

A Real-Time PES Supporting Runtime State Restoration after Transient Hardware-Faults

Martin Skambraks

FernUniversität in Hagen
Dept. of Electrical and Computer Engineering
58675 Hagen, Germany
martin.skambraks@fernuni-hagen.de

Abstract

Controlling safety-critical real-time applications that cannot immediately be transferred to a safe state requires highly reliable Programmable Electronic Systems (PESs). This demand for fault-tolerance is usually satisfied by applying redundant processing structures inside each PES and, additionally, configuring multiple PES redundantly. Instead of minimising the failure probability of single PESs, it is also desirable to provide a redundant configuration of PESs with the capability to re-start single units at runtime. This requires copying a PES's internal state at runtime, since a re-started unit must equalise its internal state with that of its redundant counterparts before the redundant processing can be rejoined. As a result, redundancy attrition due to transient faults is prevented, since failed channels can be brought back on line. This article states the problems concerned with runtime state restoration of real-time systems, discusses the advantages and disadvantages of existing techniques and introduces a hardware-supported state restoration concept.

1. Introduction

When *Programmable Electronic Systems (PESs)* have to be applied in highly safety-critical applications, 'availability' becomes one of the most significant performance measures. The term refers to the probability that a system is, at a predefined point in time, in an error-free state. In applications requiring safety licensing in accordance with the safety standard IEC 61508, it is usually not sufficient to increase availability solely

by minimising the failure rate of the built-in components. Since hardware failures are not totally avoidable – only their probability can be minimised –, it is also necessary to apply fault-tolerance techniques that ensure continuation of operation in case component failures. Almost all fault-tolerance techniques base on the principle of redundancy, e. g., on the multiple existence of functionally or characteristically equivalent objects.

In many real-time applications the regular operation can not at all, or only with unreasonably high effort, be interrupted. An example is the control electronics of an aeroplane, which can only be turned off and re-started on the ground. In case one unit of an n -fold redundant PES configuration fails during a non-interruptible processing phase, the safety of the entire system is impaired, as only $(n - 1)$ -fold redundancy remains. Remedying this redundancy attrition during regular operation is problematic, since restarting a single unit requires to harmonise its internal state with the ongoing redundant processing. Due to this, even quickly removable failure reasons or transient hardware failures, which transfer a PES to a faulty state, can cause long enduring impairment of safety. This problem is of growing importance, since the ever-increasing manufacturing density of integrated circuits causes higher sensitivity to temporary disturbances like electromagnetic noise or alpha rays.

Conventionally, the redundancy attrition problem is solved by configuring as many PES units redundantly as necessary to decrease the probability – that the number of PES units failed during a non-interruptible processing phase does not allow to maintain normal operation – below the maximum tolerable risk. In order to reduce the number of required PES units to an acceptable quantity, redundant processing structures are

also applied inside each PES to minimise the failure probability of single units. An other approach to increase the availability of a single unit is to provide a redundant configuration of PESs with the capability to re-start units that are in a faulty state, e. g., due to a transient hardware failure or unit replacement, without the need to interrupt real-time processing. This approach, which is subsequently referred to as *State Restoration at Runtime*, has various benefits and is the subject of this article.

The main body of this paper is structured as follows. Section 2 states the main problems associated with restoring the state of real-time PESs. Existing techniques are discussed in Section 3. Section 4 introduces the architecture of the PES on which the state restoration concept proposed in section 5 builds up. The conclusion at the end summarises the main aspects of this article.

2. State Restoration at Runtime

Restoring the state of a failed PES at runtime requires a redundant configuration of uniformly operating PESs. Each PES must be able

- to detect processing errors by comparing its own results with the results of the redundant counterparts,
- to drop out of the redundant operation in case of processing errors,
- to copy the internal state of the redundant units at runtime, and
- to rejoin the redundant operation when state equivalence has been reached.

This approach covers all failures caused by temporary disturbances and enables to replace defect PESs at runtime. The problem is that, while a PES equalises its internal state to that of the running counterparts, the latter continuously change their internal state.

