Proceedings of the International Multiconference on Computer Science and Information Technology pp. 425–431
ISSN 1896-7094

Task-Oriented Real-Time Execution without Asynchronous Interrupts combined with Runtime State Restoration

Martin Skambraks, Wolfgang Halang

Chair of Computer Engineering
Fernuniversität, 58084 Hagen, Germany
{martin.skambraks|wolfgang.halang}@fernuni-hagen.de

Abstract. The architectural concept of a programmable electronic system is presented, which is particularly suited for highly safety-critical applications. Its most essential characteristics are task-oriented real-time execution without the need for asynchronous interrupts and the ability for state restoration at runtime. The concept of task execution without the use of asynchronous interrupts combines the advantages of the classic synchronous and asynchronous approaches to real-time programming. It lowers the complexity of both hardware architecture and conforms particularly well with the safety standard IEC 61508. As a special form of forward recovery prevents redundancy attrition due to transient faults.

1 Introduction

The programmable systems currently employed in safety-critical applications follow either the Cyclic Executive (CE) approach (synchronous programming) or use an Concurrency Control Architecture (CCA) (asynchronous programming) [7]. The CE approach suits the needs for safety-licensing well, but its field of application is limited to simple control tasks. The CCA, on the contrary, has a less restricted field of application and a more problem-oriented programming style, but it requires far more effort for safety-licensing. With the intention to combine the advantages of both, a novel architectural concept for a task-oriented real-time execution has been devised. Time is quantised into discrete Execution Cycles, and tasks are paritioned into Execution Blocks matching these cycles. This concept lowers the complexity of both hardware architecture and temporal behaviour and, thus, conforms particularly well with the safety standard IEC 61508. Additionally, the cyclic processing scheme eases the integration of a special technique for forward recovery. This technique, which is subsequently referred to as *Runtime State Restoration*, allows to configure PESs redundantly with the capability to re-start units that are in faulty states, e. g. due to a transient hardware failure or unit replacement, without the need to interrupt real-time processing.

The main part of this paper is structured as follows. Section 2 categorises conventional PESs into two classes, and discusses their benefits and drawbacks with regard to safety aspects. The task-processing strategy without asynchronous interrupts and its advantages regarding safety-licensing are described in the third section. The fourth section explains the major problem for runtime state restoration and discusses existing techniques. Subject of the fifth section is the concept of Modification Bits, on which the applied state restoration technique is founded. Finally, a short summary recapitulates the most essential aspects and states open issues for further research.

2 Classic Concepts for Real-Time Execution

There are two fundamentally different concepts for real-time execution, called 'Synchronous Programming' and 'Asynchronous Programming'.

Synchronous Programming: This concept is also called 'Cyclic Executive Approach'. It bases on periodic execution of the application software following a predefined time pattern. Since computing processes can only be scheduled periodically, sporadic events need to be handled by cyclic polling. The fundamental characteristic of synchronous programming is that the computing processes are scheduled by the developer during application design.

Architecture and time behaviour of these systems as well as the application-specific software are of remarkably low complexity. This minimises the effort for safety-licensing, and renders this PES category particularly suitable for applications of highest safety-criticality. Unfortunately, the cyclic programming

style is not very problem-oriented. As typical for synchronous programming, the cyclic operating principle strongly restricts the implementation of process-controlled program flows, i. e., condition-controlled branching can only be realised to a certain extent. Additionally, extensive algorithms cause long cycle periods, and processing with varying response times is difficult. To sum up, synchronous programming suits the demand of safety-licensing best, but the field of application is limited to simple control tasks.

Asynchronous Programming: The asynchronous approach, also called 'Concurrency Control Architecture' is used in most contemporary systems. The application software is organised in tasks which are administrated by an operating system. A processor is assigned to one of the activated tasks, and scheduling depends on the process flow. This process-dependent scheduling causes the need for special synchronisation mechanisms. The main difference to the synchronous concept is that computing processes are scheduled at runtime.

