View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Research Database In Technology

Technical University of Denmark

Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems with Checkpointing and Replication

Pop, Paul; Izosimov, Viacheslav; Eles, Petru; Peng, Zebo

Published in:
I E E E Transactions on Very Large Scale Integration Systems

Link to article, DOI:
10.1109/TVLSI.2008.2003166

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Pop, P., Izosimov, V., Eles, P., & Peng, Z. (2009). Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems with Checkpointing and Replication. I E E E Transactions on Very Large Scale Integration Systems, 172(3), 389-402. DOI: 10.1109/TVLSI.2008.2003166

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication

Paul Pop, *Member, IEEE*, Viacheslav Izosimov, *Student Member, IEEE*, Petru Eles, *Member, IEEE*, and Zebo Peng, *Senior Member, IEEE*

***Abstract*—We present an approach to the synthesis of fault-tolerant hard real-time systems for safety-critical applications. We use checkpointing with rollback recovery and active replication for tolerating transient faults. Processes and communications are statically scheduled. Our synthesis approach decides the assignment of fault-tolerance policies to processes, the optimal placement of checkpoints and the mapping of processes to processors such that multiple transient faults are tolerated and the timing constraints of the application are satisfied. We present several design optimization approaches which are able to find fault-tolerant implementations given a limited amount of resources. The developed algorithms are evaluated using extensive experiments, including a real-life example.**

***Index Terms*—Fault tolerance, processor scheduling, real time systems, redundancy.**

I. Introduction

SAFETY-CRITICAL applications have to function correctly and meet their timing constraints even in the presence of faults. Such faults can be permanent (i.e., damaged microcontrollers or communication links), *transient* (e.g., caused by electromagnetic interference), or *intermittent* (appear and disappear repeatedly). The transient faults are the most common, and their number is continuously increasing due to the high complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels [8], [15], [28].

The rate of transient faults is often much higher compared to the rate of permanent faults. Transient-to-permanent fault ratios can vary between 2:1 and 50:1 [35], and more recently 100:1 or higher [24]. From the fault tolerance point of view, transient faults and intermittent faults manifest themselves in a similar manner: they happen for a short time and then disappear without causing permanent damage. Hence, fault tolerance techniques against transient faults are also applicable for tolerating intermittent faults and vice versa. Therefore, in this paper, we will refer to both types of faults as *transient faults* and we will talk about *fault tolerance against transient faults*, meaning tolerating both transient and intermittent faults.

Manuscript received June 26, 2007; revised December 04, 2007. First published January 20, 2009; current version published February 19, 2009.

P. Pop is with the Department of Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark (e-mail: paul.pop@imm.dtu.dk).

V. Izosimov, P. Eles, Z. Peng are with Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden (e-mail: viaiz@ida.liu.se; petel@ida.liu.se; zebpe@ida.liu.se).

Digital Object Identifier 10.1109/TVLSI.2008.2003166

Traditionally, hardware replication was used as a fault-tolerance technique against transient faults. For example, in the MARS system [21], [22] each fault-tolerant component is composed of three computation units, two main units and one shadow unit. Once a transient fault is detected, the faulty component must restart while the system is operating with the non-faulty component. This architecture can tolerate one permanent fault and one transient fault at a time, or two transient faults. Another example is the XBW architecture [7], where hardware duplication is combined with double process execution. However, such solutions are very costly and can be used only if the amount of resources is virtually unlimited. In other words, *existing architectures are either too costly or are unable to tolerate multiple transient faults*.

In order to reduce cost, other techniques are required such as software replication [6], [40], recovery with checkpointing [18], [33], [42], and re-execution [19]. However, if applied in a straightforward manner to an existing design, techniques against transient faults *introduce significant time overheads*, which can lead to unschedulable solutions. On the other hand, using faster components or a larger number of resources may not be affordable due to cost constraints. *Therefore, efficient design optimization techniques are required in to meet time and cost constraints in the context of fault tolerant systems*.

Fault-tolerant embedded systems have to be optimized in order to meet time and cost constraints. Researchers have shown that schedulability of an application can be guaranteed for preemptive online scheduling under the presence of a single transient fault [3], [4], [14], [42].

Liberato *et al.* [27] propose an approach for design optimization of monoprocessor systems in the presence of multiple transient faults and in the context of preemptive earliest-deadline-first (EDF) scheduling. For processes scheduled using rate monotonic scheduling, [33] and [26] derive the optimal number of checkpoints for a given task in isolation. [25] discusses the reliability of checkpointed systems as a function of the number of checkpoints and checkpoint overhead.

Hardware/software co-synthesis with fault tolerance is addressed in [36] in the context of event-driven fixed priority scheduling. Hardware and software architectures are synthesized simultaneously, providing a specified level of fault tolerance and meeting the performance constraints. Safety-critical processes are re-executed in order to tolerate transient fault occurrences. This approach, in principle, also addresses the problem of tolerating multiple transient faults, but does not consider static cyclic scheduling.

