System-Level Hardware-Based Protection of Memories against Soft-Errors

Valentin Gherman, Samuel Evain, Mickaël Cartron, Nathaniel Seymour, Yannick Bonhomme
CEA, LIST, Laboratoire de Fiabilisation des Systèmes Embarqués
PC 94, Gif-sur-Yvette, F-91191 France
firstname.lastname@cea.fr

Abstract

We present a hardware-based approach to improve the resilience of a computer system against the errors occurred in the main memory with the help of error detecting and correcting (EDAC) codes. Checksums are placed in the same type of memory locations and addressed in the same way as normal data. Consequently, the checksums are accessible from the exterior of the main memory just as normal data and this enables implicit fault-tolerance for interconnection and solid-state secondary storage sub-systems. A small hardware module is used to manage the sequential retrieval of checksums each time the integrity of the data accessed by the processor sub-system needs to be verified. The proposed approach has the following properties: (a) it is cost efficient since it can be used with simple storage and interconnection sub-systems that do not possess any inherent EDAC mechanism, (b) it allows on-line modifications of the memory protection levels, and (c) no modification of the application software is required.

1 Introduction

Soft errors in memory arrays are a well known reliability concern. Driven by the permanent quest for greater performance and lower power, the relentless scaling of integration density increases the soft error rates at chip-level [4, 7, 10, 11]. Fault-tolerance is a very effective approach used to improve the reliability of VLSI designs [3]. We distinguish between solutions at system and sub-system levels. At sub-system level, error detecting and/or correcting (EDAC) are the prevailing way to build fault tolerant memory sub-systems [5, 12]. In existing EDAC-based schemes, the number of internal links and storage cells is augmented to accommodate extra bits, and encoding and checking circuitry is added to ensure error detection/correction [6, 13, 16]. These solutions cannot be naturally extended to protect external links or secondary storage, unless these sub-systems are custom designs.

System-level solutions can ensure the error protection of sub-systems that do not possess any special fault tolerance attribute. In this way, the number of commercial off-the-shelf (COTS) components in a computing system can be increased and the overall cost reduced. System-level solutions based on software redundancy [14] or on software-implemented EDAC [15] enable the design of dependable computing systems using only COTS components. Unfortunately, performance penalty and code generation overhead in the former case [14], and the existence of a *window of vulnerability* with respect to the occurrence of soft errors in the latter case [15] may be prohibitive for a large spectrum of applications.

Here, a new system-level approach is proposed to ensure the error protection with the help of EDAC codes. Like in software-based approaches, the checksums are stored and handled just like the normal data. In this way, the checksums can be exchanged between the main memory and the rest of the system. Consequently, the EDAC operations can be executed very high in the system hierarchy to offer an implicit protection against the errors from the main memory, secondary storage and interconnections even if these sub-systems do not possess inherent soft error protection mechanisms.

A small hardware component, called reliability service module (RSM), is attached to each memory access port of the processor sub-system in order to (a) detect the integrity level associated with the exchanged data, and, in case of a protected memory access, (b) coordinate the sequential access to the protected data and their checksums, and (c) perform EDAC operations. If a memory access belongs to a non-critical application, the RSM is simply bypassed. Moreover, several EDAC codes can be enabled on-line [5, 12]. This flexibility may be employed to set-up cost efficient dependable systems that have to execute tasks with different criticality levels. The RSM-based approach is compatible with a component-based design which facilitates the reduction of development costs and time-to-market. Simulation results indicate a low performance impact of RSM on processors with cache memories.

2 RSM-based approach

Let us consider the interaction between a processor and the main memory in a computing system, with the processor in the master role and the main memory in the slave role. Without any loss of generality, we consider here only the case with the largest impact on system performance in which the memory words can only be accessed sequentially. As illustrated in Figure 1, an RSM is associated with a memory access port of the processor sub-system to ensure EDAC-based protection of the critical applications from the contamination with memory errors. The corresponding checksums are placed in the same type of memory locations and addressed in the same way as normal data. The silicon used to accommodate the checksums can be exploited to store normal data when non-critical applications are executed.

