Memory management unit for hardware-assisted dynamic relocation in on-board satellite systems

Borja Losa
Pablo Parra
Antonio Da Silva
Óscar R. Polo
J. Ignacio G. Tejedor
Agustín Martínez
Jonatan Sánchez
Sebastián Sánchez
Space Research Group, Universidad de Alcalá, Alcalá de Henares, Madrid, Spain

David Guzmán
NASA Goddard Space Flight Center, Greenbelt, MD, USA

Abstract—
Satellite on-board systems spend their lives in hostile environments where radiation can cause critical hardware failures. One of the most radiation-sensitive elements is memory. The so-called Single Event Effects (SEEs) can corrupt or even irretrievably damage the cells that store the data and program instructions. When one of these cells is corrupted, the program must not use it again during execution. In order to avoid rebuilding and uploading the code, a memory management unit can be used to transparently relocate the program to an error-free memory region.

This paper presents the design and implementation of a memory management unit that allows the dynamic relocation of on-board software. This unit provides a hardware mechanism that allows the automatic relocation of sections of code or data at run-time, only requiring software intervention for initialization and configuration. The unit has been implemented on the LEON architecture, a reference for European Space Agency missions. The proposed solution has been validated using the boot and application software of the instrument control unit of the Energetic Particle Detector of the Solar Orbiter mission as a base. Processor synthesis on different FPGAs has shown resource usage and power consumption similar to that of a conventional memory management unit. The results vary between ± 1-15% in resource usage and ± 1-7% in power consumption, depending on the number of inputs assigned to the unit and the FPGA used. When comparing performance, both the proposed and conventional memory management units show the same results.

This work has been supported by the Community of Madrid under the grant CM/JIN/2019-032 *(Corresponding author: Borja Losa).*

Authors Borja Losa, Pablo Parra, Antonio Da Silva, Óscar R. Polo, J. Ignacio G. Tejedor, Agustín Martínez, Jonatan Sánchez and Sebastián Sánchez are with the Space Research Group, Universidad of Alcalá, Edificio Politécnico, Crta. A-2, km. 33,600, 28805, Alcalá de Henares, Madrid, Spain (e-mail: borja.losa@edu.uah.es, pablo.parra@uah.es, antonio.dasilva@uah.es, o.rodriguez@uah.es, ignacio.garcia@uah.es, agustin.martinez@uah.es, jonatan.sanchezs@uah.es, sebastian.sanchez@uah.es). David Guzmán is with the NASA Goddard Space Flight Center, 8800 Greenbelt Road, Greenbelt, MD 20771 USA (e-mail: david.guzmangarcia@nasa.gov).

0018-9251 © 2023 IEEE

I. Introduction

Reliability is an essential feature in developing critical systems, such as satellite flight software. The correct operation of a computer system presupposes that the physical memory works correctly so that each read of a memory location returns the same value that was last written. Unfortunately, the space environment is prone to memory errors, so this assumption may prove false in certain situations. This is because electronic devices operating in space can be damaged due to what is known as *Single Event Effects* (SEE). These events are brought about by the incidence of charged particles on sensitive areas of integrated circuits, particularly memories. In a broad sense, from a software point of view, the most significant types of events are *Single Event Upsets* (SEUs) and *Single Event Latch-Ups* (SELs). The effects of the former, known as bit-flips, are transient, while the latter brings about permanent faults, known as “stuck-at” bits. So far, SEEs have caused several relevant spacecraft failures as described in [1] [2].

The *Soft Error Rate* (SER) expresses the rate at which a memory cell is disturbed or expected to encounter a SEE. The effects of radiation are not the same in each type of memory. In general, One Time Programmable (OTP) memories have a higher SEL threshold, i.e., better resilience to radiation. As an example, the EEPROM and SDRAM memories on board Solar Orbiter EPD Instrument Control Unit have a SEL threshold of 80 MeV cm^2/mg MeV while the PROM threshold rises to 128 MeV cm^2/mg [3].

In the architectural design of space-borne systems, it is widespread to divide the complete software functionality into two independent programs that are executed from different memory devices [4]. The first one, called Boot Software (BSW), performs the initial system configuration, initializes the essential communication support and implements the Safe or Standby operation mode. This software is stored and executed from a PROM to sufficiently protect it against possible failures caused by radiation, making it impossible to change it once it has been programmed. This impossibility of updating implies that the BSW is considered critical and has the highest reliability requirements.

The rest of the system functionalities that allow fulfilling the specific objectives of the mission are provided by the second of the programs, the so-called Application Software (ASW). The ASW manages the Nominal mode of operation, which allows the fulfilling of the specific objectives of the mission. In this mode, the software integrates full communications support and control of all subsystems and devices that comprise the mission. An ASW image is usually stored statically in a non-volatile EEPROM and, in order to be executed, this ASW image has to be deployed into RAM. The BSW performs this after a comprehensive check of the destination memory blocks. This dual configuration, which combines the static storage of the binary image in EEPROM and its subsequent deployment in RAM, facilitates software maintenance, allowing possible updates and changes during the mission [4].

If the code and data of the ASW are statically linked at fixed memory locations, a permanent error in any of the target cells would affect the entire image, making it invalid for use. In the event of such an error, it would be necessary to generate a new binary image avoiding the affected cells and transfer it back to the spacecraft or the corresponding instrument control unit, with the consequent consumption of both time and transmission bandwidth resources. However, this operation is not risk-free since a software upgrade is conducted in a complex computing environment that usually involves long delay spacecraft communications and coordination. These circumstances may eventually lead to the loss of the mission [5].

A *Memory Management Unit* (MMU) can be used to avoid re-linking and reloading the code. These hardware units make it possible to translate the addresses generated by the programs, called virtual addresses, into physical addresses of RAM. Thus, thanks to this translation mechanism, programs can be easily relocated in memory without having to be statically relinked.

A. Conventional memory management units

Memory management units are ubiquitous in general-purpose systems and are particularly well suited to provide virtual memory and protection mechanisms [6]. These features are generally unnecessary for embedded systems where the operating system and the application are intertwined in a single executable running in the same address space without memory protection.

Nearly all units implement a paging-based approach, in which the virtual address space is divided into power-of-two sized blocks called *pages*. Through a translation mechanism, the memory management unit can allocate each page in the space considered valid to a physical memory block or *frame* of the same size.

Depending on the level of involvement of both hardware and software in the translation process, it is possible to distinguish between *hardware-managed* and *software-managed* MMUs.

Paging-based hardware-managed MMUs implement a translation mechanism based on the use of data structures called *page tables* that are stored in main memory. Each entry in these tables is associated with one page and contains information about its validity and access permissions and the physical memory frame to which it is mapped. In order to reduce the memory footprint, architectures often define different *page table levels*. The implementation of this mechanism may vary depending on the architecture, generally allowing the coexistence of different levels and page sizes.

One of the main problems derived from using a hardware-managed table-based paging mechanism is the reduction in performance derived from the need to carry out additional memory reads for each access to obtain the contents of the page tables. In order to increase performance, architectures include one or more hardware-managed Translation Lookaside Buffers (TLBs). These units contain a fixed number of entries that store the result of previous translations and are used as caches to speed up address translation. Hardware architectures implement cache replacement algorithms that manage their occupancy which adds indeterminism to the execution, a problem that real-time systems often cannot afford [7]. Nearly all current versions of major architectures, such as x86-64 [8], [9], ARMv8 [10], SPARCv8 [11] and RISC-V [12], include a hardware-managed table-based paging mechanism.

These units allow the transparent remapping of affected pages to other error-free physical addresses in the case of permanent errors. However, as stated above, its use implies a memory overhead derived from storing the page tables in memory. Memory consumption in page tables can be reduced by using larger pages. The problem with this solution is that it introduces another type of memory overhead because pages are the minimum relocation units. Thus, even if only one bit is affected, the entire page is tagged as erroneous. This means that a significant amount of memory is rendered unusable for each such error when a page is remapped. On the other hand, if the page size is small (the minimum page size is, in most cases, 4 KiB), the system will need to store a larger number of page tables in memory. In addition, the need to store page tables in main memory makes them vulnerable to possible memory errors.

Software-managed MMUs, on the other hand, delegate to the system software the maintenance of the structures needed to define the virtual address spaces. These units provide a set of one or more TLBs whose entries can be directly modified by the software through specific instructions. Each of these entries maps a single page. The format and characteristics of the entries set the range of allowed page sizes. When performing a translation, the MMU tries to find a valid TLB entry that maps the page to which the requested address belongs. If there is no such entry, a fault or trap will be generated that will cause the execution of a memory handling routine. Executing these routines is costly; thus, a high number of faults will result in additional performance costs, which are particularly high when compared to their hardware-managed counterparts. The MIPS architecture implements an example of this kind of unit [13].

However, dynamic or virtual memory mechanisms introduce a high degree of indeterminism, so their use in spaceborne systems is avoided. In this type of system,

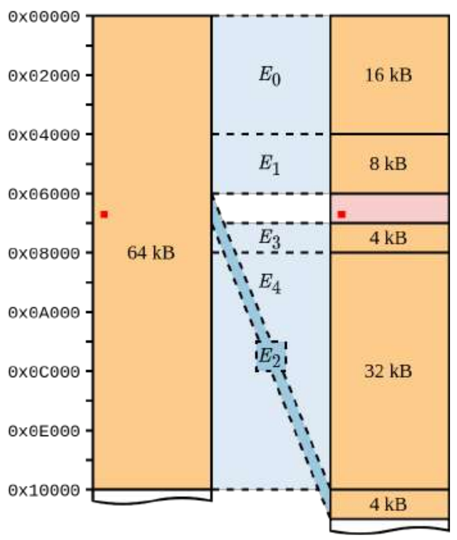


Fig. 1. Diagram showing the need to use several entries when relocating a small block belonging to a larger one.

which is the subject of our study, the address map is static, unique, and known at linking and deployment time. This way, the virtual address map can be built using a fixed number of TLB entries, avoiding the generation of memory faults and thus eliminating indeterminism and minimizing the performance impact. Therefore, these units are particularly suitable for code relocation in space systems since, by not using any external support structure to perform the translation, they do not increase memory space consumption and do not use RAM cells that could be affected by permanent failures.

In this same context, paging-based memory management units, both hardware and software-managed, make it difficult to relocate small memory blocks by setting limits on the size of the pages or using fixed page sizes. For example, in the case of the MIPS architecture, pages have a minimum size of 4 KiB or 1 KiB, depending on the version. This limit implies that, in the case of detecting a permanent error in a single memory cell, the minimum amount of memory to be relocated would be that corresponding to the lower limit of the page size.

Another problem with conventional MMUs is that the initial address of the pages always has to be aligned with their size. This results in the need to use multiple TLB entries to relocate a single damaged block. This problem is also encountered in systems implementing a hardware-managed paging mechanism that supports different page sizes. Figure 1 shows an example of this situation. In this case, a 64 KiB block is mapped initially to physical memory using a single entry.

If a fault is detected in a memory cell, for example, in cell number 0x6C00, there are two alternatives. The first would be to relocate the entire 64 KiB memory block to another location. This situation would result in a considerable loss of memory that could otherwise be used. The second alternative would be relocating only a smaller block containing the damaged cell. Suppose we assume a minimum page size of 4 KiB. In this case, the minimum relocatable amount of memory will correspond to the 4 KiB block starting from address 0x6000 to 0x7000. As can be seen in the figure, to relocate this block, it would be necessary to fragment the original page into five different-sized pages, forcing us to use the same number of entries.