In comparison with the synchronous programming style, the asynchronous approach causes less restrictions to process-dependent program flows and enables asynchronous processing of several tasks. Although its task-oriented programming style is more problem-oriented than cyclically processed program code, this PES category causes more effort for safety-licensing. This is due to high complexity of the temporal behaviour and the need for proving the feasibility of application software. In conclusion, the application field of task-oriented PESs is not restricted as that of cyclic PLCs, but safety-licensing takes far more effort.

3 Task-Oriented Real-Time Execution in Discrete Cycles

The concept introduced here combines the advantages of both PES categories by separating task execution and task administration. A Task Processing Unit (TPU) executes application-specific program code; a Task Administration Unit (TAU) is responsible for task state transitions and processor scheduling.

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 are defined for the physically separated TAU and TPU, which operate cyclically and in synchrony. They have the following characteristics.

- Each Execution Block is executable within a single Execution Interval.
- The execution of a block is not pre-emptable.
- The Execution Blocks of a task are indexed for identification.
- The blocks of a task do not need to be executed in consecutive order.

The operating principle can roughly be described as follows: The Execution Cycles consist of two phases, A and B. In phase A, an Execution Block is processed and, after the block has been completed, a pointer to the task's next block is submitted to the task administration functions. The administration functions executed during phase Bcomprise task state control and processor scheduling. At the end of phase B, the identifier of the Execution Block that needs to be processed in the next cycle is output (cf. Fig. 1).

This operating principle renouncing the use of asynchronous interrupts renders synchronisation mechanisms such as semaphores superfluous, since any task has non-interruptable exclusive access to the processor during an Execution Interval. Using variables stored in data memory, mutually exclusive access to peripheral components can be realised by simple means.

The absence of special synchronisation mechanisms results in design simplification. This simplification eases proving correctness with formal means. Moreover, since IEC 61508 restricts the use of pointers, dynamic objects, and interrupts in highly safety-critical applications, the proposed PES concept complies better with the safety standard than conventional task-oriented systems.

Obviously, the cycle length affects the minimum feasible response time. Hence, a short cycle duration is desirable. On the other hand, the shorter the cycle time the lower is the percentage of time actually spent for task execution, since the time needed for task administration is independent of the cycle duration. That is why task administration is implemented in form of a digital logic circuit. This allows to speed up execution without algorithmic techniques that lower computational effort but increase architectural complexity. Additionally, this approach enables the use of custom-built facilities to access the task-administration data. The latter is essential to integrate the concept of state restoration at runtime introduced in the next section.

In summary, this concept of task execution without the necessity of asynchronous interrupts significantly simplifies the temporal behaviour as well as the hardware structure, eases formal verification, and increases conformity with IEC 61508 for systems of highest safety criticality (SIL 3/SIL 4). Of course, the proposed operating principle requires special compilation of the application software.

4 The Problem of State Restoration at Runtime

State restoration at runtime, frequently referred to by the misleading term 'forward-recovery', requires a redundant configuration of uniformly operating PESs. Each PES must be able to

- detect processing errors and leave the redundant operation in case of processing errors,
- copy the internal state of the redundant units at runtime, and
- re-join the redundant operation when state equivalence has been reached.

In order to avoid Common Cause Failures, redundant units must be installed in spatial distribution. This demand for spatial distance between the redundant units implies that the transfer bandwidth between them is limited. Furthermore, while a PES equalises its internal state to that of its running counterparts, the latter continuously change their internal states. This leads to very high communication requirements, since frequently changed data might need to be transferred multiple times. Hence, there is always a trade-off between computing performance realisable and transfer bandwidth necessary.

Analysing software execution, we discover that a huge fraction of the data stored in RAM are intermediate values, which are irrelevant in terms of state restoration as long as the associated final data values are copied within the restoration process. Because of this, the required transfer bandwidth can be reduced through cyclic operation. Intermediate values within a processing cycle are not considered for state restoration, just the final results at a cycle's end are copied. The drawback of this operating principle is that data changes can only be transferred after a processing cycle has been completed (cf. Fig. 2).