Comparable problems have already been solved for many IT-applications. RAID controllers, for instance, can restore the data of replaced hard disks in the background by using the periods between disk accesses for copying [7]; state changes of the source disks can directly be taken into account by forwarding the corresponding write accesses to the replaced disk. Unfortunately, state restoration of real-time systems is more complex. Since they must be capable of handling several simultaneous events concurrently, the data modifications (= state changes) that are important for copying the internal state can be categorised into two

classes: *Program-controlled* and *Event-controlled Data Modifications* (PDMs and EDMs).

The PDMs comprise all data modifications the application software induces during execution. The volume of PDMs can easily be limited, e. g., by restricting the number of write accesses permitted within a specified time-frame. Appropriate compiler directives are easily realisable. Of course, the restriction to a number that allows continuous transfer of information about PDMs strongly limits the computing performance achievable. The latter, however, is not the major concern in terms of safety, and the high bandwidths of modern transfer technologies allow for performance more than sufficient for most highly safety-critical applications.

Limiting the volume of EDMs by an upper bound is more problematic. Since digital systems always operate in discrete time steps, several events can fall within one clock cycle and, thus, virtually occur simultaneously. Moreover, a single event can cause – in comparison with the program-controlled situation – a relatively large number of data modifications within an extremely short period of time. For example, in principle it is possible that one single event activates all tasks of an application software at once. In this case, storing the activation time for each task would result in a huge amount of EDMs. Obviously, restricting the frequency of write accesses limits the minimum realisable response time strongly. Thus, the capability to transfer all information about EDMs at runtime necessitates a trade-off between restricting the frequency of EDMs to a volume that allows acceptable real-time performance and the transfer bandwidth required. That is why in real-time PESs, which must provide both practicable computational performance and very short response times, more sophisticated methods need to be applied to enable copying the internal state at runtime.

3. Existing Techniques

The problem of state restoration at runtime has been widely treated in literature, and the proposed solutions can be categorised into hardware- and software-oriented methods. Unfortunately, almost all techniques proposed in literature are very application-specific [4]. This is especially true for the hardware-oriented methods.

One of the most well-known hardware-oriented techniques, which has been developed at the *Charles Stark Draper Laboratory* in Cambridge, is presented in [1], [2] and [8]. The method bases on a cyclically operating processing unit. A dedicated logic circuit generates characteristic signatures for all write accesses to

the data memory and stores them separately. Comparing the signatures of the current and the previous cycle enables to determine the modified data words. For this, a special hierarchical order of the signatures decreases the computing effort and, thus, accelerates the comparison. At the end of each cycle, each processing unit determines the modified data words and transfers them to its redundant counterparts. For this, a portion of the transfer bandwidth is reserved to transfer a fixed number of data words at the end of each cycle. This number restricts the maximum number of data modifications in a cycle. During the processing, the signatures are also transferred to the redundant processing units and enable the detection of processing errors by majority voting. Recovery can then be accomplished by restoring only those segments which have been corrupted as designated by the signatures. If the corrupted segments belong only to the data words that were modified within the current cycle, recovery can be completed at once (*One Shot Recovery*). Otherwise, the signature technique can also be used to restore the memory content incrementally. In case the exchanged amount of data is not completely needed to transfer the current data modifications, the remaining data words are used for incremental transfer of the memory content (*Incremental Recovery*).

This technique imposes only minor constraints on the application software. The restriction to a cyclic software execution is acceptable, since this operating principle is applied in many safety-related PESs because of its advantages in terms of safety licensing [9]. Disadvantageous is that – at the end of each cycle – the modified data words need to be identified by a comparison. An access logic that only stores the addresses of the write accesses would be much easier to implement and – as long as multiple write accesses to one address are avoided by storing intermediate values in unprotected memory regions – directly hint to the modified data.

Some software-oriented state restoration methods, like the ones presented in [3] and [6], found on the recovery block approach proposed first in [5]. Since they usually make use of ‘*recovery points*’ to which they can go back in case of failures, these strategies are commonly referred to as ‘*roll-back recovery*’.