In the case of software-managed MMUs, the number of TLB entries is fixed. Thus, high fragmentation would lead to excessive use of entries which would reduce relocation capabilities.

B. Proposed memory management unit

This paper presents a software-managed memory management unit that has been specifically designed to facilitate code relocation in on-board space systems. It features two fundamental features: the flexibility in the size of the pages and the possibility of defining overlapping pages.

The proposed unit allows defining pages of any size. This feature minimizes the amount of memory to be provisioned for possible relocations. This saving is essential in systems with very restrictive design requirements, such as space systems.

By allowing the definition of pages of any size, it is necessary to deal with the possible overuse of TLB entries. In order to minimize the need for entries in the event of a memory relocation, the proposed management unit implements an overlapping mechanism that allows mapping a smaller page over one already mapped by another entry. Thus, following the same example above, to map the 4 KiB memory page over the 64kB projection, it would only be necessary to add a new overlapping entry corresponding to the smaller page. Furthermore, it would not be necessary to relocate an entire 4 KiB block in this case since the unit supports the definition of pages of any size.

The proposed design has been implemented on the LEON processor version 3, which is one of the recommended processors by the European Space Agency (ESA) for space applications. This work could be understood as a sort of fault tolerance property of memory management units. This way, the risks associated with long-term uncertainties related to memory malfunction are strongly reduced. Both the memory management unit itself and the algorithm used to perform the relocation have been validated using an actual use case that involves the boot and application software of the instrument control unit of the Energetic Particle Detector of the Solar Orbiter mission [14], [15].

The rest of the paper is organized as follows. The following section contains the description of other existing solutions related to our proposal. Section III presents the design of the proposed memory management unit. Section IV introduces the use case and a generic algorithm that allows the relocation of damaged memory blocks. Section V includes the details of the implementation performed, its validation, and the subsequent analysis of the obtained results. Finally, the VI section summarizes the conclusions and future work.

II. Related works

Dealing with memory errors brought about by radiation in the space application domain is a broad research field that has generated many publications over the last few years. Siegle and Vladimirova [16] provide a detailed analysis of commonly used mitigation techniques in SRAM-Based FPGAs for space applications. A distinction is made between fault prevention and fault tolerance. The former comprises actions aimed at preventing the introduction of faults during design or during the operational runtime of the system. The latter encompasses all techniques that allow system operation to continue after a fault has occurred, either by masking or mitigating the impact of the fault.

Within fault tolerance techniques, it is common to distinguish between the fault detector and the fault corrector. The function of the detector is to find faults in memory words by indicating which bits have been affected. The most common way to perform fault detection is through Error Correcting Codes (ECC) and modular redundancy schemes. These techniques are based on using redundant information to know if a memory word has errors and the bits involved in those errors [17].

Error correction is carried out during memory access. Usually, the redundancy introduced allows the correction of just one error. Thus, if more than one error exists, it would be detected but not corrected. Therefore, to avoid possible errors, a periodic refreshing process known as memory scrubbing is carried out. This technique is very effective against transient errors, such as SEUs, but cannot solve permanent errors. In this situation, the best solution is the dynamic relocation of the faulty section, as done in Polo et al. [3]. This solution is not always possible, so an updated binary version that avoids damaged areas must be uploaded to the spacecraft in the worst case.

Our proposal would fall under fault tolerance techniques, specifically for permanent error correction, by relocating memory during system startup. It can be used together with other correction schemes such as ECC and memory scrubbing.

In the work of Świercz et al. [18], we can find one of the few examples of the use of virtual memory to implement fault-mitigation techniques. It describes the use of a conventional paging-based MMU in combination with a selection algorithm. The system makes two additional copies of each page during the initialization process. Each time a page is accessed that is not previously loaded in memory, the algorithm compares the three copies and obtains the result through a majority vote. In order to ensure the correct functioning of this mechanism, it is recommended to disable the cache. The process of comparing and selecting pages, together with the inability to use the cache, implies a high impact on performance.

Regarding the use of virtual memory in embedded systems, most works aim to provide techniques that facilitate the use of dynamic memory in a deterministic way, which is essential in real-time systems. For example, Zhou and Petrov [19] propose a new type of page table organization that reduces memory requirements while providing predictability for the table lookup process. Böhnert and Scholl [20] describe a solution with data structures that ensures that operations such as memory allocation or deallocation are performed in constant time. Meenderinck et al. [21] propose the design of a predictable MMU for an SoC by keeping page tables in on-chip memory and not allowing page swapping with any secondary storage.

Predictability is also one of the objectives of the MMU proposed in this work. It is achieved by delegating MMU management to software and focusing on systems that do not use dynamic memory since all necessary memory allocations are known and performed during system initialization and never change at runtime. Not having to use dynamic memory makes it unnecessary to implement any data structures such as page tables, which allows us to eliminate any memory overhead and thus improve radiation tolerance.

III. Design of the proposed memory management unit

This section provides a formal, platform-independent specification of the hardware behavior of the proposed Memory Management Unit (MMU) that is intended to facilitate its implementation and inclusion in different architectures. The MMU employs a software-managed approach based on a Translation Lookaside Buffer (TLB) with multiple entries. In order to integrate the unit into a processor architecture, the latter must provide a hardware-software interface to access the contents of the individual entries. This specification assumes that such a mechanism is already in place. Subsequently, in Section V, a complete implementation example for the LEON processor is provided.

Let $v \in \mathbb{N}$ be the size in bits of the virtual address space, determined by the processor architecture. The virtual address space $\mathbb{V}$ is defined as $\mathbb{V} := \{x \in \mathbb{N} \mid 0 \leq x < 2^v\}$. Let $f \in \mathbb{N}$ be the size expressed in bits of the physical address space. Depending on the architecture, it may or may not coincide with the size of the virtual address space. The physical address space $\mathbb{F}$ is defined as $\mathbb{F} := \{x \in \mathbb{N} \mid 0 \leq x < 2^{\mathrm{f}}\}$.

The virtual address space can be partitioned into n *pages* of size $s \in \mathcal{S}$, where $n := 2^{(v-s)}$ and $\mathcal{S} := \{0, 1, \ldots, f\}$. The i-th virtual page of size s, denoted as VP_i^s, is defined as $\mathsf{VP}_i^s := \{x \in \mathbb{V} \mid i \cdot 2^s \leq x < (i+1) \cdot 2^s\}$ with $i \in \mathcal{N}_s$, where $\mathcal{N}_s = \{0, 1, \ldots, n-1\}$ is the set consisting of all possible indexes or *page numbers* of size s. It should be noted that the maximum page size is fixed by the size of the physical address space so that pages can never be larger than physical memory.

For example, assuming a virtual address space of size $v = 32$ in a system implementing 1-byte memory cells, the page VP_0^{30} would represent a contiguous block of 1 GiB comprising the addresses between the `0x00000000` and the `0x3FFFFFFF`. As a second example, the page VP_1^{12}, would correspond to a block of 4 KiB in size that comprises the addresses between `0x00001000` and `0x00001FFF`.

Likewise, the physical address space can be partitioned into m *frames* of size $s \in \mathcal{S}$, where $m = 2^{f-s}$. The j-th physical frame of size s, denoted as PF_j^s, is defined as $\mathsf{PF}_j^s := \{x \in \mathbb{F} \mid j \cdot 2^s \leq x < (j+1) \cdot 2^s\}$ with $j \in \mathcal{M}_s$, where $\mathcal{M}_s = \{0, 1, \ldots, m-1\}$ is the set consisting of all possible indexes or *frame numbers* of size s.

Each of the TLB entries allows mapping a page of a given size of the virtual address space of the applications to a physical memory frame of the same size. Each time memory is accessed, the processor delegates to the TLB the translation of the corresponding address. The MMU receives a virtual address and checks if it matches any pages defined in the valid TLB entries. If a matching entry is found, it obtains the corresponding physical address. The resulting physical address will belong to the physical frame into which the virtual page has been mapped. The distance or offset between the physical address and the starting address of the frame will be equal to the offset between the original virtual address and the starting address of the page.

If the requested address does not match any of the entries, the MMU sends an exception signal to the processor, which will trigger the execution of a software handler to manage the failure.

One of the central and innovative features of the proposed MMU is that it allows the definition of overlapping pages. In this way, a TLB entry can map a page whose address range coincides totally or partially with another page mapped by another entry. To resolve the selection, the MMU defines the *overlapping flag* of each page, which allows establishing which of the entries is going to be used to perform the mapping and obtain the final physical address.

For a TLB of p entries, the i-th entry is defined as the tuple $\mathsf{E}_i := (E_i^\sigma, E_i^\nu, E_i^\mu, E_i^\delta, E_i^o)$, where:

- $E_i^\sigma \in \mathcal{S}$: is the size of the page mapped by the entry.
- $E_i^\nu \in \mathcal{N}_{E_i^\sigma}$: is the number of the page of size E_i^σ of the virtual address space from which the mapping is to be performed.
- $E_i^\mu \in \mathcal{M}_{E_i^\sigma}$: is the number of the frame of size E_i^σ of the physical address space over which the mapping is to be performed.
- $E_i^\delta \in \{0, 1\}$: denotes the validity of the entry. If $\delta = 0$, the entry is considered invalid and is therefore not evaluated when executing the translation procedure. On the other hand, if $\delta = 1$, the entry is valid and participates with its content in the translation.

TABLE I
Example of a TLB configuration

E_i	σ	ν	μ	δ	o
E_0	31	0	0	1	0
E_1	30	2	3	1	0
E_2	28	2	1	1	1
E_3	29	2	3	1	1
E_4	27	10	6	1	1
E_5	28	9	10	1	0
E_6	0	0	0	0	0
E_7	0	0	0	0	0

- $E_i^o \in \{0, 1\}$: indicates the overlapping flag of the entry. This field is used in the translation process as explained below.

Formally, a given $\mathfrak{C}$ configuration of a TLB of p entries, is defined as the indexed family $\mathfrak{C} := \{\mathsf{E}_i\}_{i \in \mathcal{I}}$, where $\mathcal{I} = \{0, 1, \ldots, p-1\}$, and E_i is the configuration of the i-th entry of the TLB.

Table I shows an example configuration of an 8-entry TLB over a 32-bit virtual and physical address spaces, i.e. $v = f = 32$. In this example, the first six entries are valid, and the last two are marked as invalid. Entry E_0 is mapping an area of size $E_0^\sigma = 31$. If we assume a cell size of one byte, the full size of the corresponding virtual memory page would be 2^{31} bytes. The page number set by the entry is $E_0^\nu = 0$, so the virtual page defined by the it would be VP_0^{31}, corresponding to virtual address range $[0 \cdot 2^{31}, ((0+1) \cdot 2^{31}) - 1] = [\mathtt{0x00000000}, \mathtt{0x7FFFFFFF}]$. The entry performs a mapping of the virtual page to the physical frame $E_0^\mu = 0$. In this case, the frame number and the page number match, so it is a 1-to-1 projection in which both share the same address range. The entry is marked as valid ($E_0^\delta = 1$), and as non-overlapping ($E_0^o = 0$).