Known 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 proposed techniques – especially the hardware-oriented ones – are very application-specific [4].

One of the most well-known hardware-oriented techniques, which has been developed at the *Charles Stark Draper Laboratory* in Cambridge, is presented in [2], [1] 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

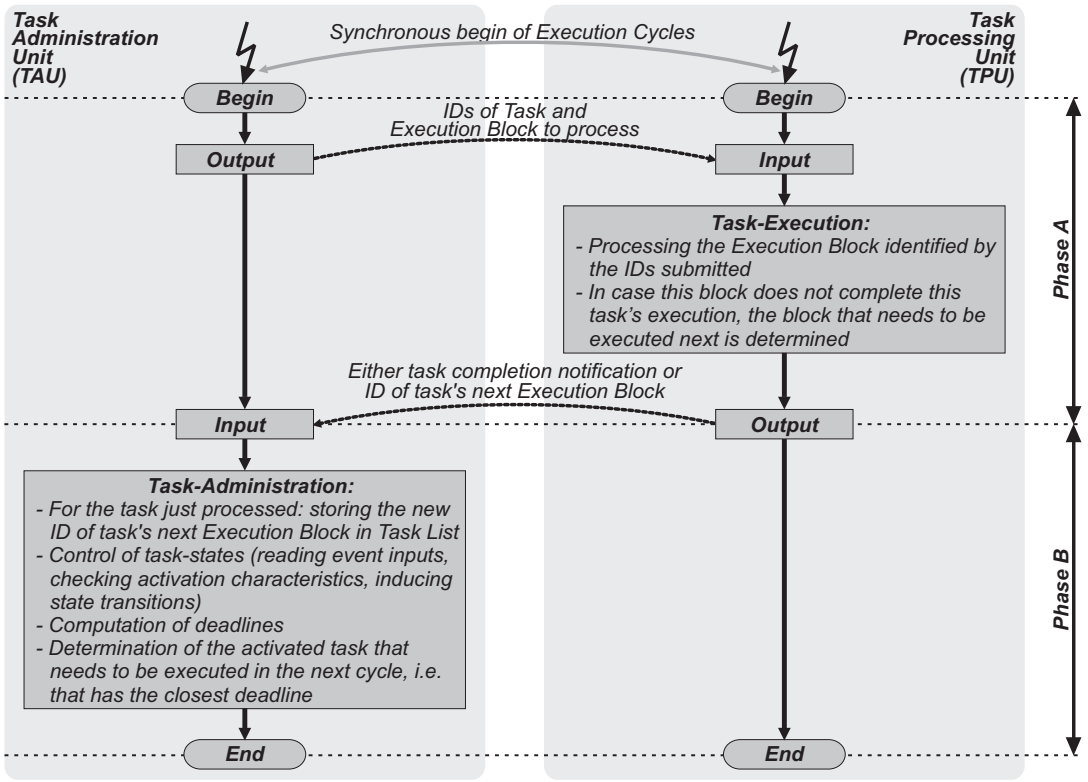

Fig. 1. The operating principle

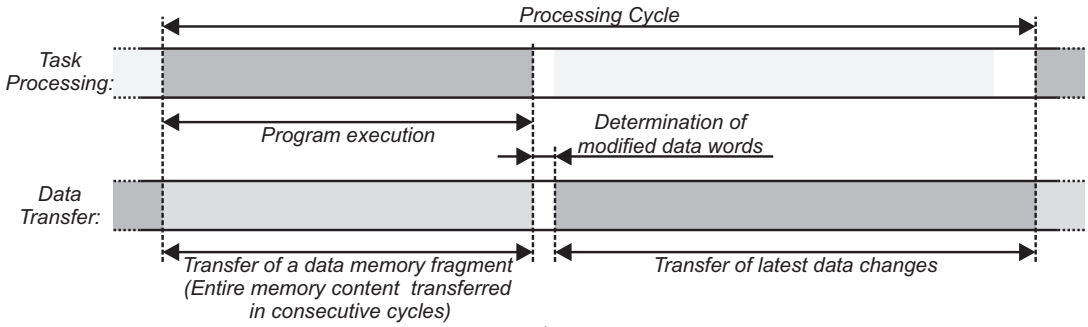

Fig. 2. A cyclic execution pattern can reduce the required transfer bandwidth

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. 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 main-cycle duration 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-cylces 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 (precedense 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 entirely restore the 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

