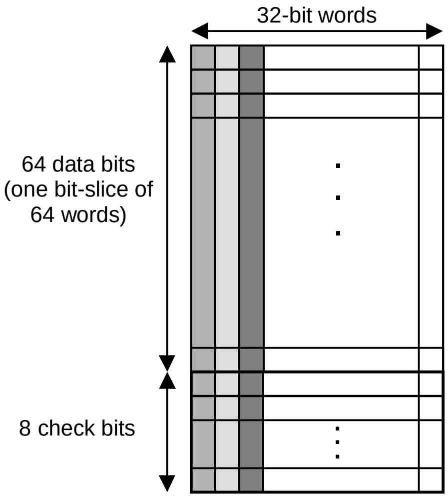

Figure 4.2 A vertical code over bit-slices of words.

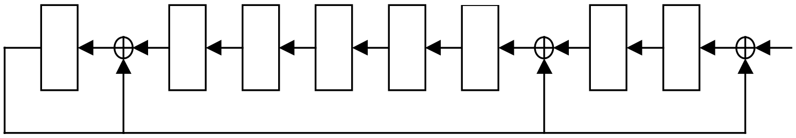

Figure 4.3 The LFSR corresponding to polynomial $P(X) = X^8 + X^7 + X^2 + 1$.

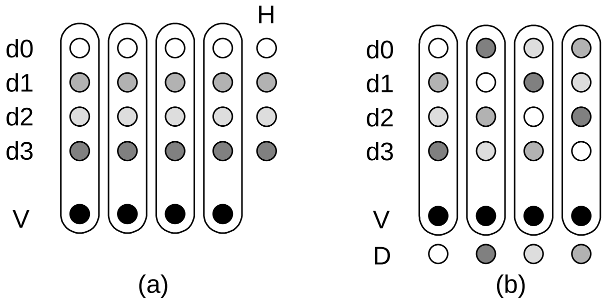

Figure 4.4 Parity codes: (a) vertical + horizontal (rectangular); (b) vertical + diagonal.

Table 4.1 Comparison of program size, check-bit overhead and decoding (error detection) speed of the four coding schemes.

Scheme	Program Size (bytes)	Check-bit Overhead = check-bit/data (words)	Decoding Speed (MB/s)	Detection/Correction Capability
Hamming	14,307	8/64=12.5%	187.80	bit-slice SEC-DED per block
Cyclic	6,731	8/64=12.5%	29.24	bit-slice SEC-DED per block
Parity	6,747	2/32=6.25%	34.68	SEC-DED per block
RS (d=3)	6,723	2/64=3.125%*	24.41	SbEC per block

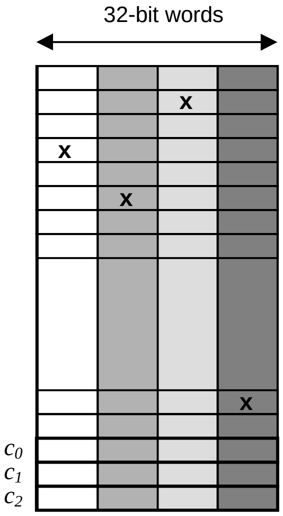

Figure 5.1 A byte-slice implementation of an EDAC code. An example of a multiple error that can be corrected is shown with the marked bytes.