Following the same approach, the remaining valid entries perform the following mappings:

- E_1. Maps virtual page VP_2^{30}, with address range $[\mathtt{0x80000000}, \mathtt{0xBFFFFFFF}]$, to physical frame PF_3^{30} with range $[\mathtt{0xC0000000}, \mathtt{0xFFFFFFFF}]$. The entry is marked as valid and non-overlapping.
- E_2. Maps virtual page VP_2^{28}, with address range $[\mathtt{0x20000000}, \mathtt{0x2FFFFFFF}]$, to physical frame PF_1^{28} with range $[\mathtt{0x10000000}, \mathtt{0x1FFFFFFF}]$. The entry is marked as valid and overlapping.
- E_3. Performs the mapping of page VP_2^{29}, with virtual address range $[\mathtt{0x40000000}, \mathtt{0x5FFFFFFF}]$, to frame PF_3^{29}, with physical address range $[\mathtt{0x60000000}, \mathtt{0x7FFFFFFF}]$. The entry is marked as valid and overlapping.
- E_4. Maps virtual page VP_{10}^{27}, corresponding to the address range $[\mathtt{0x50000000}, \mathtt{0x57FFFFFF}]$, to frame PF_6^{27}, with physical address range $[\mathtt{0x30000000}, \mathtt{0x37FFFFFF}]$. The entry is marked as valid and overlapping.
- E_5. Maps virtual page VP_9^{28}, with address range $[\mathtt{0x90000000}, \mathtt{0x9FFFFFFF}]$, to physical frame

PF^{28}_{10}, with range $[\texttt{0xA0000000}, \texttt{0xAFFFFFFF}]$. The entry is marked as valid and non-overlapping.

The complete translation procedure is formally defined as the function $translate_{\mathfrak{C}} : \mathbb{V} \to \mathbb{F} \cup Error$, where $Error := \{undefined, trap\}$ is the set of possible errors that can occur during the translation process. The $translate_{\mathfrak{C}}$ function is specific to the $\mathfrak{C}$ configuration and receives as its only input the virtual address to be translated. If the translation process completes successfully, the function returns the physical address corresponding to the input virtual address. If the translation fails, the function will return an erroneous value depending on the cause of the failure, as we will see below.

The translation function uses two additional functions. The first of these, $valid_{\mathfrak{C}}$, selects from the set of valid entries those whose pages include the virtual address that is the subject of the translation. The function is defined as:

$$\begin{array}{rccc} valid_{\mathfrak{C}}: & \mathbb{V} & \to & \mathscr{P}(\mathcal{I}) \\ & \upsilon & \mapsto & \mathcal{J} \end{array}$$

where $\mathcal{J} \in \mathscr{P}(\mathcal{I})$ is the set consisting of the indexes corresponding to the valid entries whose mapped pages contain the address υ, and $\mathscr{P}(\mathcal{I})$ is the power set of indexes. formally:

$$\mathcal{J} = \{j \in P \mid E^{\delta}_{j} = 1 \land \upsilon \in \mathsf{VP}^{E^{\sigma}_{j}}_{E^{\nu}_{j}}\}$$

In this way, the $valid_{\mathfrak{C}}$ function allows locating the valid entries within the TLB configuration whose pages contain the virtual address to be translated.

Starting from the example of Table I, we intend to translate virtual address $\texttt{0x1000}$. If we apply the function $valid_{\mathfrak{C}}$ to the address, $valid_{\mathfrak{C}}(\texttt{0x1000})$, it will return the set $\{0\}$ since E_0 is the only entry that satisfies that it is valid ($E^{\delta}_{0} = 1$) and that the virtual page it defines, i.e., $\mathsf{VP}^{E^{\sigma}_{0}}_{E^{\nu}_{0}} = \mathsf{VP}^{31}_{0} = [\texttt{0x00000000}, \texttt{0x7FFFFFFF}]$, contains address $\texttt{0x1000}$. Continuing with the example, for virtual address $\texttt{0x20000000}$, $valid_{\mathfrak{C}}(\texttt{0x20000000})$ will return the set $\{0, 2\}$, since both entries, E_0 and E_2, satisfy that they are valid and their corresponding virtual page contains address $\texttt{0x20000000}$. On the other hand, if we apply the function to address $\texttt{0xC0000000}$, it will return $valid_{\mathfrak{C}}(\texttt{0xC0000000}) = \emptyset$, since there is no valid entry whose virtual page contains the requested address.

Suppose the application of the function $valid_{\mathfrak{C}}$ does not return any valid entry. In that case, the memory management unit will cause a processor exception which, in turn, will trigger the corresponding software routine that should handle the fault. If the function returns a single valid entry, that will be the selected entry. On the other hand, if the function returns two or more valid entries, it would be necessary to check their overlapping flag. In that case, the selected entry would be the one with the overlapping flag equal to 1. If there were no entries with the overlapping flag equal to 1, or if there were more than one entry set to that flag value, the memory management unit would not perform the selection effectively and would produce an indeterminate result.

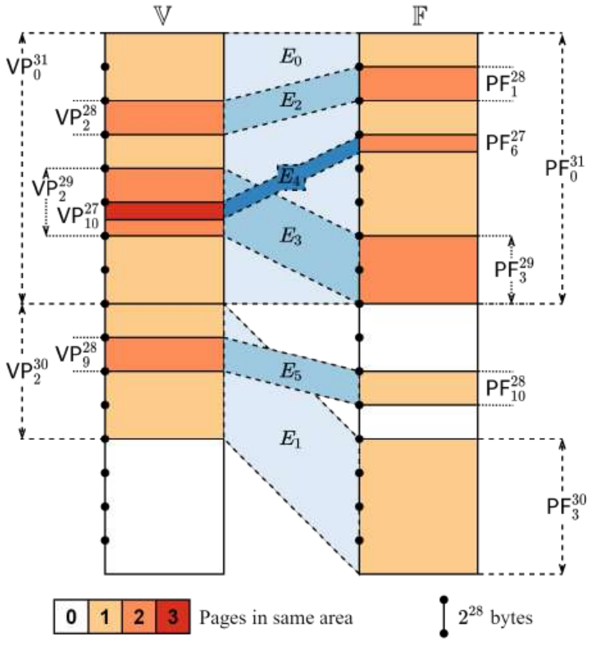


Fig. 2. Diagram showing the memory mapping example given by Table I

This selection would be performed by function $atop_{\mathfrak{C}}$, which is defined as follows:

$$\begin{array}{rccc} atop_{\mathfrak{C}}: & \mathscr{P}(\mathcal{I}) & \to & \mathscr{P}(\mathcal{I}) \\ & \mathcal{J} & \mapsto & \{t \in \mathcal{J} \mid E^{o}_{t} = 1\} \end{array}$$

Figure 2 shows a diagram with the mappings generated from the TLB configuration set in Table I. In it, the different mappings of virtual pages made by the entries marked as valid are represented. Continuing with this example, if we apply the function $valid_{\mathfrak{C}}$ to address $\texttt{0x1000}$, $valid_{\mathfrak{C}}(\texttt{0x1000}) = \{0\}$, it returns a set with a single element. Thus, the selected entry will be the one corresponding to that element, that is, E_0. If we attempt to translate address $\texttt{0xC0000000}$, the memory management unit will raise an exception (*trap*), since the application of the function $valid_{\mathfrak{C}}(\texttt{0xC0000000})$ fails to locate any valid entry containing the requested address.

On the other hand, if we apply the function to address $\texttt{0x20000000}$, $valid_{\mathfrak{C}}(\texttt{0x20000000}) = \{0, 2\}$, it will return two elements. Thus, it is necessary to check their overlapping flag using the function $atop_{\mathfrak{C}}$. If we apply the function to the set of valid entries, we obtain $atop_{\mathfrak{C}}(\{0, 2\}) = \{2\}$, since E_2 is the only entry in that set that is marked as overlapping ($E^{o}_{2} = 1$). Therefore, E_2 would be the selected entry to perform the translation.

If we apply the function $valid_{\mathfrak{C}}$ to the address $\texttt{0x50000000}$, $valid_{\mathfrak{C}}(\texttt{0x50000000})$, it will return the set $\{0, 3, 4\}$. Checking the overlapping with $atop_{\mathfrak{C}}(\{0, 3, 4\})$ yields the set $\{3, 4\}$, since both entries E_3 and E_4 are marked as overlapping. Since the

$atop_{\mathfrak{E}}$ function returns more than one element, an indeterminate result is produced and the memory handling unit triggers an $undefined$ error. If, on the other hand, we try to perform the translation of address 0x90000000, $valid_{\mathfrak{E}}(\texttt{0x90000000})$ returns the set $\{1, 5\}$. Next, $atop_{\mathfrak{E}}(\{1, 5\}) = \emptyset$ returns the empty set because neither of the two entries, E_1 and E_5, are marked as overlapping and, as in the previous case, an $undefined$ error would be triggered.

Function $translate_{\mathfrak{E}}$ would then be defined as follows:

$$\begin{array}{rccc} translate_{\mathfrak{E}}: & \mathbb{V} & \rightarrow & \mathbb{F} \cup Error \\ & v & \mapsto & \rho \end{array}$$

where:

$$\rho = \begin{cases} trap & \text{if } valid_{\mathfrak{E}}(v) = \emptyset \\ phys(v, \mathrm{E}_{j_0}) & \text{if } valid_{\mathfrak{E}}(v) = \{j_0\} \\ phys(v, \mathrm{E}_{t_0}) & \text{if } atop_{\mathfrak{E}}(valid_{\mathfrak{E}}(v)) = \{t_0\} \\ undefined & \text{otherwise} \end{cases}$$

Function $phys$ is used to obtain the target physical address from the virtual address and the configuration of the selected entry. The function calculates the target address by adding to the initial address of the physical frame the offset of the virtual address within the corresponding page. In order to obtain the offset, it is only necessary to apply the modulus function to the virtual address and the size of the page expressed in cell count, i.e., $2^{E_i^{\sigma}}$. Formally:

$$\begin{array}{rccc} phys: & \mathbb{V} \times \mathfrak{E} & \rightarrow & \mathbb{F} \\ & v, \mathsf{E}_i & \mapsto & E_i^{\mu} \cdot 2^{E_i^{\sigma}} + (v \bmod 2^{E_i^{\sigma}}) \end{array}$$

Continuing with the example of Table I, for the address 0x1000 we checked that the entry E_0 was obtained. Then, the resulting physical address will be $phys(v, E_0) = 0 \cdot 2^{31} + (\texttt{0x1000} \bmod 31) = 0 + \texttt{0x1000} = \texttt{0x1000}$. In this case, the physical and virtual addresses coincide, since entry E_0 performs a 1-to-1 mapping. If we apply the function $valid_{\mathfrak{E}}$ to virtual address 0x20001234, $valid_{\mathfrak{E}}(\texttt{0x20001234}) = \{0, 2\}$ we obtain two valid entries. After filtering those entries by applying function $atop_{\mathfrak{E}}$, we obtain $atop_{\mathfrak{E}}(\{0, 2\}) = \{2\}$. Therefore, in this case the selected entry for translation would be E_2, and the resulting physical address $phys(v, E_2) = 1 \cdot 2^{28} + (\texttt{0x20001234} \bmod 28) = \texttt{0x10000000} + \texttt{0x1234} = \texttt{0x10001234}$.