When a memory word is accessed, the RSM verifies if the memory address belongs to a protected zone and the checksum corresponding to the exchanged data is already present in the RSM. In case of a memory read access, these verifications are performed while the required data is delivered by the memory. If no protection is required, the RSM is simply bypassed without any performance penalty, except for the use of a multiplexer on the links between the processor and the memory. Otherwise, if the required checksum is not available inside the RSM an additional memory access is initiated to bring the memory word that contains this checksum. Then, the verified data is delivered to the processor or, upon error detection, an exception is raised. In case of a correctable error, the regenerated information can be restored into the memory to prevent error accumulation.

978-3-9810801-5-5/DATE09 © 2009 EDAA

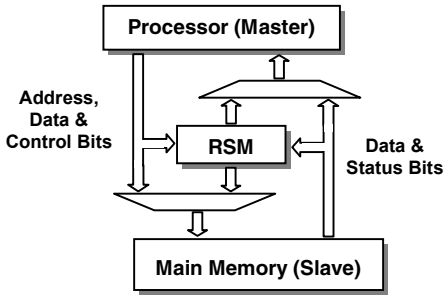

Fig. 1: RSM integration in a communication channel between processor and main memory.

During a memory write access, the data and its address reach the RSM and the slave in the same time. Subsequently, the corresponding checksum is stored into the memory.

In order to reduce storage and performance overheads, the checksums corresponding to data words stored at consecutive memory locations are packed together into a single *check word*. The number *C* of data words that correspond to a check word is always a power of 2 and the log_2C least significant bits of a data word address are used to encode the checksum position within a check word. For example, *C* is equal to 32 or 4 for 32-bit memories protected by parity and SEC-DED codes, respectively. *C* is doubled in case of 64-bit memories. In case of burst transfers, the number of additional memory accesses is minimized since on usual bus sub-systems the burst sizes are powers of 2 [1].

The address of a protected data word (*@DW*) is transformed into the address of the corresponding *check word* (*@CW*) as described below:

$$@CW = OffSet \vee \left[(Mask \wedge @DW) >> \log_2 C \right] \qquad (1)$$

The symbols '$\vee$' and '$\wedge$' represent bit-wise *or* and *and* logic operations, respectively. The operation $>>x$ represents a data shift with *x* bits to the right. Since the number *C* of checksums in a check word is always a power of 2, the operations of $>>\log_2 C$ can be executed very fast with the help of multiplexers. The parameter *OffSet* gives the address of the memory region containing the checksums for a certain protected memory zone. The role of the parameter *Mask* is to allow flexible sizes of the protected memory zones. The rest of *@CW* is given by the unmasked fragment of *@DW*.

The parameter *Offset* contains as many bits as required by the address of the smallest memory page. As illustrated in Table 1, the parameter *Offset* contains two bit fields, *offset1* and *offset2*, where the bits of *offset2* are all set to logic *0*. The parameter *Mask* is also composed of two bit fields, *mask1* and *mask2*. All bits of *mask1* are set to logic *0*, while the bits of *mask2* are set to logic *1*. The $\log_2 C$ least significant bits in the field *mask2* are always set to logic 1. Consequently, the 'min ($\log_2$C)' least significant bits in the field *mask2* do not need to be stored (the minimum function 'min' is taken over the parameters *C* corresponding to all EDAC codes simultaneously supported by the RSM). With respect to the example given in Table 1, the parameter *j* can vary between the parameters *31* and *i*.

The checksum position in a check word (*@Checksum*) that corresponds to a data word address (*@DW*) is given by the following expression:

$$@Checksum = @DW \,\%\, C \qquad (2)$$

Since the parameter *C* is a power of 2, the remainder of the arithmetic division *@DW%C* is obtained by selecting the log_2C least significant bits of the data word address *@DW*.

	Outer Page Address		Inner Page Address
	Bit-field 1	Bit-field 2	
@DW	$a_{31} \dots a_j$	$a_{j-1} \dots a_i$	$a_{i-1} \dots a_0$
Offset	*offset1*	0 … 0	-
Mask	0 … 0	1 … 1	-
@CW	*offset1*	$a_{j+1} \dots a_{i+2}$	$a_{i+1} \dots a_2$

Table 1: Calculation of *@CW* for a code with *C*=4 and 32-bit addressing.