1063-8210/$25.00 © 2009 IEEE

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

Xie *et al.* [40] propose a technique to decide how replicas can be selectively inserted into the application, based on process criticality. Introducing redundant processes into a pre-designed schedule is used in [9] in order to improve error detection. Both approaches only consider one single fault.

Power-related optimization issues in fault-tolerant applications are tackled in [41] and [17]. Zhang *et al.* [41] study fault tolerance and dynamic power management. Rollback recovery with checkpointing is used to tolerate multiple transient faults in the context of distributed systems. Fault tolerance is applied on top of a pre-designed system, whose process mapping ignores the fault tolerance issue.

Kandasamy *et al.* [19] propose constructive mapping and scheduling algorithms for transparent re-execution on multiprocessor systems. The work was later extended with fault-tolerant transmission of messages on a time-division multiple access bus [20]. Both papers consider only one fault per computation node. Only process re-execution is used as a fault-tolerance policy.

Very little research work is devoted to general design optimization in the context of fault tolerance. For example, Pinello *et al.* [30] propose a simple heuristic for combining several static schedules in order to mask fault patterns. Passive replication is used in [2] to handle a single failure in multiprocessor systems so that timing constraints are satisfied. Multiple failures are addressed with active replication in [13] in order to guarantee a required level of fault tolerance and satisfy time constraints. [43] use software replication and temporal redundancy to provide a tradeoff between energy savings and the number of faults being tolerated in the context of reliable parallel servers.

None of the previous work has considered fault-tolerance policies in the global context of system-level design for distributed embedded systems. Thus, in this paper, we consider hard real-time safety-critical applications mapped on distributed embedded systems. Both the processes and the messages are scheduled using non-preemptive *static cyclic scheduling*. We consider two distinct fault-tolerance techniques: process-level *checkpointing* with rollback recovery [10], which provides time-redundancy, and active *replication* [31], which provides space-redundancy. We show how checkpointing and replication can be combined in an optimized implementation that leads to schedulable applications which are fault-tolerant in the presence of multiple transient faults, without increasing the amount of employed resources.

II. System Architecture

We consider architectures composed of a set N of nodes which share a broadcast communication channel. Every *node* $N_i \in \mathcal{N}$ consists, among others, of a communication controller and a CPU. The communication controllers implement the protocol services and run independently of the node's CPU. We consider that the communications are performed statically based on schedule tables, and are fault-tolerant, using a protocol such as the Time Triggered Protocol (TTP). TTP was designed for distributed real-time applications that require predictability and reliability, and provides services such as message transport with acknowledgment and predictable low latency and clock synchronization within the microsecond range [23].

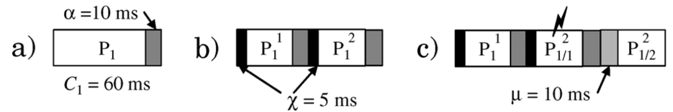


Fig. 1. Rollback recovery with checkpointing. We take into account the error detection overhead (α), the checkpointing (χ) and the recovery overhead (μ).

In this paper we are interested in fault-tolerance techniques for tolerating transient faults, which are the most common faults in today's embedded systems. If permanent faults occur, we consider that they are handled using a technique such as hardware replication. Note that an architecture that tolerates n permanent faults, will also tolerate n transient faults. However, we are interested in tolerating a much larger number of transient faults than permanent ones, for which using hardware replication alone is too costly.

We have generalized the fault-model from [19] that assumes that only one single transient fault may occur on any of the nodes in the system during an application execution. In our model, we consider that at most a given number k of transient faults[1] may occur anywhere in the system during one operation cycle of the application. Thus, not only several transient faults may occur simultaneously on several processors, but also several faults may occur on the same processor.

We have designed a software architecture which runs on the CPU in each node, and which has a real-time kernel as its main component. The kernel running as part of the software architecture on each node has a schedule table. This schedule table contains all the information needed to take decisions on activation of processes and transmission of messages, on that particular node [32].

III. Fault-Tolerance Techniques

The error detection and fault-tolerance mechanisms are part of the software architecture. The software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are themselves fault-tolerant.

We use two mechanisms for tolerating faults: equidistant checkpointing with rollback recovery and active replication. Rollback recovery uses time redundancy to tolerate fault occurrences. Replication provides space redundancy that allows to distribute the timing overhead among several processors. We assume that all the faults we are interested to tolerate are detected, i.e., perfect error detection. However, error detection is not always perfect, and the consequences for the two fault-tolerance mechanisms we use are different. On one hand, replication is susceptible to correlated faults, whereas checkpointing can detect them. On the other hand, an error might be present undetected in a checkpoint, which might necessitate rollback [5].

Once a fault is detected, a fault tolerance mechanism has to be invoked to handle this fault. The simplest fault tolerance technique to recover from fault occurrences is re-execution [19]. In re-execution, a process is executed again if affected by faults.

The time needed for the detection of faults is accounted for by the *error-detection overhead* α. When a process is re-executed after a fault was detected, the system restores all initial inputs of

[1]The number of faults k can be larger than the number of processors.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

that process. The process re-execution operation requires some time for this that is captured by the *recovery overhead* μ. In order to be restored, the initial inputs to a process have to be stored before the process is executed first time.