IV. Use case: spaceborne systems

The memory management unit described in this work is specifically designed to facilitate code relocation in embedded systems and, more specifically, in on-board space systems. As already mentioned, in this kind of system, it is common to divide the software into two independent programs: the boot software (BSW) and the application software (ASW). The first one is executed from a highly reliable PROM and provides a minimum functionality linked to the so-called Safe or Standby mode. The ASW, much more prominent in terms of complexity and space, provides the system's full functionality and is associated with the Nominal mode of the mission. In order to allow for future modifications and upgrades, the ASW is statically stored in the system in a non-volatile EEPROM.

The BSW performs the system's initial configuration and, if all the requirements are met, deploys the ASW from the EEPROM to RAM and transfers the control to it. Before deployment, the BSW thoroughly checks the RAM, verifying the proper functioning of all the memory cells in which the software application will be placed. If any damaged cell is detected, the boot software cancels the deployment and remains in safe mode awaiting instructions from the spacecraft or ground in each case.

For reasons of simplicity, space and performance, in these systems, the application or user code and the system code are linked together and share the physical address space. This linking is done statically, so that the addresses of the various code and global data sections are set at build time. Thus, if a permanent error occurs in a memory cell occupied by code or global data, this causes the entire application software image to become invalid. In this case, it would be necessary to re-generate the image making the linker avoid the affected memory addresses, and then transmit the new static image to the spacecraft for updating. The BSW of the affected system performs this update.

The proposed memory management unit (MMU) can be used to avoid having to perform a software update every time a permanent RAM error is detected. The proposed approach, based on the use of a software-managed Translation Lookaside Buffer (TLB), allows the code relocation to be performed autonomously and transparently to the ASW itself. It is worth noting that since the MMU does not use any external support structure to perform the translation, the proposed solution does not consume any memory space, nor does it use any RAM cells that could potentially also be affected by a permanent failure. In addition, the possibility of overlapping TLB entries reduces the number of entries needed to remap damaged blocks and significantly simplifies the remapping algorithm. The following paragraphs describe the design of a possible algorithm that, using the proposed MMU, makes code relocation possible, avoiding the need for an application software update.

A. Design of the relocation algorithm

A spaceborne system like the one described before, which also implements the proposed MMU, allows for the easy integration of an algorithm for the relocation of damaged memory blocks. This algorithm is implemented as part of the BSW, and is executed before the ASW deployment in order to be able to relocate the memory blocks that may be damaged. In order to correctly implement the algorithm, it is necessary to have a specifically reserved

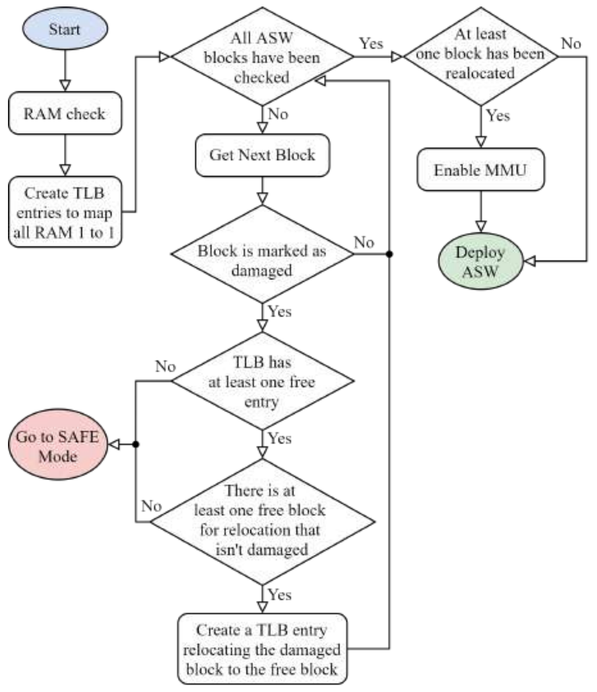

Start
RAM check
Create TLB entries to map all RAM 1 to 1
All ASW blocks have been checked
Yes
No
At least one block has been realocated
No
Yes
Get Next Block
Enable MMU
Block is marked as damaged
No
Deploy ASW
Yes
TLB has at least one free entry
No
Yes
Go to SAFE Mode
There is at least one free block for relocation that isn't damaged
No
Yes
Create a TLB entry relocating the damaged block to the free block

Fig. 3. Diagram of the proposed relocation algorithm.

RAM area where the relocations can be performed. Thus, the complete set of available RAM addresses can be divided into two partitions or regions: a first partition where the initial ASW deployment would be conducted and a second partition composed of the memory cells available to perform the possible relocations.

Figure 3 shows an activity diagram describing the operations performed by the relocation algorithm. First, the BSW checks the state of all RAM cells. The entire RAM is divided into fixed-size blocks as part of this check. The size of the blocks is a design consideration, which will depend primarily on the size of memory available for deployment. The checking process analyzes each block separately, applying specific tests to all cells in the block to rule out possible permanent faults. The BSW implements a map in which it maintains the status of every memory block, indicating whether the block has successfully passed all tests and is considered suitable for deployment or whether, on the contrary, an error has been detected in any of its cells it is marked as invalid.

After performing the memory check, the boot software configures the required TLB entries to perform a 1-to-1 mapping from virtual memory to physical memory. The number of required entries will depend on the memory management unit's implementation and available capabilities. It then executes a loop that iterates over the deployment memory blocks occupied by the application software. The algorithm checks whether each block has been marked as corrupted during the checking process. If so, performs the following operations:

1) Checks if there is an entry available in the TLB to perform the relocation. If not, it generates an error.
2) Checks if there is a free and undamaged block in the relocation area. If not, it generates an error.
3) Uses the selected TLB entry to map the damaged memory block onto the block in the relocation area.

If an error occurs during the process, the boot software must remain in Safe mode, waiting for instructions from a hierarchically superior system. The MMU is disabled during the whole process and is activated only in the case that, after having finished checking all the ASW blocks, at least one relocation has to be performed. After the execution of the algorithm, the boot software starts the ASW deployment phase in RAM on the original addresses without any further modification of the configuration. Any relocation that has been made will be resolved in a transparent manner by the MMU, which will make the corresponding translations in each case.

To illustrate this mechanism, we assume an on-board system with 4 MiB of RAM to deploy the ASW. The code, data, and stack sections of the ASW occupy 2 MiB. Therefore, the system has an additional 2 MiB of RAM available to perform possible relocations. We divide the memory into 16 blocks of 256 KiB to implement the relocation algorithm. Of these, eight would be occupied by the ASW itself ($\{ASW_0, \ldots, ASW_7\}$), and the remaining eight would be available to be able to relocate the different sections if necessary ($\{REL_0, \ldots, REL_7\}$). Assuming a TLB of 8 entries and that, after running the block check, the second relocation block (REL_1) and the first and third ASW blocks (ASW_0 and ASW_2) have been marked as damaged, the relocation algorithm would perform the following actions:

- Uses the first TLB entry (E_0) to map the entire system memory 1 to 1.
- Checks the first ASW block (ASW_0). Since the block is marked as damaged, perform the following operations:
  - Checks that it has free entries in the TLB. In this case, it still has seven entries available.
  - Check that the first block in the relocation area (REL_0) is free and not damaged.
  - Uses the second TLB entry (E_1) to map the virtual page whose addresses match those of block ASW_0 to the physical frame corresponding to block REL_0 and mark this entry as overlapping.
- Checks the second ASW block (ASW_1). Since it is not marked as damaged, continue with the next block.
- Checks the third block of the ASW (ASW_2). Since the block is marked as damaged, perform the following actions:
  - Checks that it has free entries in the TLB. In this case, it still has six entries available.
  - Checks the status of the second block of the relocation area (REL_1). Since, despite being

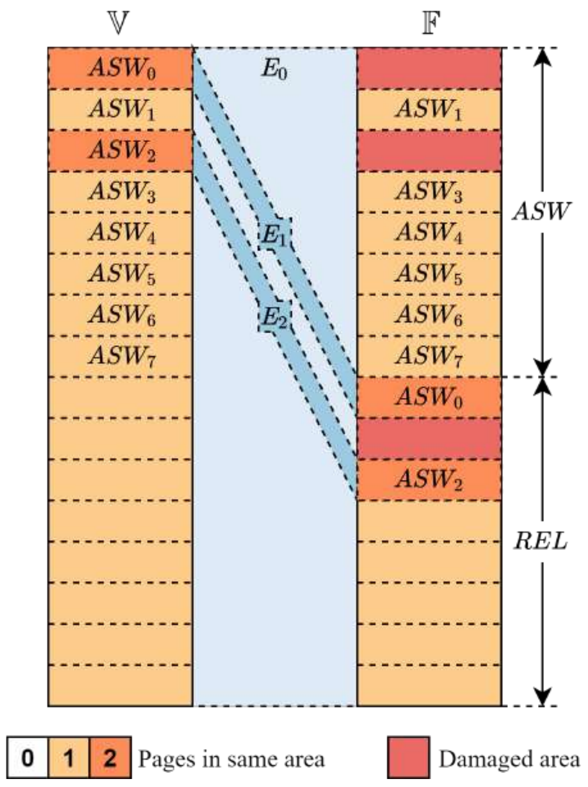


Fig. 4. Diagram showing the resulting projection of the BSW example using the proposed relocation algorithm.

free, it is marked as damaged, it does not select it and checks the next block.

- Checks that the third block of the relocation area (REL_2) is free and not damaged.
- Uses the third available TLB entry (E_2) to map the virtual page whose addresses match those of block ASW_2 to the physical frame corresponding to block REL_2 and marks this entry as overlapping.

- Checks the status of the rest of the ASW blocks (ASW_3, ASW_7) and verifies that none are marked as damaged.
- Since relocations have been necessary, enables the MMU and proceed to the ASW deployment phase.

The result after deployment will match what is shown in Figure 4. As can be seen, entry E_0 is used to perform a projection of the entire memory. In addition, the diagram also shows how blocks 0 and 2 of the application software ($\{ASW_0, ASW_2\}$) have been mapped through entries E_1 and E_2 respectively to the first relocation blocks that were not corrupted, i.e., REL_0 and REL_2.

V. Implementation and validation

As previously mentioned in the introduction, to validate the design of the proposed Memory Management Unit (MMU), we have performed a hardware implementation on the LEON3 processor [22]. These processors, based on the SPARC version 8 [11] architecture, define a 32-bit virtual and physical address space, with a memory cell size and access granularity of 1 byte. To develop the MMU, we have started from a generic design based on the one described in section III. Although not required, we opted to include access permission bits to facilitate the adaptation of the proposed MMU to the original LEON3 design. The following paragraphs detail the format of the TLB entries and control registers, the translation process, and the hardware/software interface.

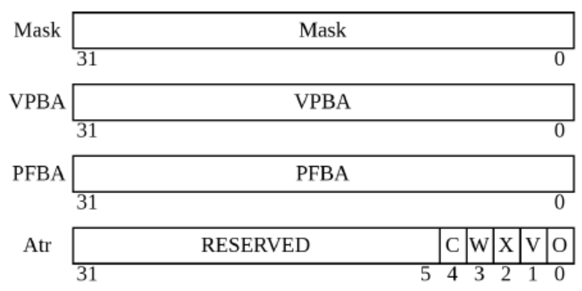


Fig. 5. Fields of the TLB entries for the implementation of the proposed MMU.

A. Format of the TLB entries

The TLB entries contain all the necessary attributes to perform the translation process. Their design follows the same approach proposed in the section III, incorporating new elements that add additional functionalities.