Like the formerly discussed method, the method described in [3] also bases on cyclic software execution, but distinguishes between the main-cycle and sub-cycles. The duration of the main-cycle is a common integer multiple of the sub-cycle durations and all sub-cycles begin simultaneously at begin of each main cycle. Each computing task is assigned to one of the sub-cycles or the main-cycle. Each time a task withdraws

its processing till the end of the associated cycle, it stores special recovery data. In case a processing error occurs during the task’s next execution, these recovery data enable a restart at the *recovery point*. The restriction to cyclically synchronous software execution allows to take data dependencies (precedence relations) into account.

The techniques founded on recovery blocks have the advantage that recoveries can completely be conducted internally; it is not necessary to transfer data between redundant units. This prevents computing performance to depend on transfer bandwidth. A drawback is that merely isolated errors can be corrected; a complete state restoration, as it is necessary to replace units at runtime, is not possible. Even if the task administration functions are unalterably stored in a read-only memory, their execution can be unrecoverably affected by processing errors inside a processor. Moreover, not all errors can be detected (e.g., errors leading to plausible results), and the fact that faulty execution of a task might alter the data of other tasks is not taken into account (*Domino effect*). Thus, these techniques do not cover all error types and are inappropriate for applications of highest safety criticality.

In [4], two variants of software-oriented state restoration are presented, which also found on cyclic software execution. Both variants re-start units affected by a processing error and, then, transfer data between redundant units to completely restore the internal state. The first variant transfers within every cycle all current data modifications and, additionally, within a number of successive cycles the entire memory content in fractions. The second variant assigns to each data word an *Identification Bit*. A data word is only transferred to a faulty unit, if the associated identification bit is set. If a unit is informed about a re-start of one of its counterparts, it sets all identification bits. Subsequently, an identification bit is set when the data word is modified, it is reset when the data word is transferred. The state restoration is complete when all identification bits are in the reset state.

The first variant requires less implementation effort and completes state restoration within a predictable time-frame, but the immediate transfer of a cycle’s data modifications causes an unacceptable high dependence of the maximum achievable performance on the transfer bandwidth. The second variant attains the same computational performance with a lower transfer bandwidth, but it is problematic to predict the time required for state restoration. Besides, identifying the data words whose identification bits are reset causes additional computational effort, and – as long as not carried out in parallel to the task processing – decreases

the performance achievable.

4. The Dedicated PES Architecture

The hardware-assisted state restoration method proposed here finds on the PES-architecture presented in [9]. This architecture, which was particularly devised for safety-critical applications, performs task-administration and -processing on two physically separated units, the *Task Administration Unit (TAU)* and the *Task Processing Unit (TPU)*. Time is quantised into discrete *Execution Intervals*, and tasks are *partitioned* into a number of *Execution Blocks* each. The Execution Intervals have a fixed duration, and define the cyclic synchronous operation of the TAU and the TPU. The Task's Execution Blocks are completely executable within one interval and not pre-emptable.

The operating principle can be roughly described as follows: At the beginning of each Execution Interval, the TAU outputs the '*IDs of Task and Execution Block to process*', which identifies the next Execution Block of the task that must be executed according to the scheduling algorithm. The ID corresponds to the task's *Next-Block-Pointer* stored in the task list. After the TPU has read this ID, it processes the associated Execution Block. When the TPU completes the block at the end of the Execution Interval, it outputs the '*ID of Task's next Execution Block*' identifying the task's Execution Block that needs to be executed next. The TAU reads this ID and stores it in the task list as new *Next-Block-Pointer*. Fig. 1 illustrates this mode of operation in more detail.

If the executed block was a task's last one, i. e., if a task has been executed completely, the TPU outputs the block ID 'Nil'. This completion is taken into account when the TAU determines the '*IDs of Task and Execution Block to process*' for the next Execution Interval. That is why the TAU – while the TPU processes an Execution Block – does not only determine the task with the earliest, but also the task with the earliest-but-one deadline. This enables the TAU to immediately output the *NextBlock* identifier of the task with the next-but-one deadline, in case the task with the next deadline corresponds to the task just been processed and just been completed by the TPU.