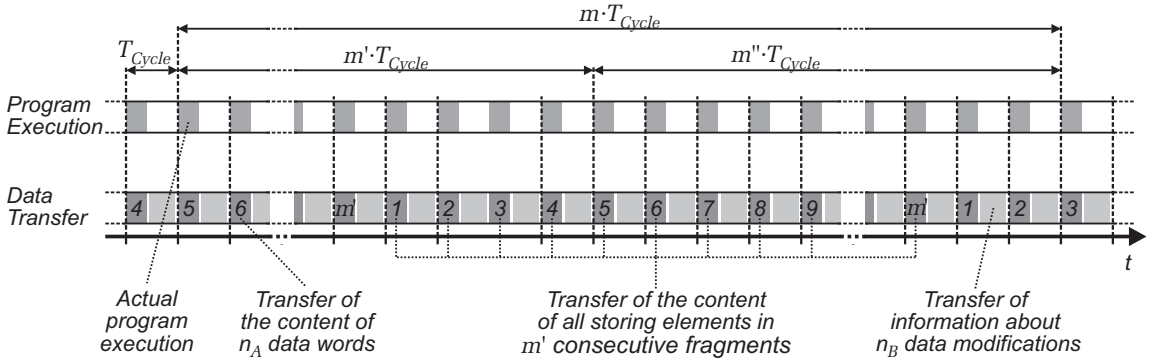

Fig. 3. The two phases – I and II – of the state restoration process.

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. State restoration is complete when all identification bits are in 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.

5 The Applied State Restoration Technique

As discussed in section 4, a cyclic operating principle enables higher computing performance for a given transfer bandwidth. Hence, the concept of task-oriented real-time execution introduced in Section 3 is perfectly suited for state restoration at runtime.

In order to minimise the transfer effort, data modifications are not immediately transferred, i. e., not at the end of the cycle they occurred in. Instead, the concept of *Modification Bits* is applied, which is the underlying principle of the state restoration approach proposed by Bondavalli et al. (cf. Section 4).

Every data word is assigned a Modification Bit, which signals whether the data word has been modified. This bit is set when the data word's value is changed, and reset when the new value has been transferred to the redundant PESs. Instead of transferring information about data modifications immediately, i. e., within the cycle they occurred in, the values and addresses of a fixed amount of data words is transferred at the end of any cycle.

This technique has the advantage that the transfer bandwidth available does not need to suffice a cycle's the maximum number of data modifications. In other words, the approach does not necessitate to exchange at any cycle's end as many data bits as the transfer of the information about a cycle's data modifications would require in the worst case. Instead, the average frequency of data modifications defines the amount of data that must be exchanged every cycle.

As a drawback, the technique increases the effort to prove that the state restoration will complete within a given time-frame. This is because as for every data word it is necessary to know the maximum number of value changes within a given time-frame.

If the maximum frequencies of value changes are known, the amount of data to be transferred any cycle is computable. The calculation depends on how data words are handled whose values are not changed within a specified time-frame. Best performance is provided by the policy depicted in Fig. 2, which uses the transfer medium during actual program execution to transfer the content of all storage elements in consecutive fragments. This approach does not add the handling of non-modified data words to the transfer after actual program execution and, thus, requires less data to be restored by the Modification Bit scheme.

In order to prove that the state restoration completes within m cycles, the state restoration process is divided into two phases (cf. Fig. 3).