The parameters *Offset*, *Mask* and other control flags like those used for the identification of the EDAC code can be stored in status registers. These registers should be reprogrammed each time the protection level of the memory accesses changes. These reconfigurations can be performed as a memory storage operation if a memory address is associated to each status register.

When the use of each type of EDAC code is confined to a fixed memory address zone which is known at design time, the status information can be hardwired. In this situation, the protection level of each memory access can be identified in a way similar to the detection mechanism employed in a memory protection unit (MPU) [2]. Let us assume a 32-bit memory address space, where each address zone that requires a certain protection level is identified by a pattern of the most significant address bits $c_{31}...c_m$. The belonging of a memory address $a_{31}...a_0$ to a protected zone is given by the following logic expression:

$$\left[\left(a_{31} \overline{\oplus} c_{31} \right) \vee zone_{31} \right] \wedge \cdots \wedge \left[\left(a_m \overline{\oplus} c_m \right) \vee zone_m \right] \qquad (3)$$

Here, the symbol '$\overline{\oplus}$' represents the '*exclusive nor*' logic operation. The size of the protected memory zone can be adapted by modifying the number of *zone* bits set to logic *0* where: $zone_{31}=0...zone_k=0$ and $zone_{k-1}=1...zone_m=1$. For a certain EDAC code, the parameters *m* and *k* can be derived from the parameters *i* and *j* in Table 1:

$$m = i + \log_2 C, \quad k = j + \log_2 C$$

Expression (3) provides the support for a flexible on-line partitioning of the memory address space into different protection zones. *Zone* and $c_{31}...c_m$ bits can be stored in status registers or hardwired just like the other RSM status information. Extending this information with memory access rights allows the use of the RSM as an MPU.

The RSM contains a small fully associative cache that can accommodate one *check word* per line. The number of cache lines is chosen to minimize the supplementary accesses to the memory for the largest wrapping burst [1] supported by the master and the EDAC code with the largest checksum size. For example, let us consider a burst transfer in which 8 data words are read in the following sequence of their least significant address bits: 00, 01, … , 07. Consider a protection scheme based on an EDAC code characterized by a parameter *C*=4. Except for a lucky cache hit or the detection of possible errors, an RSM with a single cache line needs 2 supplementary memory accesses to retrieve the corresponding check words. In case of a wrapping burst with the sequence of least significant address bits: 01, 02…07, 00, the RSM needs 3 supplementary memory accesses corresponding to the se-

quences 01…03, 04...07 and 00. The accesses generated by the sequences 01…03 and 00 will bring the same check word from the memory, since all the other address bits are identical. This supplementary cost can be avoided by providing an additional cache line combined with a least recently used (LRU) replacement policy. This will limit the number of supplementary accesses for all wrapping bursts of size 8 and a code characterized by C=4. According to this rule, 4 cache lines are required in order to efficiently support wrapping bursts of size 16, which is the maximum size of a wrapping burst supported by the AMBA AHB protocol [1].

In the RSM cache, a dirty bit is associated with each cache line. The dirty bit is useful in case of memory write accesses during which the corresponding checksums cannot be written into the memory without corrupting other checksums. In this case, the whole check word needs to be loaded into the RSM cache. For example, such a supplementary access is required if a single data word is accessed and the smallest addressable data word sub-unit, e.g. a byte, is used to store checksums corresponding to several data words. The supplementary access is not needed when the master initiates a burst of memory write accesses and the corresponding checksums can completely fill an addressable data word subunit.

The dirty bit is also useful to point out a corrected checksum error in case of a memory read access protected by error correcting codes. During idles cycles when no memory transactions are initiated, the dirty bits in the RSM cache are checked and modified check words are saved into the memory. This cleaning of the RSM cache will help to reduce the latencies of the future memory transactions.

In our RSM implementations, each cache tag contains the memory address of the last accessed data word with the checksum in the corresponding cache line. This is useful if a corrected data word has been delivered to the processor during a memory read access. If the address of the corrected data word is not available on the interconnection sub-system in the cycle subsequent to data delivery, the address stored into the cache tag can be used to update the corrected data word into the memory.