This concept of *task-oriented real-time execution without asynchronous interrupts* suits safety demands best. The cyclic operating style simplifies the temporal behaviour, complies perfectly well with the safety standard IEC 61508 for applications of the two highest safety classes, and eases formal verification of the application software. Despite operating in discrete cycles, it allows for arbitrarily process-controlled program

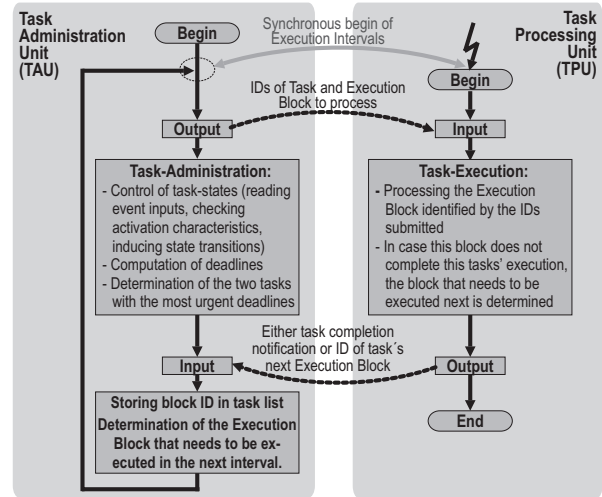


Figure 1. Illustration of the task-oriented operating principle without asynchronous interrupts

flows, just like in conventional task-oriented systems. Of course, special compilation of the application software is required.

The TPU consists of a conventional processor with the Harvard architecture; the TAU is realised as a dedicated logic circuit. For this, the administration algorithms are structured in a way as to allow for parallel processing of the operations related to a single task, whereas all tasks are sequentially subjected to these operations. Fig. 2 illustrates this processing pattern. It shows the main parts of the TAU, the *Task Data Administration (TDA)* and the unit for *Activation Control and Scheduling (ACS)*.

The TDA administrates a *Task List (TL)*, which contains a set of parameters for each task such as its current state and its execution characteristics. It cooperates closely with the ACS while sequentially processing all tasks within each Execution Interval. During this *Sequential Task Administration (STA)*, the TDA initiates for each task a three-phase process:

1. First, the TDA accesses the TL and transfers the task's entire data set to dedicated input registers of the ACS.
2. Then, the ACS processes the task data and outputs an updated data set. This is done by combinational logic within one clock cycle.
3. During the last phase, the TDA transfers the updated task data from the ACS back to the TL.

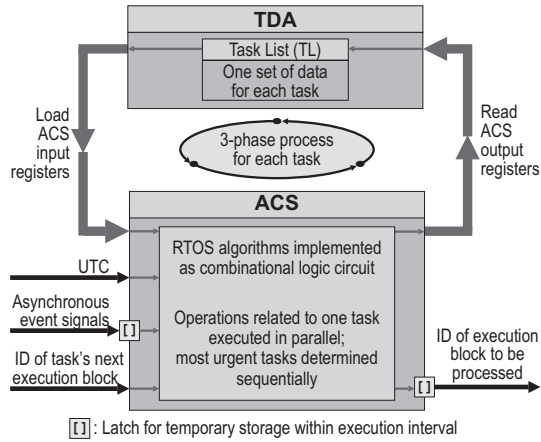


Figure 2. The TAU consists of the TDA and the ACS. While they do the Sequential Task Administration (STA), a 3-phase process is carried out once for each task.

This way, the ACS carries out the following operations in the course of the STA:

1. Checking the activation characteristics (e.g., checking time schedules or asynchronous occurrences)
2. Supervising task state transitions
3. Computing deadlines
4. Generating updated task parameters
5. Identifying the task with the earliest deadline and the one next in line
6. Output of the *ID of Block to Execute*

The first four operations are separately executable for each task. Therefore, they are performed in parallel by a combinational digital circuit. The fifth item requires comparing the deadlines of all activated tasks. This is carried out sequentially, while the IDs of the two most urgent tasks are temporarily stored within the iterations of the 3-phase process. The ID of the Execution Block that needs to be processed in the subsequent interval is output at the end of an Execution Interval, after the TPU submitted an ID to the TAU.