Figure 5 shows the format and the fields that form each of the entries. The semantics of each of these fields are described below:

- mask: this attribute is used to encode the size of the page defined by the entry. The binary value of this field will consist of one or more set bits followed by zero or more cleared bits. As detailed below, this value is used during the translation process as a mask that is applied to the virtual address. The total number of cleared bits corresponds to the page size defined by the entry (E_i^σ). For example, if the mask field of a given entry has a value of `0xFFFFF000`, the corresponding page size will be 12 bits, i.e., $E_i^\sigma = 12$.
- VPBA: this field is used to define the virtual page base address associated with the entry. The page number E_i^ν will be the result of applying a logical AND to the fields VPBA and mask and logically shifting the resulting value to the right by the number of bits given by the page size. Continuing with the previous example, if the field VPBA has a value of `0x40100000`, the page number E_i^{nu} will be equal to `0x40100`.
- PFBA: this field is used to define the physical frame base address to which the mapping of the virtual page is to be made. As with the previous field, the frame number E_i^μ will be the result of applying a bitwise AND to the attributes PFBA

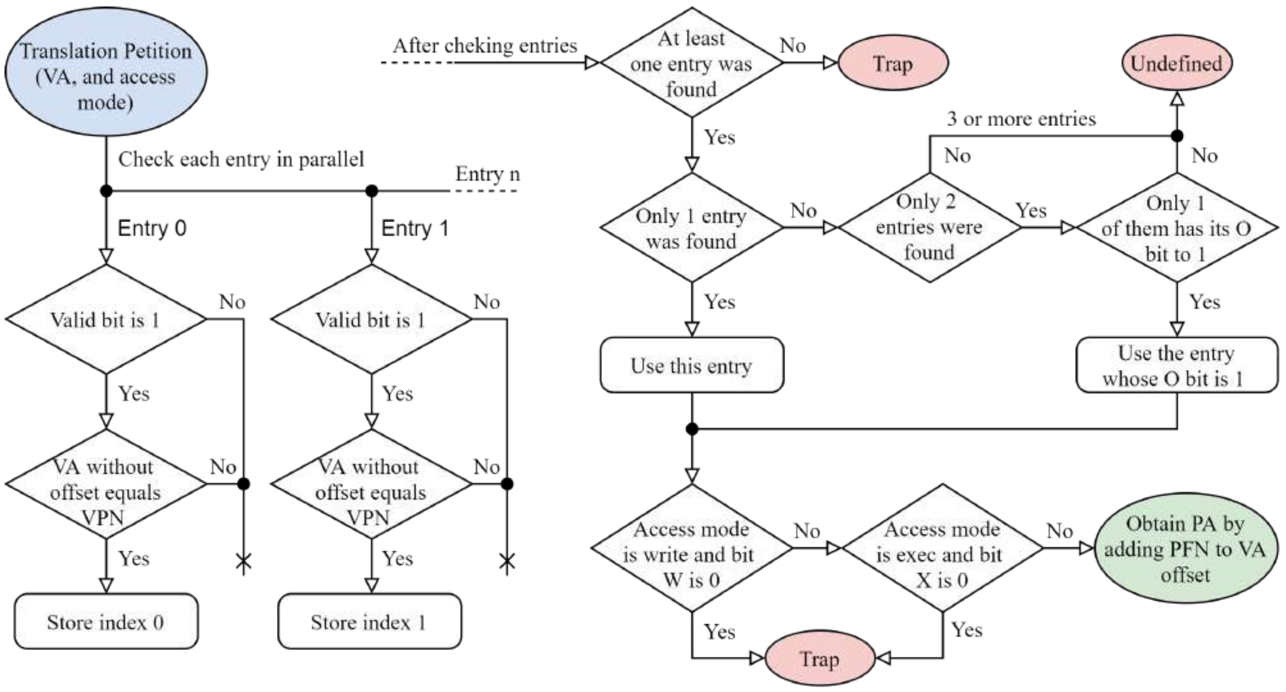

Translation Petition (VA, and access mode)
Check each entry in parallel
Entry 0
Entry 1
Entry n
Valid bit is 1
VA without offset equals VPN
Store index 0
Store index 1
Yes
No
After cheking entries
At least one entry was found
Trap
Only 1 entry was found
Only 2 entries were found
3 or more entries
Only 1 of them has its O bit to 1
Undefined
Use this entry
Use the entry whose O bit is 1
Access mode is write and bit W is 0
Access mode is exec and bit X is 0
Obtain PA by adding PFN to VA offset

Fig. 6. Diagram showing the translation process of the implementation for the proposed MMU.

and mask and logically shifting the resulting value to the right by the number of bits given by the page size. As an example, for a value of the PFBA field of `0x08010000` and the mask field of `0xFFFFF000`, the frame number E_i^{μ} would be equal to `0x08010`.

- C: indicates whether the contents of the page should be cached. If C $= 1$, the data or instruction to be read or written after the translation will be stored in the corresponding cache.
- W: this field is used to set the write permissions of the page. If W $= 1$, the contents of the cells in the page can be modified. Otherwise, the page is marked as read-only, and a write access to any of the cells in the page will cause an exception.
- X: indicates whether the memory cells of the page can contain executable instructions. If X $= 1$, the page is defined as executable and can be used to store instructions that can be fetched for execution. Otherwise, any access for instruction fetching triggers an exception.
- O: this field corresponds to the overlapping flag of the entry (E_i^o).
- V: this field corresponds to the validity attribute of the entry (E_i^{δ}). Thus, if V $= 1$, the entry is considered as valid. Conversely, if V $= 0$, the entry would be marked as invalid and, therefore, must be ignored during the translation process.

The hardware/software interface, described later in this section, allows configuring and modifying all the fields of the TLB entries.

B. Description of the translation process

Figure 6 shows a diagram of the translation procedure. In this case, the procedure includes, in addition to the virtual address, an additional argument which would be the access mode. Thus, three access modes are defined: read, write and execute. Read mode accesses are those in which the processor attempts to retrieve data from memory. On the other hand, read mode accesses occur when the processor wants to modify a piece of data stored in memory. Finally, execution mode accesses are those in which the processor tries to fetch an instruction from memory.

The first stage of the translation process corresponds to the function $valid_{\mathfrak{E}}$ and consists of searching for valid TLB entries whose associated pages contain the virtual address to be translated. This search is performed in parallel on all the entries present in the TLB. In this case, for a given address υ, an entry will be selected if and only if it is marked as valid, i.e., its V bit is set, and if the page associated with the entry contains υ. In order to make this decision, the unit uses the mask and VPBA fields. As mentioned above, the binary value of the mask field consists of one or more set bits followed by zero or more cleared bits, where the number of cleared bits corresponds to the size of the page. The field VPBA, on the other hand, contains the virtual page number shifted logically to the left as many bits as the page size. The most significant bits of this field will therefore correspond to the page number, and the number of these bits will be the same as the set bits of the mask field. Furthermore, by the very definition of a page, all addresses belonging to the page will have the same values in these most significant bits. Therefore, to decide whether or not a given virtual address belongs to the page defined by an entry, it is only necessary to check whether the bits corresponding to the page number of the address match those stored in the VPBA field. In order to do so, it is sufficient to apply a bitwise AND between the mask field of the entry and the virtual address and check whether the resulting value is equal to the one stored in the VPBA field, i.e., whether $\text{AND}(\upsilon, \mathsf{mask}) = \mathsf{VPBA}$.

If no entries are found after the selection procedure, the unit generates an exception trap. If only one entry has been found, this is the one that is finally selected for translation. Suppose more than one entry meets the criteria. In that case, the unit checks their overlapping flag following a procedure equivalent to that of the $atop_{\mathfrak{E}}$ function, described in Section III. In this way, from the set of selected entries, only those marked as overlapping, i.e., those with the O bit set, are filtered out. If only one overlapping entry is found, this will be selected to continue the translation procedure. If no entry is marked as overlapping, or if two or more entries are detected with the O bit set, and according to the formal description provided in Section III, the unit should exhibit undefined behavior. This condition is considered a configuration error, and it is the responsibility of the software to ensure that it cannot arise. If this situation occurs, the unit does not trigger any traps in our implementation but silently selects one of the entries and proceeds with the translation.

Once the entry has been selected, the memory management unit performs a permission check according to the access mode. In this way, the unit prevents programs from performing unallowed memory accesses and facilitates the implementation of mechanisms for detecting faults and, if necessary, isolating and recovering from errors resulting from them. The behavior of the unit in this step of the process is as follows:

- If the access mode is write and the W bit of the selected entry is cleared, the unit generates an exception trap.
- If the access mode is execution and the X bit of the selected entry is cleared, the unit generates an exception trap.

In any other case, the unit proceeds with the translation of the virtual address. First, and following a procedure similar to the one defined in Section III, the offset of the virtual address within the selected page is obtained using the mask field. As discussed above, the number of bits cleared from the mask field sets the size of the virtual page (and, by extension, the physical frame), while the set bits indicate how many bits of the virtual address correspond to the page number. All addresses in the same page have the same values for the most significant bits, i.e., those corresponding to the page number. For the same reason, the least significant bits of the address, as many as cleared bits in the mask field, will indicate the offset within the page. The unit applies a bitwise AND between the virtual address and the negated mask to obtain this offset. Thus, for a virtual address υ, the offset within the frame computes as $\text{offset}(\upsilon) = \text{AND}(\upsilon, \text{NOT}(\mathsf{mask}))$. This operation is equivalent to the modulo operation described in Section III.

TABLE II
Example of a TLB configuration of the proposed MMU

$\mathfrak{E}$	Mask	VPBA	PFBA	Atr				
				C	W	X	V	O
0	`0x80000000`	`0x0`	`0x0`	1	0	1	1	0
1	`0xC0000000`	`0x80000000`	`0xC0000000`	1	1	0	1	0
2	`0xF0000000`	`0x20000000`	`0x10000000`	1	0	1	1	1
3	`0xE0000000`	`0x40000000`	`0x60000000`	1	0	1	1	1
4	`0xF8000000`	`0x50000000`	`0x30000000`	1	0	1	1	1
5	`0xF0000000`	`0x90000000`	`0xA0000000`	1	0	0	1	0
6	`0x0`	`0x0`	`0x0`	0	0	0	0	0
7	`0x0`	`0x0`	`0x0`	0	0	0	0	0

The PFBA field contains the physical frame number shifted logically to the left as many bits as the page size. In this way, the value of this field corresponds to the initial address of the physical frame where the mapping is to be performed. Therefore, to obtain the physical address, it is only necessary to perform a bitwise OR operation between the offset obtained in the previous step and the value stored in the PFBA field, that is, $\text{OR}(\mathsf{PFBA}, \text{offset}(\upsilon)$. Since the bits in the PFBA field corresponding to the page size are set to zero, this operation is equivalent to the summation performed by the $phys$ function, described in Section III.

Table II shows the value of each of the fields and attributes of the TLB entries corresponding to the example introduced in Section III. Thus, entry E_1 maps a virtual page of size 2^{30} bytes, which in our implementation is achieved by defining a mask with the last 30 bits cleared. The VPBA field will contain the first address of the virtual page $(2 \cdot 2^{30})$, while the PFBA will contain the first address of the target frame $(3 \cdot 2^{30})$. The configuration of the rest of the entries follows a similar scheme. In this case, the page mapped by entry E_0, as well as the pages overlapping with it and mapped by entries E_2, E_3 and E_4, are code pages that contain instructions and therefore have only the execution permission set. On the other hand, the page mapped by entry E_1 contains data and allows both read and write access. This page is overlapped by the one mapped by entry E_5. In this case, the memory cells included in the latter page contain only constants, and therefore the page has been marked as read-only.