3 RSM-based system architectures

RSM may be used in several architectures. Let us consider a multi-master system with a bus-based interconnection subsystem and a communication protocol that supports a variable memory access time. In this case, the RSM may be integrated at the interface between the bus arbiter and main memory as shown in Figure 2. This configuration scales well with the number of masters in the system and it can hide to the arbiter the supplementary memory accesses initiated by the RSM. In this way, the access to the bus sub-system is not granted to any other master until the current protected access is executed. The atomicity of memory read accesses is especially important to limit the impact on system performance. Since the data integrity levels are assigned to zones of the memory address space, the RSM does not need to know to which master a memory access belongs.

Associating an RSM to each master, as illustrated in Figure 3, offers larger fault coverage of the interconnection subsystem than in the previous configuration and enables a better system performance. One reason for the lower performance overhead is the possibility to use a *lighter* protocol between the masters and the associated RSM to reduce the depth of the interconnection pipelining as seen by the master.

Non-conflicting configurations (*OffSet's*, *Mask's, Zone's* and *flags* used to indicate protection levels) of the RSM units associated with each master in the system must be ensured. Only slight differences exist between the implementations of the RSM modules used in the configurations sketched in Figure 2 and Figure 3.

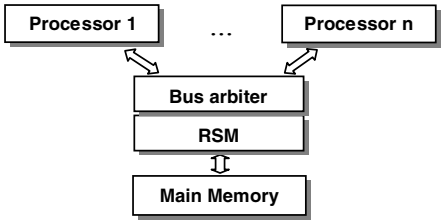

Fig. 2: RSM associated with the arbiter of the interconnection sub-system.

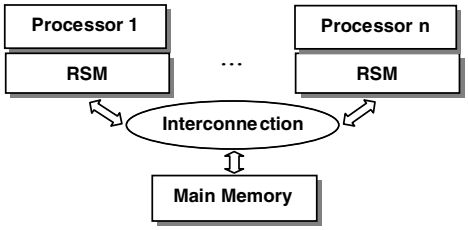

Fig. 3: RSM as wrapper of the master interface.

When a hardware-based memory management unit (MMU) [2, 8] is present in the system, an option is to place an RSM on top of the MMU (Figure 4). This will allow a maximal flexibility and granularity for the partitioning of the physical address space into different protection zones. In the physical address space, for each single page any protection level can be chosen independently of the neighbouring pages and the corresponding checksums can be placed anywhere. The virtual address space contains a number of contiguous protection zones which is equal to the number of integrity levels guaranteed by the system. The memory accesses generated by the MMU in case of a miss in the translation lookaside buffer (TLB) must also pass through the RSM in order to ensure the protection of the translation table from the main memory.

Efficient implementations of the RSM can be provided if the storage sub-system is accessible on a word-wise basis. Fortunately, before its execution the software is usually loaded into the main memory that is addressable on a word-wise basis. The transfer into the main memory of the protected data together with the corresponding checksums ensures implicit protection against the errors occurred in solid-state secondary storage that is accessible on a block-wise basis, such as flash memories. Blocks from the secondary storage can be transferred via an unprotected processor or a DMA port (see Figure 5) or with an RSM in bypass mode. The operating system should be aware of the presence of the redundant information associated with the protected data. Blocks of checksums or unprotected data can be transported in the same way from the main memory to the secondary storage. The blocks of protected data can be transferred from the main memory to the secondary storage via a processor or a DMA port with an RSM in a mode that protects only the memory read accesses.

Due to the separate addresses of data and verification bits, the RSM-based scheme enables the containment of address

errors. This additional protection is not offered by standard EDAC-based schemes excepting the case when specific detection mechanisms are provided like calculating the check bits for the concatenation of data and address bits [13].

We used the APS3 processor from Cortus Company [17] to implement the configuration presented in Figure 2 and we simulated several benchmark programs from the MiBench suite [9]. Compared to a custom design in which the data is encoded/checked at system-level, the performance overhead induced by the presence of RSM is low on processor with cache memories. Even in case of processors without cache memories, the performance impact is limited. On the other hand, an RSM-based approach might have a lower performance overhead than a standard component-based protection scheme in which an encoder/checker is allocated to each subsystem.

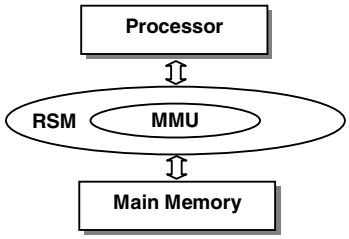

Fig. 4: RSM as MMU wrapper.