Since the TDA and the ACS are implementable in form of a digital logic circuit, extremely short response times are realisable without the use of a multi-layered operating system structure and without minimising the computational effort by applying a priority-based

scheduling algorithm, as it is usually done in conventional real-time systems. This results in a low complexity of the proposed hardware scheduling concept and makes it particularly suited for highly safety critical applications.

It is an important fact that, at the begin of an Execution Interval, the state of a PES is completely defined by the content of the TPU's RAM and of the TL. Thus, these instants are most appropriate to rejoin the redundant processing after state restoration.

For further details of the PES concept please refer to [9] or [10].

5. Hardware-Assisted State Restoration

In a redundant configuration of multiple PESs, each PES outputs a *Serial Data Stream (SDS)* that provides full information about the internal processing. By monitoring the SDSs of redundant units, a restarted PES can copy the internal state, and – after state equivalence has been reached – rejoin redundant processing. The SDSs are organised in *Transfer Cycles* that match the Execution Intervals. Fig. 3 illustrates this. Within every cycle, an SDS transfers information about a fixed number of data modifications. This number sets the limit of modifications permitted.

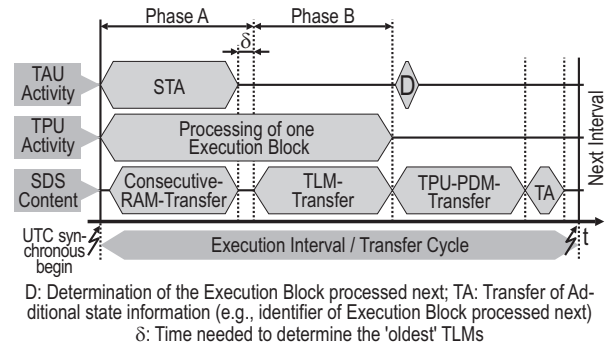


Figure 3. Data transfer of Serial Data Streams (SDS)

Restoring the TPU's RAM Content: Changes of the data memory content of the TPU are always PDMs, e.g., they are always induced by the application software. As already discussed in Section 2, the volume of these PDMs can easily be limited by restricting the number of write accesses permitted within an Execution Interval. This allows complete transfer of the associated information at the end of every cycle (*TPU-PDM Transfer*). The write accesses can be limited by the compiler software while partitioning a program's

code into Execution Blocks. Complete transfer of the RAM content within a predefined time-frame t_1 also requires to transfer the memory content not altered within t_1 . This is carried out by transferring subsets of the RAM content while the TAU sequentially administers the tasks.

The method described so far copies only the RAM content that is modified at least once. In order to guarantee that the complete content is copied within in a predefined time frame, the entire RAM content is transferred within a number of consecutive cycles. This *Consecutive RAM transfer* is performed in synchrony with the international time standard UTC (Universal Time Co-ordinated) to obtain identical SDSs from redundant PESs.

Restoring the TAU’s Task List: The volume of *TL Modifications (TLMs)* depends upon the frequency of task state transitions, which is bounded by the execution characteristics of all tasks. Since TLMs can be either program-controlled (PDM) or event-controlled (EDM), a special technique is applied to minimise the required transfer bandwidth. Each TL data word i is assigned an integer A_i that represents the *Age* of the stored value. By default, each Age parameter is set to its maximum representable value, A_{Max} . Any time a data word is modified, the associated Age integer is set to 0. As long as an Age value is lower than $(A_{Max} - 1)$, it is incremented by one at the begin of every Execution Interval. Thus, the highest integer values (except A_{Max}) identify the *oldest* data values. Within each cycle, only a fixed number n of oldest data words is transferred via SDS, and the associated Age integers are set to A_{Max} . As shown in Fig. 3, this *TLM-Transfer* starts after the TAU completed the STA. The moment all Age integers become A_{Max} , all recent TLMs have been transferred and a notification signal is sent via SDS.