The translation mechanism will produce the outputs as described in Section III and will also perform the permission checking procedure depending on the type of access.

C. Hardware/software interface

The memory management unit provides the software with the following configuration and status registers:

- *Control register*. This register allows enabling and disabling the MMU. It also has a field indicating the number of installed and available TLB entires.
- *Fault Status Register (FSR)*. This is a read-only register that identifies the type of the last fault

that has occurred. The unit supports three types of failure: translation, write, and execution. The first occurs when the virtual address to be translated is not contained by any of the pages defined by the valid TLB entries. The other two errors are related to the access permissions of the pages. The write failure occurs when writing to a virtual page marked as read-only. The execution failure occurs when attempting to fetch an instruction in a page not marked as executable.

- *Fault Address Register (FAR)*. It is a read-only register that provides the virtual address whose access caused the last error.
- *TLB entry input/output registers*. It is a group of 4 registers that follow the same format as the TLB entries shown in Figure 5. They are used to read and write to and from the TLB entries. The use of these registers is described below.

The memory management unit maps these registers to an *alternate address space*. This mechanism, defined by the SPARC architecture, allows the implementation of address spaces different from the main space in which the memory and the registers of the input and output devices are mapped. The processor uses these spaces to provide different functionalities associated with its components. Some of these alternative address spaces are used, for example, to perform fine-grained management of the state of the different cache memories.

To access these spaces, the processor provides two specific instructions called `LDA` and `STA`. These instructions are used, respectively, to read or write a word to an address belonging to a specific alternate address space. The number of spaces is implementation-dependent, and each one is assigned a unique identifier called address space identifier (ASI). The LDA and STA instructions both have an 8-bit attribute corresponding to the ASI of the space they are to be applied. In our case, the MMU uses ASI 0x1A since this identifier does not interfere with any other identifier implemented by the LEON3 processor.

Each space has its address map, in which the corresponding component or subunit can associate to each address different registers or actions triggered by the read or write instructions. In the case of our implementation, we chose to define the actions and map the registers shown in Table III. The registers are selected using only bits 11 to 8 of the alternate space address and ignoring the value of the seven least significant bits. In this case, this limitation is analogous to that defined by the SPARC Reference MMU for accessing its registers [11]. This MMU is defined as part of the SPARC architecture itself and is supported by the base implementation of the LEON processor. In our case, we have chosen to keep this same limitation to facilitate interconnection and guarantee the compatibility of our solution with LEON processors.

The input/output registers are used to modify and read the contents of the different TLB entries. There is one register for each field. To modify the content of an entry, it is necessary to previously load the new value that we want each of the fields to have in the input/output registers. Once the new values have been loaded, to proceed with the update, it is necessary to perform a write on the memory address `0x000000xx` using the instruction `STA`. The value written to the memory address will correspond to the index of the TLB entry to be modified. When the instruction is executed, the values stored in the input/output registers will be stored into the corresponding fields of the selected entry.

TABLE III
Memory map of ASI `0x1A`

Address	Access type	Description
`0x000000xx`	Write-only	Updates a TLB entry with the contents of the input/output registers.
`0x000001xx`	Write-only	Loads the contents of a TLB entry into the input/output registers.
`0x000002xx`	Read-only	Performs a TLB probe
`0x000003xx`	Read/Write	Mask input/output register
`0x000004xx`	Read/Write	VPBA input/output register
`0x000005xx`	Read/Write	PFBA input/output register
`0x000006xx`	Read/Write	Attribute word input/output register
`0x000007xx`	Read/Write	Control register
`0x000008xx`	Read-only	Fault Status Register
`0x000009xx`	Read-only	Fault Address Register

The memory management unit also allows reading the values of the fields of a given TLB entry. In order to obtain these values, it is necessary to execute an instruction `STA` that writes to the address `0x000001xx`. Analogous to the previous case, the value written to the above address will correspond to the index of the TLB entry to be read. After the execution of the instruction, the input/output registers will contain the values of the corresponding fields of the selected entry.

Finally, the MMU provides a mechanism to check if a given address is part of a page defined by one of the TLB entries marked as valid. In this case, the input/output register corresponding to the VPBA field is used to store the virtual address to be checked. Once the address is loaded, it is necessary to execute an `LDA` instruction on the address `0x000002xx`. The value read in this case will correspond to the index of the TLB entry that would be selected should a translation of that address be performed. Also, as a side effect, the unit updates the input/output registers with the contents of the matching entry. If the translation process failed to locate any valid entry containing the address, the resulting read value would be `0xFFFFFFFF`. The input/output registers would not be updated in this second case. The checking mechanism does not take access permissions into account and would not generate an exception trap in any case.

D. Validation of the proposed approach

To validate the proposed solution, we have adapted the relocation algorithm described in Section A for the software of the instrument control unit (ICU) of the Energetic Particle Detector (EPD) of the Solar Orbiter

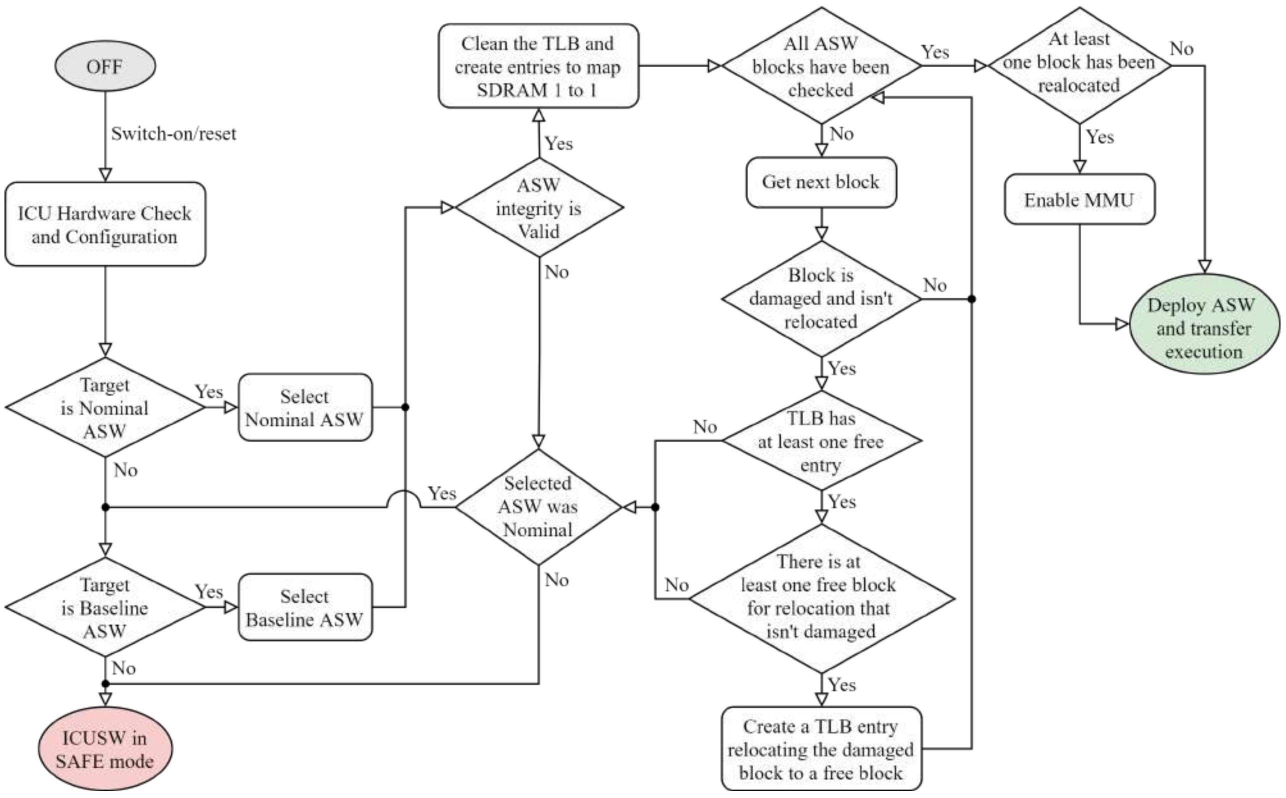

OFF
Switch-on/reset
ICU Hardware Check and Configuration
Target is Nominal ASW
Yes
Select Nominal ASW
No
Target is Baseline ASW
Yes
Select Baseline ASW
No
ICUSW in SAFE mode
ASW integrity is Valid
Yes
No
Clean the TLB and create entries to map SDRAM 1 to 1
Selected ASW was Nominal
Yes
No
All ASW blocks have been checked
Yes
No
Get next block
Block is damaged and isn't relocated
No
Yes
TLB has at least one free entry
No
Yes
There is at least one free block for relocation that isn't damaged
No
Yes
Create a TLB entry relocating the damaged block to a free block
At least one block has been realocated
No
Yes
Enable MMU
Deploy ASW and transfer execution

Fig. 7. Diagram of the deployment process and the proposed relocation algorithm.

mission. This software follows a scheme analogous to that described in Subsection IV. It consists of two programs: the start-up software (BSW), which takes control after each system reset and implements the Safe mode of the instrument, and the application software (ASW), which provides the Nominal mode in which the sensor units are activated, and supports the science generated by these units.

To increase the reliability of the system, the EEPROM stores two images of the ASW: one Baseline and one Nominal. The two images, in addition to providing a tolerance mechanism in the event that one of them is corrupted in its EEPROM storage, allow handling two versions of the ASW. The Nominal one is intended to contain the latest ASW update, while the Baseline would store an earlier, more stable version which, although it has proven to work properly in orbit, could not incorporate the latest modifications. The control of the ASW image deployed by the BSW depends on a set of status variables located in the EEPROM. These variables determine the enabling state of each image, determined both by the configuration commanded from the ground and by errors in previous executions. Depending on the value of these variables, the target ASW image to execute is determined, prioritizing the Nominal one in case both are enabled for deployment.

First, the BSW starts by performing the initial system configuration and hardware health check, including RAM. Next, it determines which of the two ASW images is enabled for deployment. If the Nominal image is enabled, it checks its integrity before deploying. If the integrity test fails, the BSW will disable it and try to deploy the Baseline image. If the Baseline image is disabled or its integrity test fails, the BSW would maintain control of the system, entering Safe mode.

In the original implementation [3], the BSW then checks the health of the RAM blocks on which the selected image is to be deployed. If any of the blocks has a permanent failure in one or more cells, it would not be possible to deploy the image, and, subsequently, it will be disabled. This original behavior has been modified to adapt it to use the new memory management unit.

Figure 7 shows an activity diagram modeling the new relocation algorithm. As already discussed, its final implementation follows a similar approach to that outlined in Section A. Thus, if the boot software detects a failure in one of the target memory blocks, it will use the memory management unit to relocate it by introducing an overlapping page mapped to a specific, error-free physical memory block. In the case of exhausting all available TLB entries or free blocks intended for relocation, the boot software will be forced to disable the image.