A. Rollback Recovery With Checkpointing

The time overhead for re-execution can be reduced with more complex fault tolerance techniques such as *rollback recovery with checkpointing* [29], [33], [42]. The main principle of this technique is to restore the last non-faulty state of the failing process, i.e., to *recover* from faults. The last non-faulty state, or *checkpoint*, has to be saved in advance in the static memory and will be restored if the process fails. The part of the process between two checkpoints or between a checkpoint and the end of the process is called *execution segment*.

An example of rollback recovery with checkpointing is presented in Fig. 1. We consider process P_1 with the worst-case execution time of 60 ms and error-detection overhead α of 10 ms, as depicted in Fig. 1(a). Fig. 1(b) presents the execution of P_1 in case no fault occurs, while Fig. 1(c) shows a scenario where a fault (depicted with a lightning bolt) affects P_1. In Fig. 1(b), two checkpoints are inserted at equal intervals. The first checkpoint is the initial state of process P_1. The second checkpoint, placed in the middle of process execution, is for storing an intermediate process state. Thus, process P_1 is composed of two execution segments. We will name the kth execution segment of process P_i as P_i^k. Accordingly, the first execution segment of process P_1 is P_1^1 and its second segment is P_1^2. Saving process states, including saving initial inputs, at checkpoints, takes a certain amount of time that is considered in the *checkpointing overhead* χ, depicted as a black rectangle. In Fig. 1(c), a fault affects the second execution segment P_1^2 of process P_1. This faulty segment is executed again starting from the second checkpoint. Note that the error-detection overhead α is not considered in the last recovery in the context of rollback recovery with checkpointing because, in this example, we assume that a maximum of one fault can happen.

We will denote the jth execution of the kth execution segment of process P_i as $P_{i/j}^k$. Accordingly, the first execution of execution segment P_1^2 has the name $P_{1/1}^2$ and its second execution is named $P_{1/2}^2$. Note that we will not use the index j if we only have one execution of a segment or a process, as, for example, P_1's first execution segment P_1^1 in Fig. 1(c).

When recovering, similar to re-execution, we consider a recovery overhead μ, which includes the time needed to restore checkpoints. In Fig. 1(c), the recovery overhead μ, depicted with a light gray rectangle, is 10 ms for process P_1.

The fact that only a part of a process has to be restarted for tolerating faults, not the whole process, can considerably reduce the time overhead of rollback recovery with checkpointing compared to simple re-execution. Simple re-execution is a particular case of rollback recovery with checkpointing, in which a single checkpoint is applied, at process activation.

B. Active and Passive Replication

The disadvantage of recovery techniques is that they are unable to explore spare capacity of available computation nodes and, by this, to possibly reduce the schedule length. If the process experiences a fault, then it has to recover on the same computation node. In contrast to rollback recovery and re-execution, *active and passive replication* techniques can utilize spare capacity of other computation nodes. Moreover, active replication provides the possibility of *spatial redundancy*, e.g., the ability to execute process replicas in parallel on different computation nodes.

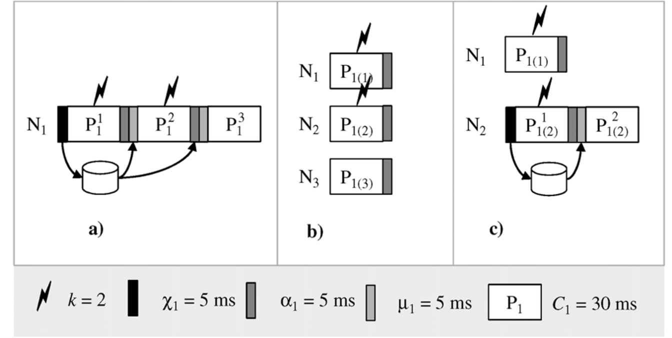


Fig. 2. Policy assignment. To tolerate $k = 2$ transient faults we use: (a) only checkpointing; (b) only replication; or (c) a combination of the two.

In the case of active replication [40], all replicas of processes are executed independently of fault occurrences. In the case of passive replication, also known as *primary-backup* [1], on the other hand, replicas are executed only if faults occur. We will name the jth replica of process P_i as $P_{i(j)}$. Note that, for the sake of uniformity, we will consider the original process as *the first replica*. Hence, the replica of process P_1 is named $P_{1(2)}$ and process P_1 itself is named as $P_{1(1)}$.

In our work, we are interested in active replication. This type of replication provides the possibility of spatial redundancy, which is lacking in rollback recovery. Moreover, rollback recovery with a single checkpoint, in fact, is a restricted case of primary-backup where replicas are only allowed to execute on the same computation node with the original process.