The method described so far copies only the TL data words that are modified at least once. Complete transfer of the TAU data is achieved by changing the *Ages* of all data words at UTC-synchronous instants. This also ensures that the SDSs of redundant PESs are identical.

Implementation as Digital Circuit: The SDS is generated by a digital logic circuit. This enables to improve performance by special custom-built opportunities to access the TL of the TAU, which is also implemented as digital circuit. Realising the functions to restore the TPU RAM content as digital circuit is straightforward and, therefore, not discussed here. For this, it is just necessary to temporarily store the write accesses in a dedicated memory block. This is preferably a Dual-Port-Memory to enable separate data access for the digital logic that generates the SDSs.

Determining the oldest TL data words is more prob-

lematic. In order to avoid detrimental delays which lengthen the cycle duration, the oldest data words should instantaneously be accessible after the TAU completed the STA. This detrimental delay, which is depicted in figure 3 as δ , decreases the computational performance: the shorter the STA and δ , the higher the proportion of cycle time that can be used for TLM- and TPU-PDM-Transfer. The time spent for the consecutive RAM-Transfer can be freely chosen.

Realising the administration of the age variables is easy to implement. During the STA, the TAU processes all TL data words in consecutive order: each word is read, modified if necessary, and stored again (comp. figure 2). The associated age integers can easily be handled in parallel to that: if a TL data word is modified, the associated age integer is set to 0 before it is stored again; if a TL data word is not modified, its age integer is incremented as long as it is lower than $(A_{Max} - 1)$.

The n oldest data words are determined by the digital circuit illustrated in Fig. 4. Its main components are the *Pointer Register Unit (PRU)*, the *Temporary Storage Unit (TSU)*, and the *Read-Out Logic (ROL)*, which co-operate within a two-phase process. Phase 1 commences at interval begin, and phase 2 starts when the STA completes (comp. figure 3).

The PRU contains for each admissible age value (except A_{Max}) a register-set which comprises a *Current Pointer (CP)*, a *Previous Pointer (PP)*, and a *Next Existing Age (NEA)*. Both CP and PP point to an address of the TSU, whereas NEA indicates the lower age value that occurs next in a sequence of age integers. If, for example, the sequence $\{1, 4, 1, 3\}$ is processed, e. g. only the age values 1, 3 and 4 occur, NEA of the register-set associated with age 4 will point to the age value 3, and NEA of the register-set associated with age 3 will point to the age value 1 afterwards. In addition to the register-sets, the PRU contains a register to store the *Maximum Age Value (MAV)* and an *Address Counter (AC)*. At the begin of every processing cycle, the CP of each register-set is set to its associated age value, each register-set’s NEA is set to ‘-1’, and the AC is set to A_{MAX} .

During phase 1, PRU and TSU operate as follows. While the TAU sequentially handles all TL data words within the STA, it always provides the currently processed *TL Data Word*, its *TL Address*, as well as the associated *Age Value*. Every time a new data word, address and age are provided, the PRU accesses the register-set associated with the new age value, and outputs its stored CP as *Storage Address (SA)* at the next clock transition. Simultaneously, induced by the same clock transition, CP is stored as new PP, and the AC