Phase I: The first phase ensures that the content of all storage elements is copied at least once. If α is the number of data words in which the storage elements are organised, and n_A is the number of data

words whose content is transferred during actual program execution of a cycle, then, Phase I endures not longer than

$$m' = 1 + \left\lfloor \frac{\alpha}{n_A} \right\rfloor \tag{1}$$

cycles. Thus, after m' consecutive cycles, all data words have been copied at least once, and the Modification Bit scheme only needs to keep track of data modifications.

Phase II: The second phase, which endures $m'' = (m - m')$ cycles, keeps track of data modifications until the state restoration completes. For this, there must be at least one cycle within Phase II, after which all data modifications since the begin of Phase I have been transferred.

This is the case if the number of data modifications whose information is transferred at every cycle's end equals

$$n_B \geq \frac{1}{m''} \cdot \left(\alpha + \sum_{i=1}^{\alpha} \left\lfloor \frac{m'' \cdot t_{Cycle}}{P_i} \right\rfloor \right). \tag{2}$$

In the equation, P_i is the minimum period between two value changes of the i-th data word. The formula results from the fact that every data word i is changed at most one time more than the complete period P_i fits in m'' cycles. n_B does not necessarily need to be an integer number. Since it is possible to transfer the information about a data modification within a number of consecutive cycles, n_B can be any rational number.

The major drawback of this state restoration policy is that the handling of the Modification Bits causes additional computing load. This makes a software-based implementation as presented in [4] inefficient. Particularly the search for the data words with set modification bits strongly limits the performance achievable, since – in the worst case – all modification bits must be evaluated.

Here, dedicated hardware turns out to be valuable. The prototype of the proposed PES showed that it is not only possible to administrate the modification bits in parallel to task execution and task administration, but also to identify the data words with set modification bits. Thus, there is no extra delay for the determination of the data words that need to be transferred.

6 Combining Execution Policy and Restoration Scheme

The state restoration is carried out in synchrony with the global time standard *Universal Time Coordinated (UTC)*: at certain UTC-synchronous instants, which coincide with the beginnings of cycles, all Modification Bits are reset. Thus, assuming that the internal task processing – which is also carried out in reference to UTC – of all redundant PESs is identical, the modification bit handling will also be identical from such a moment on. These UTC-synchronous instants are subsequently called *Restoration Synchronisation Instants (RSIs)*. Obviously, state restoration must be completed between two such RSIs (cf. Fig. 4).

Assume that a PES assesses its state to be faulty, e. g., because its own *Serial Data Stream (SDS)* deviates from the other SDSs' majority. Then, this PES re-starts itself and initiates state restoration. For this, it must wait for the next RSI before the state restoration process can begin.

At the moment of the next RSI, all Modification Bits are reset and the state restoration process begins. During the subsequent m' cycles, the entire content of the Task List and the data memory is