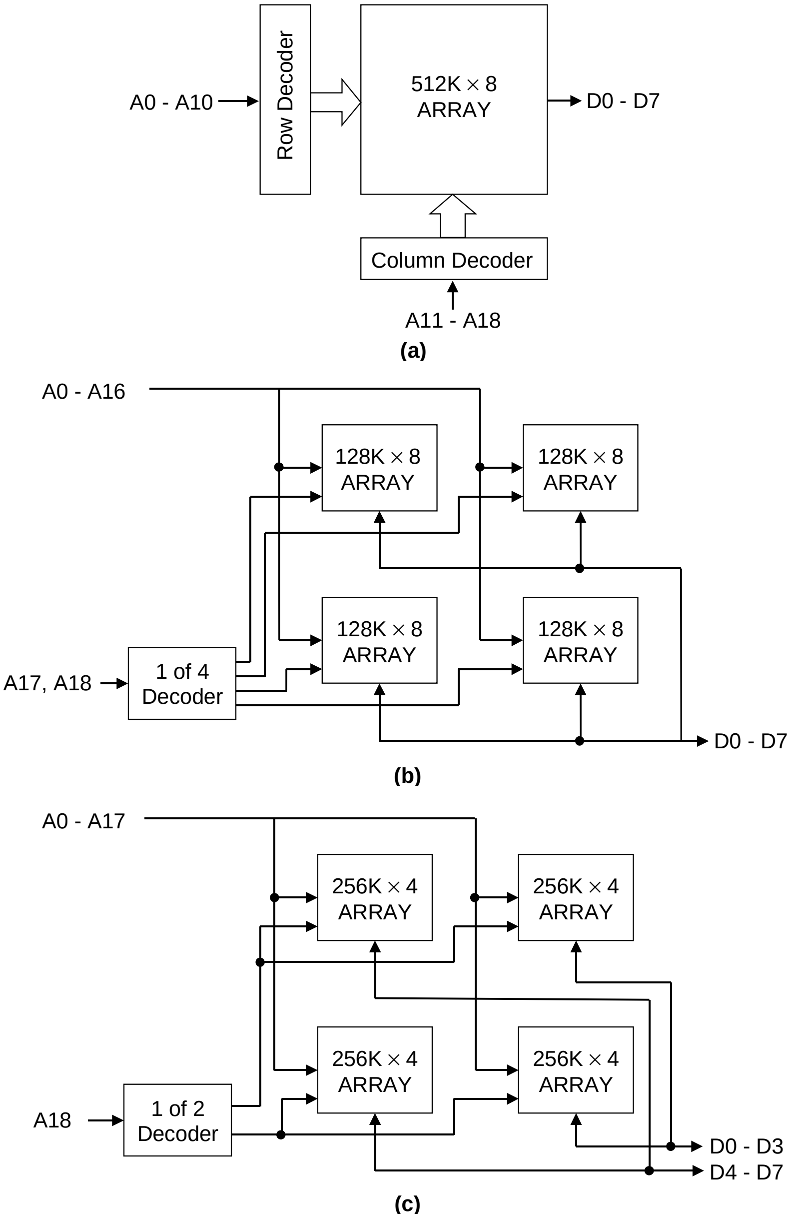

Figure 5.2 Three different implementation of a 512K×8 memory chip: (a) one ×8 array (Cypress CY62128), (b) four ×8 arrays (CYM1465), (c) four ×4 arrays (CYM1464).

Table 5.1 Mapping of external to internal address bits in the 128K×8 array.

Internal Address Bits	External Address Bits
AG 0,1,2	A 15,16,10
AR 0-9	A 4,5,6,7,8,9,11,12,13,14
AC 0-3	A 0,1,2,3

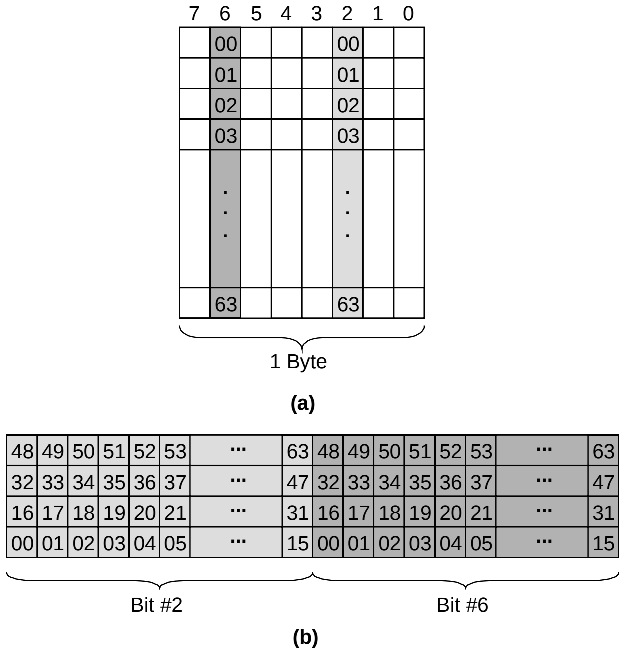

Figure 5.3 Bit positions for a small portion of the memory array of Fig. 5.2(b): (a) logical positions, (b) physical positions.

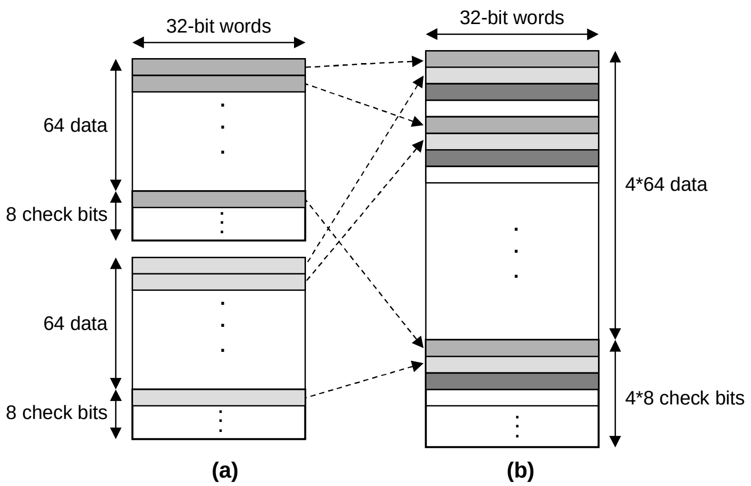

Figure 5.4 Logical mapping of words in a 4-way interleaving technique: (a) blocks of EDAC protected data and the corresponding check-bit words; (b) the location of these words in memory address space.