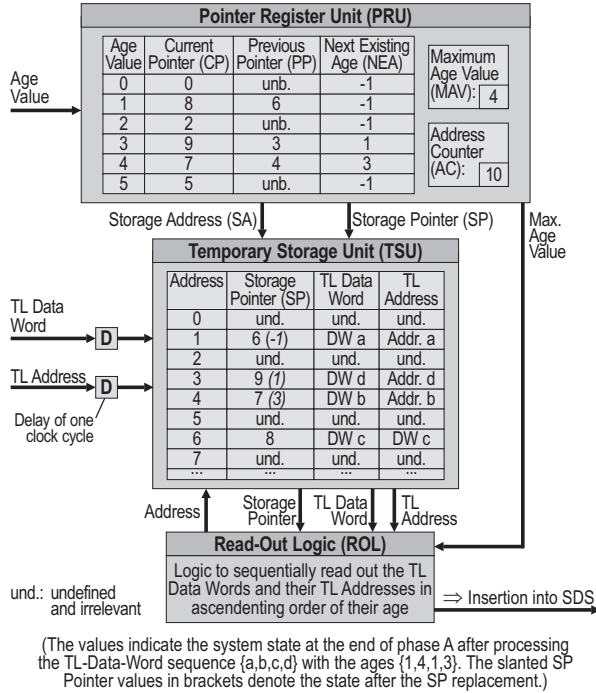


Figure 4. Digital circuit concept to determine the oldest TL data words. Here, A_{Max} is 6.

value is stored as new CP as well as sent out as *Storage Pointer (SP)*. Additionally, all register-sets associated with higher age values check whether they must take over the new age value as new NEA. In case the new age value is higher than the current MAV, it is stored as new MAV. At the next clock transition, the AC is incremented, and the TSU stores the new TL data word together with its TL address at the address to which the output SA of the PRU points. Additionally, the SP provided by the PRU is stored.

The completion of the STA initiates phase 2, during which some of the most recently stored SPs are replaced, and the data stored in the TSU are read out. This *SP replacement* links the data words stored in the TSU to a chain, which allows to sequentially read out the stored TL data words and their addresses in descending order of their age. The replacement is carried out by processing all register-sets of the PRU in descending order of their assigned age values, and takes only one clock cycle for each register-set. For the replacement, the PP of each register-set is used to address the TSU data word whose SP value is replaced by the NEA stored in the register-set.

By accessing the Dual-Port-RAM of the TSU via

its second port, the ROL can read out the temporarily stored data in descending order of their age. The MAV, which is provided by the PRU, points to the address that needs to be accessed first. The TL data words and TL addresses read can directly be added to the SDS. The SP pointer read indicates the TSU address that the ROL must access next. Thus, in the example of Fig. 4, the ROL would read out the TL data words in the order b, d, a, c. Once the SP pointer read is '-1', all TL data words whose age values are not A_{MAX} have been handled and the ROL adds a special notification signal to the SDS.

The ROL does not need to wait for completion of the SP replacement, it can immediately start reading out the TSU at the begin of phase 2. Since the read out starts with the oldest data word, and the SP replacement takes only one clock cycle per pointer, it is guaranteed that a necessary replacement will have been performed before the ROL accesses the associated address.

State Restoration Process: As can be seen in Fig. 5, the restoration process has two phases, *I* and *II*. The former begins together with the first transfer cycle after re-starting a PES, which is automatically initiated after detecting a processing error or replacing a unit. This phase endures as many module cycles as the SDS transfer of the TPU-RAM content in consecutive subsets requires to cover the entire RAM. Thus, after completion of phase *I*, the TPU-PDM transfer enables to reach state equivalence of the TPU-RAM at the end of every transfer cycle. This phase takes at least as much module cycles as the SDS transfer of the TPU-RAM content in consecutive subsets requires, and at least one of the UTC-synchronous points in time, at which all age integers are set to the maximum value, must fall within it.

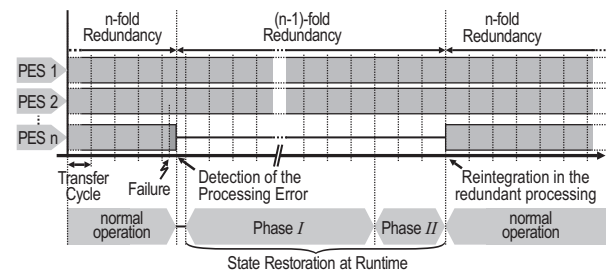


Figure 5. The state restoration process

Phase *II* endures until state equivalence of the TAU's TL of the re-started PES and the redundant counterparts is also reached at the end of a transfer cycle. This is the case when at least one of the UTC-synchronous points in time, at which all age integers are set to the

maximum value, has occurred since the beginning of phase *I*, and the sending PESs have transferred all recent TLMs via SDS, which is signalled via SDS by the previously mentioned notification signal. This signal denotes that all TL data words whose age values do not equal A_{MAX} have been handled. Thus, upon occurrence of this notification signal, state restoration is complete, and the restored PES can re-join the redundant processing at the next module cycle's begin.