IV. Application Model

We consider a set of real-time periodic applications $\mathcal{A}_k$. Each application $\mathcal{A}_k$ is represented as an acyclic directed graph $\mathcal{G}_k(\mathcal{V}_k, \mathcal{E}_k)$. Each process graph is executed with the period T_k. The graphs are merged into a single graph with a period T obtained as the least common multiple (LCM) of all application periods T_k. This graph corresponds to a virtual application $\mathcal{A}$, captured as a directed, acyclic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each node $P_i \in \mathcal{V}$ represents a process and each edge $e_{ij} \in \mathcal{E}$ from P_i to P_j indicates that the output of P_i is the input of P_j.

Processes are non-preemptable and cannot be interrupted by other processes. Processes send their output values encapsulated in messages, when completed. All required inputs have to arrive before activation of the process.

In this paper, we will consider *hard real-time* applications. Time constraints are imposed with a global hard deadline D, at which the application $\mathcal{A}$ *has to* complete.

The application processes have to be *mapped* (allocated) on the computation nodes. The mapping of an application process is determined by a function $\mathcal{M}: \mathcal{V} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of nodes in the architecture. We consider that the mapping of the application is not fixed and has to be determined as part of the design optimization.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

For a process $P_i, \mathcal{M}(P_i)$ is the node to which P_i is assigned for execution. Each process can potentially be mapped on several nodes. Let $\mathcal{N}_{P_i} \subseteq \mathcal{N}$ be the set of nodes to which P_i can potentially be mapped. We consider that for each $N_k \in \mathcal{N}_{P_i}$, we know the worst-case execution time [34] (WCET) $C_{P_i}^{N_k}$ of process P_i, when executed on N_k.

In the case of processes mapped on the same computation node, message transmission time between them is accounted for in the worst-case execution time of the sending process. If processes are mapped on different computation nodes, then messages between them are sent through the communication network. We consider that the worst-case size of messages is given, which, implicitly, can be translated into a worst-case transmission time on the bus.

The combination of fault-tolerance policies to be applied to each process (see Fig. 2) is given by the following four functions.

- $\mathcal{P}$: $\mathcal{V} \rightarrow$ { *Replication*, *Checkpointing*, *Replication & Checkpointing* } determines whether a process is replicated, checkpointed, or replicated and checkpointed. When replication is used for P_i, and considering k the maximum number of faults, we introduce several replicas into the application $\mathcal{A}$, and connect them to the predecessors and successors of P_i.
  The function $\mathcal{Q} : \mathcal{V} \rightarrow \mathrm{N}$ indicates the number of replicas for each process. For a certain process P_i, and considering k the maximum number of faults, if $\mathcal{P}(P_i) =$ *Replication*, then $\mathcal{Q}(P_i) = k$; if $\mathcal{P}(P_i) =$ *Checkpointing*, then $\mathcal{Q}(P_i) = 0$; if $\mathcal{P}(P_i) =$ *Replication & Checkpointing*, then $0 < Q(P_i) < k$.
  Let $\mathcal{V}_R$ be the set of replica processes introduced into the application. Replicas can be checkpointed as well, if necessary. The function $\mathcal{R} : \mathcal{V} \cup \mathcal{V}_R \rightarrow \mathrm{N}$ determines the number of recoveries for each process or replica. In Fig. 2(a), $\mathcal{P}(P_1) =$ *Checkpointing*, $\mathcal{R}(P_1) = 2$. In Fig. 2(b), $\mathcal{P}(P_1) =$ *Replication*, $\mathcal{R}(P_{1(1)}) = \mathcal{R}(P_{1(2)}) = \mathcal{R}(P_{1(3)}) = 0$. In Fig. 2(c), $\mathcal{P}(P_1) =$ *Replication & Checkpointing*, $\mathcal{R}(P_{1(1)}) = 0$ and $\mathcal{R}(P_{1(2)}) = 1$.
- $\mathcal{X}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow \mathrm{N}$ indicates the number of checkpoints to be applied to processes in the application and the replicas in $\mathcal{V}_R$. We consider equidistant checkpointing, thus the checkpoints are equally distributed throughout the execution time of the process. If process $P_i \in \mathcal{V}$ or replica $P_{i(j)} \in \mathcal{V}_R$ is not checkpointed, then we have $\mathcal{X}(P_i) = 0$ or $\mathcal{X}(P_{i(j)}) = 0$, respectively.

Each process $P_i \in \mathcal{V}$, besides its worst-case execution time C_i, is characterized by an error detection overhead α_i, a recovery overhead μ_i, and checkpointing overhead χ_i.

V. Scheduling Policy and Recovery

In this paper, we consider a *static non-preemptive* scheduling approach, where both communications and processes are statically scheduled. The start times of processes and sending times of messages are determined offline using scheduling heuristics. These start and sending times are stored in form of schedule tables on each computation node. Then, the real-time scheduler of a computation node will use the schedule table of that node in order to invoke processes and send messages on the bus. In general, however, an application can have different execution scenarios, depending on fault occurrences. At execution time, the real-time scheduler will choose the appropriate schedule that corresponds to the current fault scenario. The corresponding schedules for each fault occurrence scenario are called *contingency schedules*.