Table 8.1 An example set of APIs for software EDAC.

Function Name and Parameters	Description
EDAC_add_block(*StartAddr*, *EndAddr*)	Add the block between *'StartAddr'* and *'EndAddr'* to the list of blocks to be scrubbed periodically.
EDAC_delete_block(*StartAddr*)	Delete the protected block that starts at *'StartAddr'*.
EDAC_read(*ReadAddr*, *Size*, &*Buffer*)	Read *'Size'* words into *'Buffer'* starting at *'ReadAddr'* from the corresponding protected block. The data are checked for errors before copying into *'Buffer'*.
EDAC_write(*WriteAddr*, *Size*, *Buffer*)	Write *'Size'* words from *'Buffer'* to locations starting at *'WriteAddr'*. New check bits are calculated for the corresponding protected block.

Table 7.1 Program environment parameter definitions and typical values.

Parameter	Description
U	Upset rate (probability of single-bit upset in a cycle). The units are upset/bit-cycle. Typical value assumed is 5.52 x 10^{-19}/bit-cycle. This is derived from 10 upsets/Mbyte-day using a clock rate of 25 MHz.
T_r	Number of cycles the program is in the run state. Typical value assumed is 10^9 cycles.
T_d	Number of cycles the program is in the dormant state. Typical value assumed is 6.5 x 10^9 cycles.
T_s	Number of cycles it takes to scrub the program data. For hardware EDAC, the typical value is 1.25x10^5 cycles. For software EDAC, the typical value is 2.5 x 10^7. The scrubbing interval is T_r+T_d+T_S (5 minutes in 25MHz clock rate cycles).
N	Horizontal bit width of program data. Typical values assumed are n=32 for no EDAC or software EDAC and n=39 for hardware EDAC.
M	Number of words in a block of vertical code (for software EDAC). Typical value assumed is 72 (64 data words + 8 check-bit words).
S	Number of program words protected by EDAC. Typical value assumed is 131,072 (program size = 0.5 Mbyte).
S'	S plus number of check-bit words needed for S (S' = S x 72/64).

Table 7.2 Survival probability of a program with no EDAC support.

Time (in Minutes)	Reliability
10	0.97
20	0.93
30	0.90
40	0.87
1 day	0.0067

Table 7.3 Software and hardware EDAC reliability comparison.

Time (in Days)	Reliability with Software EDAC	Reliability with Hardware EDAC
1	0.9355	~1.0
2	0.8752	0.999999
3	0.8187	0.999999
4	0.7659	0.999998

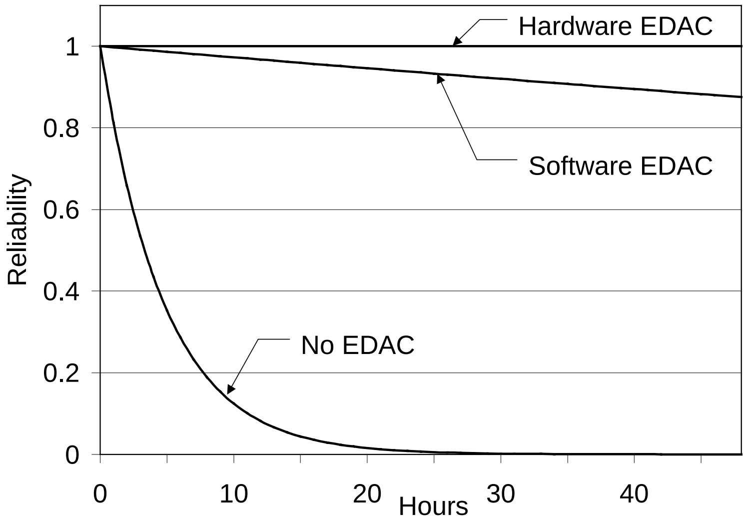

Figure 7.1 Reliability comparison using the parameters in Table 7.1.

Table 7.4 Software and hardware EDAC reliability sensitivity to scrubbing interval.

Scrubbing Interval	**Reliability for a Period of One Day**			
	Software EDAC		**Hardware EDAC**	
	U = 5.52 x 10^{-19}	u = 5.52 x 10^{-18}	u = 5.52 x 10^{-19}	u = 5.52 x 10^{-18}
10 minutes	0.935506	0.513345	0.999999	0.999904
20 minutes	0.935504	0.513274	0.999998	0.999808
30 minutes	0.935503	0.513202	0.999997	0.999712
40 minutes	0.935502	0.513130	0.999996	0.999617
1 day	0.935319	0.503198	0.999862	0.986297