6. Conclusion

The problems related to 'State restoration at runtime', which refers to rehabilitating a PES that is in a faulty state by copying the internal state of its redundantly operating counterparts at runtime, were discussed, and a concept that integrates this functionality in a dedicated real-time system was introduced. What distinguishes this concept from others is that different methods are applied to restore the data of the application software and the task-administration data. This allows to take the differences between program-induced and event-induced state changes into account. The concept bases on the PES-architecture presented in [9], which operates in discrete cycles and implements the task administration as digital logic circuit. This enables special custom-built opportunities to access the task-administration data, like, e.g. hard-wired linkage to a digital logic circuit. In contrast to various other methods, the proposed state restoration concept does not impose special constraints on the software development, it only affects computational performance.

The transfer policy 'oldest data first', which is applied for the task-administration data, enables to completely transfer the task-administration data by copying a fixed number of oldest data values each cycle. As major benefit, the amount of event-induced task-administration data modifications within a cycle does not need to be restricted to a number that enables transfer of the associated state changes within the cycle. Since the transfer policy avoids repeated transfer of often modified data to a large extent, it reduces the required transfer bandwidth, or results in a higher computational performance for a fixed bandwidth. The implementation as digital circuit partially enables to transfer information about the internal state while the application software is executed. Moreover, it allows to determine the age of task-administration data values without requiring additional computing time as a software-based implementation would do.

Unfortunately, various aspects could not be discussed here due to lack of space. The interested reader may contact the author for further information.

References

- [1] J. Adams. Hardware-assisted recovery from transient errors in redundant processing systems. In *Proc. IEEE Fault-Tolerant Computing Symp.* 19, pages 517–519, 1989.
- [2] J. Adams and T. Sims. A tagged memory technique for recovery from transient errors in fault-tolerant systems. In *Proc. Teal-Time Systems Symp.*, pages 312–321, 1990.
- [3] D. Basu and R. Paramasivam. An approach to software assisted recovery from hardware transient faults for real time systems. In F. Koornneef and M. van der Meulen, editors, *Safecom 2000*, volume 1943 of *LNCS*, pages 264–274. Springer, 2000.
- [4] A. Bondavalli, F. Giandomenico, F. Grandoni, D. Powell, and C. Rabéjac. State restoration in a COTS-based N-modular architecture. In *1st IEEE Int. Symposium on Object-oriented Real-time distributed Computing (ISORC '98)*, pages 174–183, Kyoto, Japan, Apr. 1998.
- [5] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium*, pages 171–187, London, UK, 1974. Springer-Verlag.
- [6] M. Patino-Martinez, R. Jimenez-Peris, and A. Romanovsky. Bridging the gap between hardware and software fault tolerance. Technical Report 766, University of Newcastle upon Tyne, School of Computing Science, 2002.
- [7] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD Conference*, pages 109–116, 1988.
- [8] T. Sims. Real-time recovery of fault-tolerant processing elements. *IEEE Aerospace and Electronic Systems Magazine*, 12:13–17, 1997.
- [9] M. Skambraks. A safety-related PES for task-oriented real-time execution without asynchronous interrupts. In R. Winther, B. A. Gran, and G. Dahll, editors, *Safecom 2005*, volume 3688 of *LNCS*, pages 261 – 274, Berlin-Heidelberg-New York, 2005. Springer.
- [10] M. Skambraks and W. A. Halang. A PES for use in highly safety-critical control. In K. Man, editor, *IEEE Intl. Conf. on Industrial Technology*, pages 308 – 313, Piscataway, 2005. IEEE.