Let us consider the example[2] in Fig. 3, where we have five processes, P_1–P_5 mapped on three nodes, N_1–N_3. P_2, P_3, and P_4 are mapped on N_1, P_1 on N_2, and P_5 on N_3. The worst-case execution times for each process are given in the table below the application graph. By "X" we show mapping restrictions. We consider that at most two faults can occur, and the overheads due to the fault-tolerance mechanisms (χ, α and μ) are given in the figure. Although the assumption is that at most two faults can happen, the third execution of a process will still be followed by error detection. Even if redundancy is not provided at that point, the system might take some emergency action in case of the very unlikely event of a third fault. However, for simplicity, we will not depict this last error detection overhead in the figures. All processes in the example use checkpointing: P_1–P_4 have one checkpoint, while P_5 has two checkpoints (see Fig. 3).

When checkpointing is used for tolerating faults, we have to introduce in the schedule table *recovery slack*, which is idle time on the processor needed to recover the failed process segment. For example, for P_1 on node N_2, we introduce a recovery slack of $2 \times (C_1 + \mu) + \alpha = 75$ ms to make sure we can recover P_1 even in the case it experiences the maximum number of two faults [see Fig. 3(a)]. In the figure, the re-executions are depicted with a hashed rectangle, while the detection and recovery overheads are depicted with grey rectangles[3]. The recovery slack can be shared by several processes, like is the case of process P_2, P_3, and P_4 on node N_1, Fig. 3(a). This shared slack has to be large enough to accommodate the recovery of the largest process (in our case P_4) in the case of two faults. This slack can then handle any combination with maximum two faults: two faults in P_2 or in P_3, which take less to execute than P_4, or a fault in P_2 and one in P_4, etc. Note that the recovery slack for P_5, which has two checkpoints, is only half the size of the recovery slack that would be needed if P_5 had a single checkpoint, since only one segment of P_5 (either P_5^1 or P_5^2) has to be recovered from its corresponding checkpoint, and not the whole process.

In this paper, we use for checkpointing a particular type of recovery, called *transparent recovery* [19], that hides fault occurrences on a processor from other processors. On a processor N_i where a fault occurs, the scheduler has to switch to a *contingency schedule* that delays descendants of the faulty process running on the same processor N_i. However, a fault happening on another processor is not visible on N_i, even if the descendants of the faulty process are mapped on N_i. For example, in Fig. 3(a), where we assume that no faults occur, in order to isolate node N_1 from the occurrence of a fault in P_1 on node N_2, message m_1 from P_1 to P_2 cannot be transmitted at the end of P_1's execution. Message m_1 has to arrive at the destination at a fixed time, regardless of what happens on node N_1, i.e., *transparently*. Consequently, m_1 can only be transmitted after a time

[2]For clarity, bus communication is ignored in this particular example.

[3]The depiction of these grey rectangles will be omitted from the figures (but counted in the total of the recovery slack) in the remaining of the paper, unless they are needed to support the discussion.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

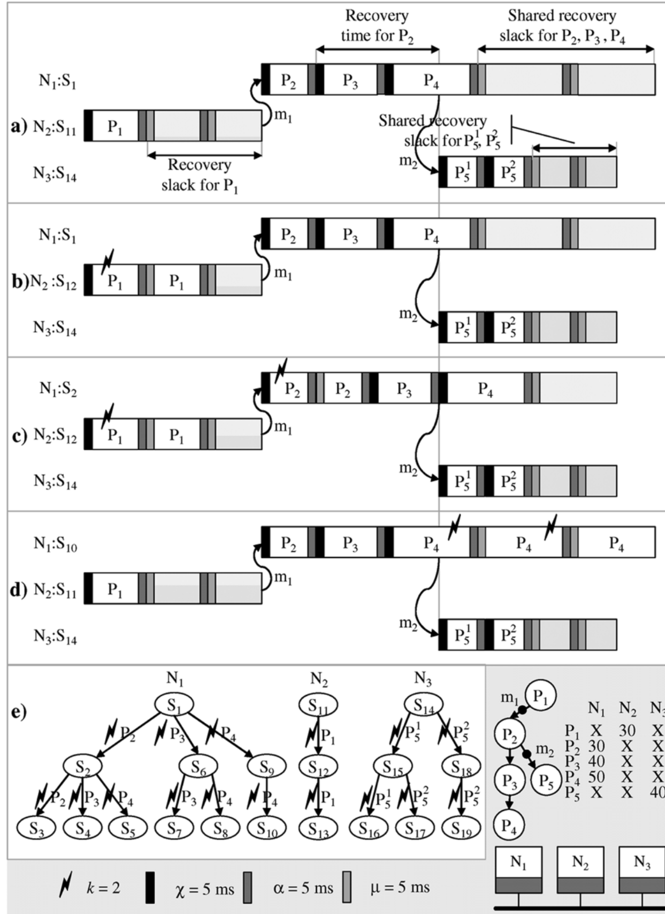


Fig. 3. Fault tolerant scheduling example. The initial schedules (corresponding the no-faults scenario) S_1, S_{11}, and S_{14}, for each node, are the root schedules. If a fault occurs in P_1, we switch from S_{11} to S_{12} on N_2, but S_1 and S_{14} on N_1 and N_3, respectively, are undisturbed because the fault occuring on N_2 is transparent to N_1 and N_3.