Two techniques were used to validate the implementation of the algorithm. The first one involved the use of a virtual platform called LeonViP [23], [24]. This platform simulates a System-on-Chip (SoC) based on a LEON processor, and it was initially implemented to support the development and validation of the software of the EPD's ICU. LeonViP allows simulated memory errors to be injected to facilitate the validation of error

TABLE IV
Resources and Power consumption in the Microchip RTAX FPGAs.

	No MMU	SRMMU			Proposed MMU		
Resource\TLB#	-	4	8	16	4	8	16
Sequential (R-cells)	4,317	5,261	5,644	6,047	5,168	5,570	6,427
Combinational (C-cells)	14,832	16,046	16,588	17,260	16,521	17,385	18,580
RAM/FIFO	39	39	39	40	39	39	39
Power (mW)	341.71	351.06	355.80	362.47	355.99	361.79	374.60

TABLE V
Resource and Power consumption in the Microchip RT ProASIC FPGAs.

	No MMU	SRMMU			Proposed MMU		
Resource\TLB#	-	4	8	16	4	8	16
Core Tiles	28,653	31,488	32,369	34,159	32,627	34,569	39,178
RAM/FIFO	61	63	63	63	61	61	61
Power (mW)	566.09	591.57	599.69	618.30	601.43	620.52	662.79

TABLE VI
Resource and Power consumption in the Microchip RTG4 FPGAs.

	No MMU	SRMMU			Proposed MMU		
Resource\TLB#	-	4	8	16	4	8	16
4-LUTs	14,214	15,691	16,098	16,897	15,346	15,961	17,188
DFFs	5,707	6,253	6,526	7,080	6,256	6,666	7,473
RAM1K18	6	8	8	8	8	8	8
RAM64x18	49	43	43	43	41	41	41
Power (mW)	265.64	269.41	270.81	273.44	268.79	270.85	274.79

TABLE VII
Resource and Power consumption in the Xilinx Virtex VQV space-grade FPGAs.

	No MMU	SRMMU			Proposed MMU		
Resource\TLB#	-	4	8	16	4	8	16
Slices	3,539	4,205	4,478	5,034	4,271	4,679	5,496
Flip-flops	10,123	11,443	11,866	12,791	11,422	12,337	14,025
LUTs	9,031	10,042	10,233	10,661	9,953	10,464	11,349
BRAMs	9	9	9	9	9	9	9
Power (W)	2.119	2.121	2.122	2.123	2.121	2.122	2.124

detection and correction algorithms. The original platform has been modified in the context of this work to include the simulation of the proposed memory management unit.

In order to conduct the first validation technique, we used a regular PC with the operating system Ubuntu and the Eclipse development environment. After compiling the modified version of LeonViP that implements the proposed MMU, we run the BSW image that has been modified to implement the relocation algorithm and manually inject errors in memory areas reserved for the deployment of the ASW. During execution, we verified that the damaged areas were relocated to new, non-damaged memory areas previously reserved for that purpose. The correct deployment and execution transfer to the ASW was also verified.

The second validation technique follows a similar idea but is performed directly on the final hardware. In this instance, the validation scenario consists of an FPGA that has been programmed using the modified version of the LEON3 core that implements the proposed MMU. We also include other cores needed for the correct operation and to help the debugging process, namely: AHB debug UART, JTAG debug link, Memory Controller, AHB/APB bridge, Debug Support Unit, one generic UART, Interrupt unit, one Modular Timer unit and one General Purpose I/O port. Using the hardware monitor GRMON and the Debug Support Unit, we flash the PROM with the same image of the BSW as in the first validation, which includes the relocation algorithm and injects errors in areas reserved for the ASW deployment. The error injection is achieved by directly modifying the data structure used by the error detection algorithm to indicate which RAM areas are valid. We use the UART to obtain specific outputs and the debug unit to check the hardware status, verifying the same behavior as with the first validation technique. i.e., all damaged areas were relocated to new, non-damaged memory areas, and the execution was transferred to the ASW after its successful deployment.

E. Synthesis results

We have synthesized the LEON processor with the implemented Memory Management Unit (MMU) on the leading space-qualified Xilinx and Microsemi FPGAs. The SoC includes the same cores described in the second validation scenario in Section D. The processor also deploys a data cache with one set, 4-Kbyte per set and 32 bytes per line, and an instruction cache with two sets, 4-Kbyte per set and 32 bytes per line. Both caches implement a Least Recently Used (LRU) replacement algorithm.

Tables IV, V, VI and VII show the results obtained with the original version of the processor without MMU, with the SPARC Reference MMU (SRMMU), and those obtained after deploying our unit in terms of consumed resources and estimated power consumption for the evaluated FPGAs using a clock frequency of 24 MHz. For the systems deploying MMUs, the most significant resource consumptions for different numbers of TLB entries are shown for each family. Thus, for the Xilinx Virtex VQV space-grade qualified FPGAs, Table VII shows the consumed slices, flip-flops, LookUp Tables (LUTs) and Block RAMs (BRAMs). For the Microsemi RTG4 FPGAs, Table VI lists the required 4-input LookUp Tables (4-LUTs), the D-type flip-flops (DFFs), and the RAM1K18 and RAM64x18 memory blocks. Table IV shows the consumption for the Microsemi RTAX FPGAs in terms of Sequential (R-cells) and Combinational (C-cells) logic cells and RAM/FIFO memory blocks. Finally, for the Microsemi RT ProASIC FPGAs, Table V shows the required core tiles and RAM/FIFO memory blocks.

The results show an increase in resource and power consumption for the proposed MMU comparable to that of the SRMMU. In both cases, it can be observed that the increase is practically linear with the number of TLB entries for both MMUs. It has been found that the increase in consumption of the proposed MMU compared to the SRMMU is mainly due to the increased amount of data that each TLB entry stores in addition to the extra ports needed for the communication to allow reading and writing each entry. This can be optimized by serializing the writing and reading process and limiting page sizes if they are found unnecessary. Furthermore, for the case of the proposed MMU, the number of available TLB entries has a direct impact on the damaged block relocation capabilities and thus shall be chosen, in all cases, depending on the particular parameters of the mission.

The main objective of our proposal is to provide an MMU to avoid permanent errors in systems that do not require the use of dynamic memory and, therefore, do not need any form of virtual memory management. In such systems, the entire memory can be mapped using only two pages: a cacheable page that maps the entire address space, including the application's data and code, and an overlapping non-cacheable page to map the I/O device registers.

Given the low rate of permanent memory failures and the possibility of combining our solution with other fault tolerance techniques, we have concluded that it is unnecessary to implement a configuration that deploys more than eight TLB entries. In that case, and needing two of them to map the entire memory address space, six entries would remain to avoid at least six permanent memory failures.

The SoC implemented for the instrument control unit of EPD was deployed into a Microchip RTAX FPGA. The design did not implement an MMU. This SoC had a resource consumption of 8,158 R-cells, 20,990 C-cells, and 31 RAM/FIFO. Based on the results obtained for the 8 TLB entry configuration of our MMU implemented in an FPGA of the same family, shown in the Table IV, we found a nominal increase of 1,253 R-cells, 2,553 C-cells, and 0 RAM/FIFO when compared to not implementing MMU. Compared to the overall resource consumption of the SoC of the EPD's instrument control unit, this represents a resource increase of 15.3% R-cells, 12.2% C-cells, and 0% RAM/FIFO.

F. Performance analysis

In order to analyze the impact on the overall software performance, we have used the Dhrystone [25] benchmark. This benchmark is implemented as a single-task program whose original code has been modified to add the necessary methods to configure and activate the MMU. The program has been compiled using the BCC [26] suite and configured with a value of 100,000 iterations.

To measure the degree of overhead introduced by the memory management unit, we have run the test on two different configurations of the LEON architecture. In the first one, the system does not implement any memory management unit or translation mechanism and, therefore, programs are executed directly on physical memory. In the second configuration, the system deploys the memory management unit presented in this article. In this second case, the system is configured using a single TLB entry that maps 1 to 1 the entire virtual address space onto physical memory.

The results obtained are shown in Tables VIII and IX. The former corresponds to the benchmark execution with the caches disabled, while the latter shows the results using the caches. In the latter case, the system has been configured with an 8 KiB, 8-ways instruction cache, and an 8 KiB, four-ways data cache.

In both cases, the execution results on the original LEON processor-based system (No MMU) and the one including the proposed memory management unit are shown. Both tables show the results as a function of the optimization level set in the test program compilation process. These results include the number of instructions executed and the number of cycles consumed, which have been obtained from an L3STAT statistics unit deployed together with the processor. From this data, the average number of cycles per instruction is derived. In addition, the tables show the time in seconds taken to execute all iterations of the loop, the average iterations per second (Dhry/s), and the so-called Dhrystone MIPS (DMIPS). These data are obtained programmatically by the test itself. In the case of execution with MMU, the increase of cycles consumed with respect to the execution without MMU is also included.

As can be seen, the translation process causes an increase in the number of cycles per instruction. The reason for this increase is that the memory management unit, in its current implementation, consumes an additional cycle to perform the address translation. The results also show that if the system has the caches enabled, the increase in cycles is smaller. This is because the translation process and the cache lookup process are performed in parallel. This search process also takes one cycle to execute. Thus, if a read access provokes a cache miss, the final physical memory address is available at the beginning of the necessary memory access operation.

In order to contextualize the performance impact produced by the proposed MMU, we have also measured the overhead introduced by the SPARC Reference MMU (SRMMU). In order to do so, we used the same Dhrystone benchmark, adding the initialization of the page tables needed for the SRMMU. Since the performance impact derived from the use of the SRMMU varies widely depending on how often the page tables are consulted, we focused on the best and worst cases. The best case corresponds to the one where all required pages are present at all times in the TLB. In this scenario, the performance is identical to that observed with the proposed MMU. The worst case would be one in which none of the target pages were previously present in the TLB, so it would be necessary to traverse the page tables to locate the physical address for every single memory access. In order to emulate this case, we ran the benchmark with the TLB disabled, finding a 140% increase in the number of cycles compared to not using the MMU.

Although the average increase in execution times may result in a performance loss of about 11%, we believe

TABLE VIII
Results of the Dhrystone benchmark using 100,000 iterations with caches disabled

Mode	Optimization	Instructions	Cycles	Time (s)	Dhry/s	DMIPS	CPI	% increase in cycles
No MMU	O0	101,900,553	839,703,891	21.0	4,763.6	2.7	8.240	-
	O1	53,700,524	392,209,110	9.8	10,198.6	5.8	7.304	-
	O2	37,400,533	305,375,326	7.6	13,098.6	7.5	8.165	-
	O3	37,100,533	303,228,718	7.6	13,191.4	7.5	8.173	-
Proposed MMU	O0	101,900,553	948,554,265	23.7	4,216.9	2.4	9.309	12.96%
	O1	53,700,524	446,292,996	11.2	8,962.7	5.1	8.311	13.79%
	O2	37,400,533	342,680,621	8.6	11,672.7	6.6	9.162	12.22%
	O3	37,100,533	342,104,216	8.6	11,692.3	6.7	9.221	12.82%

TABLE IX
Results of the Dhrystone benchmark using 100,000 iterations with caches enabled

Mode	Optimization	Instructions	Cycles	Time (s)	Dhry/s	DMIPS	CPI	% increase in cycles
No MMU	O0	101,900,553	148,326,753	3.7	26,967.5	15.3	1.456	-
	O1	53,700,524	77,990,033	1.9	51,288.6	29.2	1.452	-
	O2	37,400,533	59,647,878	1.5	67,060.2	38.2	1.595	-
	O3	37,100,533	56,550,265	1.4	70,733.5	40.3	1.524	-
Proposed MMU	O0	101,900,553	164,514,098	4.1	24,314.0	13.8	1.614	10.91%
	O1	53,700,524	85,763,764	2.1	46,639.7	26.5	1.597	9.97%
	O2	37,400,533	66,394,151	1.7	60,246.3	34.3	1.775	11.31%
	O3	37,100,533	62,800,889	1.6	63,693.3	36.3	1.693	11.05%

that the benefits provided by the translation mechanism in facilitating the relocation of damaged blocks of memory are sufficient to justify its use in on-board space environments.