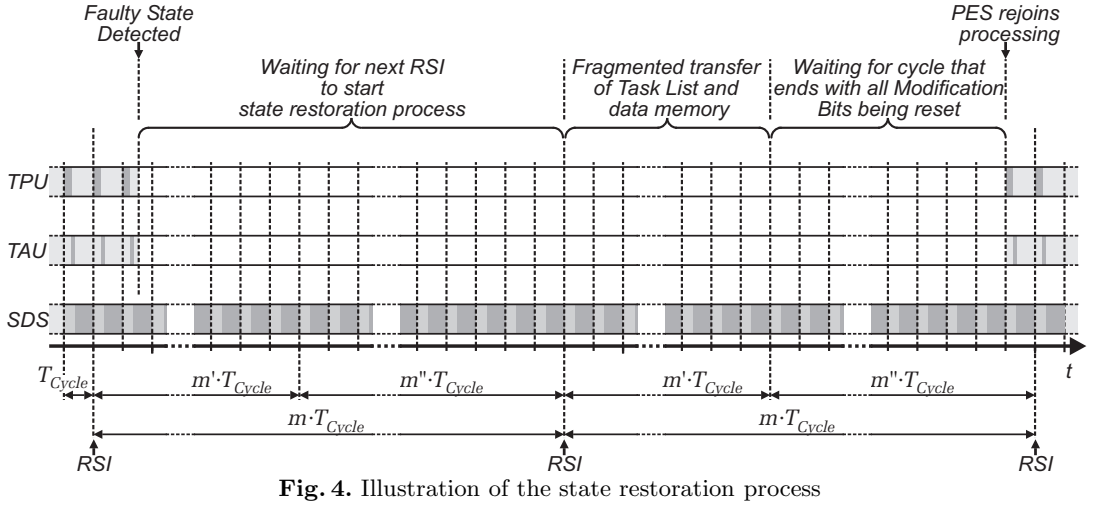

Fig. 4. Illustration of the state restoration process

transferred in consecutive fragments. Even if one of these m' cycles ends with all Modification Bits being in reset state, i. e., all recent data modifications of the running PESs have been copied, the re-started PES cannot rejoin the redundant processing at the beginning of the subsequent cycle. This is because, so far, some data words might have neither been covered by the fragmented transfer in consecutive cycles nor by the Modification Bit scheme.

When the last of the m' cycles has elapsed, all data words of the Task List and the data memory have been copied to the re-started PES at least once. Furthermore, the Modification Bit scheme kept track of all data modifications that happened since the last RSI. Hence, during the remaining m'' cycles before the next RSI, the PES just needs to wait for a cycle that ends with all Modification Bits being in reset state, i. e., with all recent data modifications of the running PESs being copied. As soon as such a cycle occurs, the state restoration is complete and the re-started PES can rejoin the redundant processing in the subsequent cycle. Equation 2 allows to derive conditions that must be fulfilled to ensure that in any case state restoration completes within the m'' cycles.

7 Conclusion

A novel architectural concept for a PES especially suited for safety-critical applications was presented. What distinguishes this concept from other ones is task-oriented real-time execution in discrete intervals and its hardware-supported capability for state restoration at runtime.

The application software is organised in tasks, but – contrary to conventional task-oriented systems – the tasks are subdivided into a number of blocks which are executed in discrete intervals of constant duration. This operating principle renders the use of asynchronous interrupts superfluous. As a result, not only the hardware architecture is simplified, but also the temporal behaviour of the total system. Task administration is carried out by a special-purpose logic circuit. Besides other advantages, it eases the implementation of a hard-oriented approach for the state restoration. This – in turn – provides essential performance benefits against conventional software-oriented techniques.

Unfortunately, many aspects could not be discussed here in detail due to lack of space. Among the aspects that could not be covered are the structure of the hardware architecture, i. e., the dedicated circuitry, and the constraints imposed by the operating principle on the compiler software. The associated problems have been properly solved, but the solutions could not be presented, since this paper is to focus on the state restoration concept. The interested reader is encouraged to ask the author for further information.

References

1. D. 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.
2. S. J. Adams.: *Hardware-Assisted Recovery from Transient Errors in Redundant Processing Systems.* In: Proc. IEEE Fault-Tolerant Computing Symp. **19**, pages 517–519, 1989.
3. D. Basu and R. Paramasivam. *An Approach to Software Assisted Recovery from Hardware Transient Faults for Real Time Systems.* In Computer Safety, Reliability and Security – Safecomp 2000 Conference Proceedings, pages 264 – 274, Berlin Heidelberg, 2000. Springer-Verlag.
4. A. Bondavalli, F. Di 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, April 20–22 1998.
5. James J. Horning, Hugh C. Lauer, P. M. Melliar-Smith, and Brian 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. 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. A. C. Shaw. *Real-Time Systems and Software.* John Wiley, New York, 2001.
8. T. Sims. *Real-time Recovery of Fault-Tolerant Processing Elements.* IEEE Aerospace and Electronic Systems Magazine, 12:13–17, 1997.