$2 \times (C_1 + \alpha) + \mu$, at the end of the recovery slack for P_1. Similarly, the transmission of m_2 also has to be delayed, to mask in the worst-case two consecutive faults in P_2. However, a fault in P_2 will delay processes P_3 and P_4 which are on the same processor (see in Fig. 3(c) the time-line for node N_1). Note that this mechanism is part of the *transparent recovery* scheme proposed in [19]. The disadvantage in delaying the transmission of a message to provide fault masking is that a later fault can prevent its transmission. In order to increase the reliability, we can schedule the message for immediate transmission, tagged with its delivery time. This might, however, necessitate allocating several slots for a message in the bus schedule table. Since in this paper we do not address the faults at the level of the communication infrastructure, assuming that they are handled with a protocol such as TTP, we will not investigate this issue further.

Once a fault happens, the scheduler in a node has to switch to a *contingency schedule*. For example, once a fault occurs in process P_1 in the schedule depicted in Fig. 3(a), the scheduler on node N_2 will have to switch to the contingency schedule in Fig. 3(b), where P_1 is delayed with $C_1 + \alpha + \mu$ to account for the fault. A fault in P_2 will result in activating a contingency schedule on N_1 which contains a different start time not only for P_2, but also for P_3 and P_4 [see Fig. 3(c)]. There are several contingency schedules, depending on the combination of processes and faults. For maximum two faults, and the processes in Fig. 3, there are 19 contingency schedules, depicted in Fig. 3(e) as a set of trees. There is one tree for each processor. Each node S_i in the tree represents a contingency schedule, and the path from the root node of a tree to a node S_i, represents the sequence of faults (labelled with the process name in which the fault occurs) that lead to contingency schedule S_i. For example, in Fig. 3(a) we assume that no faults occur, and thus we have the schedules S_1 on node N_1, S_{11} on N_2 and S_{14} on N_3. We denote such initial schedules with the term "*root schedule*," since they are the root of the contingency schedules tree. An error in P_1 [see Fig. 3(b)] will be observed by the scheduler on node N_2 which will switch to the contingency schedule S_{12}.

The end-to-end worst-case delay of the application is given by the maximum finishing time of any contingency schedule, since this is a situation that can happen in the worst-case scenario. For the application in Fig. 3, the largest delay is produced by schedule S_{10}, which has to be activated when two consecutive faults happen in P_4 [see Fig. 3(d)]. The end-to-end worst-case delay is equal to the maximum delay among the root schedules, including the recovery slack [depicted in Fig. 3(a)]. This is due to the fact that the root schedules have to contain enough recovery slack to accommodate even the worst-case scenario.

VI. Fault-Tolerant System Design

In this paper, by policy assignment we denote the decision whether a certain process should be checkpointed or replicated, or a combination of the two should be used. Mapping a process means placing it on a particular node in the architecture.

There could be cases where the policy assignment decision is taken based on the experience and preferences of the designer, considering aspects like the functionality implemented by the process, the required level of reliability, hardness of the constraints, legacy constraints, etc. Most processes, however, do not exhibit certain particular features or requirements which obviously lead to checkpointing or replication. Decisions concerning the policy assignment to this set of processes can lead to various trade-offs concerning, for example, the schedulability properties of the system, the amount of communication exchanged, the size of the schedule tables, etc.

For part of the processes in the application, the designer might have already decided their mapping. For example, certain processes, due to constraints like having to be close to sensors/actuators, have to be physically located in a particular hardware unit. For the rest of the processes (including the replicas) their mapping is decided during design optimization.

Thus, our problem formulation for mapping and policy assignment with checkpointing is as follows.

- As an input we have an application $\mathcal{A}$ given as presented in Section IV and a system consisting of a set of nodes $\mathcal{N}$ connected to a bus B (see Section II).
- The parameter k denotes the maximal number of transient faults that can appear in the system during one cycle of execution.

We are interested to find a system configuration ψ such that the k transient faults are tolerated and the imposed deadlines

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

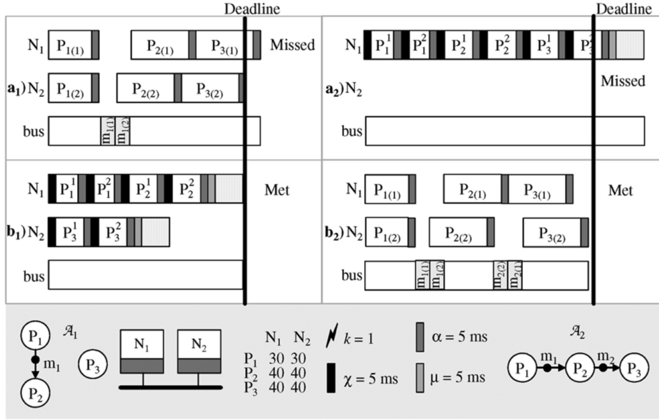


Fig. 4. Comparison of checkpointing and replication. Checkpointing is better (in terms of meeting the deadline) for application $\mathcal{A}_1$. However, for $\mathcal{A}_2$ replication is better.