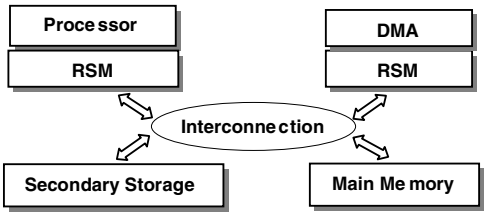

Fig. 5: Use of RSM with solid-state secondary storage subsystem.

4 Conclusions

We have presented a system-level hardware-based approach for the protection of main memory, interconnection and solid-state secondary storage sub-systems against soft errors. The new approach is characterized by the following properties:

- **Programmability and flexibility**: The location, the size and the integrity level of the protected zones in the memory address space can be set with the help of memory write accesses at especially reserved addresses. All these settings can be performed on-line or at design time.
- **Non-intrusive**: The proposed scheme can be implemented with the help of an IP module plugged into the system. None of the protected sub-systems needs modifications. Moreover, no modification and no augmentation of the application software are required, at least as long as no real-time applications are executed.
- **Low cost**: The protected storage sub-systems and the interconnections in-between may be constituted of simple and cheap modules without any intrinsic EDAC mechanism. The impact on the system performance is low, especially in systems where the processors use memory caches. Moreover, when no error protection is required, the proposed IP module is simply bypassed, which results in zero clock cycle overhead.

Acknowledgements

This work has been partially supported by the French Government and the Pôle SYSTEM@TIC in the region of Paris within the framework of the ConCEPT project.

We would like to thank Michael Chapman, president and chief executive officer of Cortus Company [17] for his help and advices concerning the APS3 architecture.

We are grateful to Olivier Heron, Erwan Piriou, Raphael David and Yves Lhuillier for their valuable suggestions.

References

[1] "AMBA™ Specification (Rev 2.0)," *ARM*.
[2] "ARM946E-S Technical Reference Manual," *ARM*.
[3] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, 2004, pp. 11-33,.
[4] R. Baumann "Soft Errors in Advanced Computer Systems," *IEEE Design and Test of Computers*, 2005, pp. 258-266.
[5] C.L. Chen, M.Y. Hsiao "Error-Correcting Codes for Semiconductor Memory Applications: A State of the Art Review," *Reliable Computer Systems - Design and Evaluation, Digital Press, 2nd edition*, 1992, pp. 771-786.
[6] T.J. Dell "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," *IBM Microelectronics Division*, 1997.
[7] E. Dupont, M. Nicolaidis, P. Rohr "Embedded Robustness IPs for Transient-Error-Free ICs," *IEEE Design & Test of Computers*, Vol. 19, No. 3, 2002, pp. 56-70.
[8] Gaisler Research "GRLIB IP Core User's Manual," Version 1.0.17, November 2007, pages 212, 233.
[9] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, "MiBench: A Free Commercially Representative Embedded Benchmark Suite," *IEEE Sixth Ann. Workshop Workload Characterization*, 2001, pp. 3-14.
[10] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, C. Dai "Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes," *Symposium on VLSI Technology Digest of Technical Papers*, pp. 73-74, 2001.
[11] T. Karnik, P. Hazucha, J. Patel "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 2, April-June 2004, pp. 128-143.
[12] P.K. Lala "Self-Checking and Fault-Tolerant Design," *San-Francisco, CA: Morgan Kaufmann*, 2001.
[13] R. Mariani, G. Boschi "A System Level Approach for Embedded Memory Robustness," *Solid-State Electronics 49*, 2005.
[14] N. Oh, P. P. Shirvani, E. J. McCluskey "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Transactions on Reliability*, 51(1):63–75, March 2002.
[15] P.P. Shirvani, N. Saxena, E. J. McCluskey "Software Implemented EDAC Protection against Seus," *IEEE Transactions on Reliability*, Vol. 49, No. 3, September 2000, pp. 273–284.
[16] L. Spainhower, T.A. Gregg "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," *IBM J. Research and Development,* vol. 43, nos. 5/6, 1999.
[17] http://www.cortus.com/aps3