VI. Conclusions and future work

This paper presents the design and implementation of a memory management unit that allows the dynamic relocation of on-board space software. This capability allows avoiding possible damaged memory blocks due to radiation without needing to apply modifications to the software that would require intervention from ground mission control.

The memory management scheme presented in this paper combines the benefits of using large memory blocks while allowing fine tuning in the re-mapping of memory locations with permanent errors. It implies a small memory loss in terms of internal fragmentation compared to paging schemes where the entire page should be discarded. In addition, the use of software-managed TLBs eliminates the use of memory-resident translation tables, thus avoiding the problems associated with errors in the translation tables themselves.

The unit has been implemented on the LEON architecture. The validation of the proposed solution has been carried out using as a base the boot and application software of the instrument control unit of the Energetic Particle Detector of the Solar Orbiter mission. The results of the implementation show an assumable increase in FPGA resource requirements, as well as a minor loss of performance due to the need to perform address translation. These two factors are clearly offset by the advantages of being able to relocate code and data in case of permanent memory errors.

Future work includes the study of possible modifications to the implementation of the unit to improve its temporal and spatial performance. For this purpose, we will study the possible improvements derived from integrating the proposed mechanism with other elements such as the instruction pipeline or the cache memory.

ACKNOWLEDGMENT

The authors would like to thank Professor Rafael Sendra for his invaluable help in formalizing the specification of the behavior of the proposed memory management unit.

REFERENCES

[1] G. Hubert, S. Aubry, and J. A. Clemente, “Impact of ground-level enhancement (gle) solar events on soft error rate for avionics,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 56, no. 5, pp. 3674–3684, 2020.

[2] E. Petersen, *Single event effects in aerospace*. John Wiley & Sons, 2011.

[3] Ó. R. Polo *et al.*, “Reliability-oriented design of on-board satellite boot software against single event effects,” *Int. J. High Perform. Syst. Archit.*, vol. 114, p. 101920, Mar. 2021.

[4] Space Avionics Open Interface Architecture (SAVOIR), *SAVOIR-GS-002 - SAVOIR Flight Computer Initialisation Sequence Generic Specification*, SAVOIR, ESA.

[5] A. Witze, “Software error doomed japanese hitomi spacecraft,” *Nature*, vol. 533, no. 7601, pp. 18–19, May 2016. [Online]. Available: https://doi.org/10.1038/nature.2016.19835

[6] P. J. Denning, “Virtual memory,” *ACM Comput. Surv.*, vol. 2, no. 3, p. 153–189, sep 1970. [Online]. Available: https://doi.org/10.1145/356571.356573

[7] S. Feizabadi, B. Ravindran, E. E. Fox, E. D. Jensen, J. D. Arthur, and R. E. Nance, “Dynamic memory management in a resource-constrained real-time utility accrual environment,” 2004.

[8] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, Intel Corporation, Jun. 2021.
[9] *AMD64 Architecture Programmer's Manual - Volume 2: System Programming*, Advanced Micro Devices, Mar. 2021.
[10] *Arm Architecture Reference Manual - Armv8, for A-profile architecture*, Arm Ltd., Jul. 2021.
[11] *The SPARC Architecture Manual - Version 8*, SPARC International Inc., 1992.
[12] *The RISC-V Instruction Set Manual - Volume II: Privileged Architecture*, University of California, Berkeley, Jun. 2019.
[13] *MIPS Architecture For Programmers - Vol. III: MIPS32 / microMIPS32 Privileged Resource Architecture*, MIPS Technologies, Jul. 2015.
[14] S. Sánchez *et al.*, "HW/SW co-design of the instrument control unit for the energetic particle detector on-board solar orbiter," *Adv. Space Res.*, vol. 52, no. 6, pp. 989–1007, Sep. 2013.
[15] J. Rodríguez-Pacheco *et al.*, "The energetic particle detector - energetic particle instrument suite for the solar orbiter mission," *Astron. Astrophys. Suppl. Ser.*, vol. 642, p. A7, Oct. 2020.
[16] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of radiation effects in sram-based fpgas for space applications," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, pp. 1–34, 2015.
[17] A. J. Olazábal and J. P. Guerra, "Multiple cell upsets inside aircrafts. new fault-tolerant architecture," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 55, no. 1, pp. 332–342, 2018.
[18] B. Swiercz, D. Makowski, and A. Napieralski, "A novel approach for operating systems protection against single event upset," in *Proceedings of the International Conference Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006.* IEEE, 2006, pp. 61–64.
[19] X. Zhou and P. Petrov, "The interval page table: virtual memory support in real-time and memory-constrained embedded systems," in *Proceedings of the 20th annual conference on Integrated circuits and systems design*, 2007, pp. 294–299.
[20] M. Böhnert and C. Scholl, "A dynamic virtual memory management under real-time constraints," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications.* IEEE, 2014, pp. 1–10.
[21] C. Meenderinck, A. Molnos, and K. Goossens, "Composable virtual memory for an embedded soc," in *2012 15th Euromicro Conference on Digital System Design.* IEEE, 2012, pp. 766–773.
[22] *GRLIB VHDL IP Core Library - GRLIB IP Core User's Manual*, Jul. 2021.
[23] A. da Silva, S. Sánchez, O. R. Polo, and P. Parra, "Injecting faults to succeed. verification of the boot software on-board solar orbiter's energetic particle detector," *Acta Astronautica*, vol. 95, pp. 198 – 209, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0094576513003962
[24] P. Parra, A. da Silva, O. R. Polo, and S. Sánchez, "Agile deployment and code coverage testing metrics of the boot software on-board solar orbiter's energetic particle detector," *Acta Astronautica*, vol. 143, pp. 203 – 211, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0094576516311614
[25] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.
[26] *BCC User's Manual*, Cobham Gaisler AB, Sep. 2020.

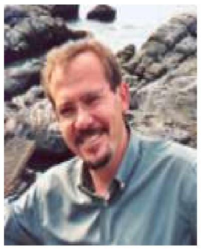
Borja Losa received his B.S. degree in Computer Engineering from the University of Alcalá in 2021, and his M.S. degree in Space Science and Technology in 2022. Since 2019, he has been participating with the Computer Engineering department and the Space Research Group of Universidad de Alcalá working on different projects focused on the SPARC architecture and LEON3 processor, researching new ways to improve the reliability for space systems. He is currently pursuing a Ph.D. in Space Research and Astrobiology aimed at facilitating the virtualization of the LEON3 processor by providing hardware assisted virtualization.

Pablo Parra received his Ph.D. in Information and Communication Technologies from the University of Alcalá in 2012. Since 2006, he has been working with the Computer Engineering Department and the Space Research Group (SRG) of the University of Alcalá. His research interests include component-based software engineering and model-driven engineering applied to the field of real-time embedded systems. He has taken part in numerous research projects in the field of on-board satellite software development, such as the NANOSAT programme, Solar Orbiter and Euclid.

Antonio Da Silva Born in Germany, Antonio da Silva has received the MSc and Ph.D. degrees in computer engineering from Universidad de Alcalá (UAH), in 2001 and 2015, respectively. He has worked as embedded software developer for primary radar systems and since 1991 he lectures in the field of Computing Science. He has worked with the Space Research Group of Universidad de Alcalá on projects related to the development of the energetic particle detector (EPD) on-board Solar Orbiter. His research interests include Serious Games for Education, Fault Tolerant systems design and critical software early development and verification.

Óscar R. Polo received his M.S. degree in physics from the Universidad del Pais Vasco in 1994 and his Ph.D. in physics from the Universidad Complutense de Madrid, Spain, in 2003. Since 2004, he has been with the Computing Engineering Department at the University de Alcalá. He is currently an assistant professor of embedded and real-time systems. His research interests include computer architecture, satellite on-board software, model driven engineering and embedded real-time systems. He has actively participated in several research projects in the area of computer engineering of satellite platforms in NANOSAT-01, NANOSAT-1B, Solar Orbiter and Euclid missions.

J. Ignacio G. Tejedor is a M.Eng. in Telecommunications and Ph.D. from the University of Alcalá (UAH). He’s assistant professor at the Computer Engineering Department of UAH since 1999 in the areas of operating systems and computer architecture, and a researcher in these fields plus in embedded systems, reconfigurable logic, data acquisition systems and nuclear instrumentation. He has actively participated in relevant research projects both in the field of onboard hardware and software for space applications in relation to missions such as Nanosat, Microsat or Exomars, and in the field of radiation detectors for the study of cosmic rays and solar activity, such as CaLMa (the Castilla-LaMancha Neutron Monitor) and ORCA (the Antarctic Cosmic Ray Observatory of UAH). In this context, he has participated in the XXXII, XXXIII and XXXV Spanish Antarctic Campaigns from 2018 to 2022, on the BAE Juan Carlos I on Livingstone Island (Antarctica).



Agustín Martínez received the M.S. degree from the Universidad Politécnica de Madrid (UPM) in 1986 and the Ph.D. degree from the Universidad de Alcalá in 2001. Professor of Computer Engineering Department at the University of Alcalá in Spain, Head of Department from 2010 to 2016. His research and teaching activities are in the areas of DSP, Computer Architecture, Embedded Systems and Space Systems. He has been in charge of technical and management issues at Telettra, Alcatel and Alcatel-Lucent over more than 20 years, accounting a deep experience over technical and management leadership activities in international projects.

Jonatan Sánchez received his Ph.D in Space Research and Astrobiology from the University of Alcalá in 2022. He worked for the Space Research Group since 2019, where he researched about Space Software testing, Fault Tolerance techniques and Hardware Simulation. Since 2021 he works on Cadence Design Systems Inc. developing Electronic Design Automation Verification tools.

Sebastián Sánchez received his M.S. degree from the Universidad Politécnica de Madrid in 1994 and his PhD. at the Universidad de Alcalá, in 1998. Since 1990, he has been with the Computing Engineering Department at the University de Alcalá. He is full professor of operating systems and computer architecture. His research interests include space instrumentation, embedded real-time systems, and mobile robots. He has actively participated in several national and international research projects in the area of hardware and software on board satellites such as SOHO, PHOTON, FUEGO 2, NANOSAT, Exomars, MICROSAT, Solar Orbiter and Euclid.

David Guzmán received his M.S. in Information and Communications Technology from the University of Alcalá in 2008 and his Ph.D in 2012. His research interests include computer architecture, embedded real-time systems, system on chip development, on-board technologies, and fault tolerant systems. He has actively participated in several international research projects in the area of hardware and software for on-board satellites, such as INTA Microsat, NASA FireStation, Firefly, Restore-L and JPSS.