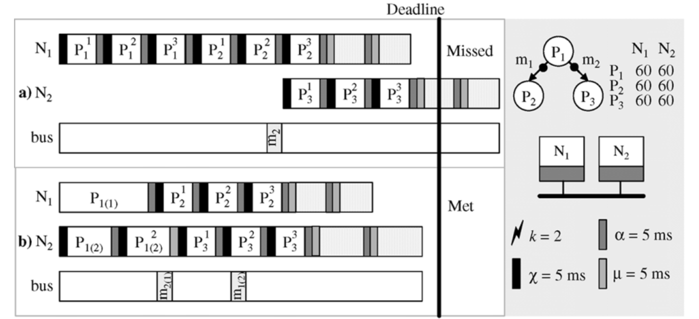


Fig. 5. Combining checkpointing and replication. The deadline cannot be met by replication or (a) checkpointing alone. (b) They have to be combined, as is the case with P_1.

are guaranteed to be satisfied, within the constraints of the given architecture $\mathcal{N}$.

Determining a system configuration $\psi = \langle \mathcal{F}, \mathcal{M}, \mathcal{S} \rangle$ means as follows:

1) finding a fault tolerance policy assignment, given by $\mathcal{F} = \langle \mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{X} \rangle$, for each process P_i (see Section IV) in the application $\mathcal{A}$, for which the fault-tolerance policy has not been a priory set by the designer; this also includes the decision on the number of checkpoints $\mathcal{X}$ for each process P_i in the application $\mathcal{A}$ and each replica in $\mathcal{V}_R$;
2) deciding on a mapping $\mathcal{M}$ for each unmapped process P_i in the application $\mathcal{A}$;
3) deciding on a mapping $\mathcal{M}$ for each unmapped replica in $\mathcal{V}_R$;
4) deriving the set $\mathcal{S}$ of schedule tables on each node.

A. Motivational Examples

Let us illustrate some of the issues related to policy assignment with checkpointing. In the example presented in Fig. 4, we have the application $\mathcal{A}_1$ with three processes, P_1–P_3, and an architecture with two nodes, N_1 and N_2. The worst-case execution times on each node are given in a table to the right of the architecture, and processes can be mapped to any node. The fault model assumes a single fault, thus $k = 1$, and the fault-tolerance overheads are presented in the figure. The application $\mathcal{A}_1$ has a deadline of 140 ms depicted with a thick vertical line. We have to decide which fault-tolerance technique to use.

In Fig. 4 $(\mathrm{a}_1), (\mathrm{a}_2), (\mathrm{b}_1)$, and (b_2), we depict the root schedules[4] for each node and the bus. Comparing the schedules in Fig. 4 (a_1) and Fig. 4(b_1), we can observe that using active replication (a_1) the deadline is missed. However, using checkpointing (b_1) we are able to meet the deadline. Each process has an optimal number of two checkpoints in Fig. 4(b_1). If we consider application $\mathcal{A}_2$, similar to $\mathcal{A}_1$ but with process P_3 data dependent on P_2, as illustrated in the right lower corner of Fig. 4, the deadline of 180 ms is missed in Fig. 4(a_2) if checkpointing is used, and it is met when replication is used as in Fig. 4(b_2). Although the replication in Fig. 4(b_2) introduces more delays on the bus than a_1, the schedule length compared to checkpointing (a_2) is actually smaller.

[4]The schedules depicted are optimal.

This example shows that the particular fault-tolerance policy to use has to be carefully adapted to the characteristics of the application and the amount of available resources. Moreover, the best result is often to be obtained when both techniques are used together, some processes being checkpointed, while others replicated, as in the following example.

Let us now consider the example in Fig. 5, where we have an application with three processes, P_1–P_3, mapped on an architecture of two nodes, N_1 and N_2. A maximum of two transient faults have to be tolerated. The processes can be mapped to any node, and the worst-case execution times on each node are given in a table. In Fig. 5(a) all processes are using checkpointing, and the depicted root schedule is optimal for this case. Note that m_2 has to be delayed to mask two potential faults of P_1 to node N_2. With this setting, using checkpointing will miss the deadline. However, combining checkpointing with replication, as in Fig. 5(b) where process P_1 is replicated, will meet the deadline. $P_{1(1)}$ is a simple replica without checkpointing and message $m_{2(1)}$ from this replica is sent directly after completion of $P_{1(1)}$. In the second replica $P_{1(2)}$ of process P_1, one fault has to be masked, which delays the message $m_{1(2)}$. However, the delay of message $m_{1(2)}$ is less then the delay of message m_2 in Fig. 5(a).

B. Checkpointing

Regarding checkpoints, we will first illustrate issues of checkpoint optimization when processes are considered in isolation. In Fig. 6, we have process P_1 with a worst-case execution time of $C_1 = 50$ ms. We consider a fault scenario with $k = 2$, the recovery overhead μ_1 equal to 15 ms, and checkpointing overhead χ_1 equal to 5 ms. The error-detection overhead α_1 is considered equal to 10 ms. Recovery, checkpointing and error-detection overheads are shown with light grey, black, and dark grey rectangles, respectively.

In Fig. 6, we depict the execution time needed for P_1 to tolerate two faults, considering from one to five checkpoints. Since P_1 has to tolerate two faults, the recovery slack S_1 has to be double the size of P_1 including the recovery overhead, as well as the error-detection overhead α_1 that has to be considered for the first re-execution of the process. Thus, for one checkpoint, the recovery slack S_1 of process P_1 is $(50 + 15) \times 2 + 10 = 140$ ms.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

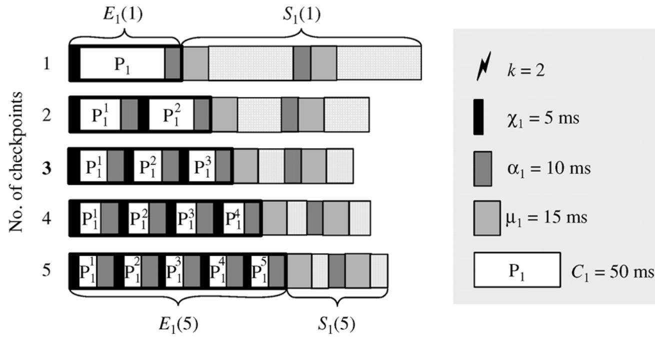


Fig. 6. Locally optimal number of checkpoints. Beyond a certain (locally optimal) number of checkpoints, the reduction in recovery slack is offset by the increase in overheads.

If two checkpoints are introduced, process P_1 will be split into two execution segments P_1^1 and P_1^2. In general, the execution segment is a part of the process execution between two checkpoints or a checkpoint and the end of the process. In the case of an error in process P_1, only the segments P_1^1 or P_1^2 have to be recovered, not the whole process, thus the recovery slack S_1 is reduced to $(50/2+15) \times 2+10 = 90$ ms.

By introducing more checkpoints, the recovery slack S_1 can be thus reduced. However, there is a point over which the reduction in the recovery slack S_1 is offset by the increase in the overhead related to setting each checkpoint. We will name this overhead as a *constant checkpointing overhead* denoted as O_i for process P_i. In general, this overhead is the sum of checkpointing overhead χ_i and the error-detection overhead α_i. Because of the overhead associated with each checkpoint, the actual execution time E_1 of process P_1 is constantly increasing with the number of checkpoints (as shown with thick-margin rectangles around the process P_1 in Fig. 6).

For process P_1 in Fig. 6, going beyond three checkpoints will enlarge the total worst-case execution time $R_1 = S_1 + E_1$, when two faults occur.

In general, in the presence of k faults, the execution time R_i in the worst-case fault scenario of process P_i with n_i checkpoints can be obtained with the equation

$$R_i(n_i) = E_i(n_i) + S_i(n_i) \tag{1}$$

where

$$E_i(n_i) = C_i + n_i \times (\alpha_i + \mathcal{X}_i)$$
$$S_i(n_i) = \left(\frac{C_i}{n_i} + \mu_i\right) \times k + \alpha_i \times (k-1)$$

where $E_i(n_i)$ is the execution time of process P_i with n_i checkpoints in the case of no faults. $S_i\ (n_i)$ is the recovery slack of process P_i. C_i is the worst-case execution time of process P_i. $n_i \times (\alpha_i + \chi_i)$ is the overhead introduced with n_i checkpoints to the execution of process P_i. In the recovery slack S_i (n_i), $C_i/n_i + \mu_i$ is the time needed to recover from a single fault, which has to be multiplied by k for recovering from k faults. The error-detection overhead α_i of process P_i has to be additionally considered in $k-1$ recovered execution segments for detecting possible fault occurrences (except the last, kth, recovery, where all k faults have already happened and been detected).

```
MPAOptimizationStrategy(𝒜, 𝒩)
 1  Step 1:  ψ0 = InitialMPA(𝒜, 𝒩)
 2           if 𝒮0 is schedulable then return ψ0 end if
 3  Step 2:  ψ = TabuSearchMPA(𝒜, 𝒩, ψ)
 4           if 𝒮 is schedulable then return ψ end if
 5  return "system not schedulable"
end MPAOptimizationStrategy
```

Fig. 7. Design optimization strategy for fault tolerance policy assignment. We use Tabu Search for deciding the mapping and fault-tolerance policies, and a list-scheduling-based heuristic for the fault-tolerant schedules.

Let now n_i^0 be the optimal number of checkpoints for P_i, when P_i is considered in isolation. Punnekkat *et al.* [33] derive a formula for n_i^0 in the context of preemptive scheduling and single fault assumption

$$n_i^0 = \begin{cases} n_i^- = \lfloor\sqrt{\frac{C_i}{O_i}}\rfloor, & \text{if } C_i \le n_i^-(n_i^-+1)O_i \\ n_i^+ = \lfloor\sqrt{\frac{C_i}{O_i}}\rfloor, & \text{if } C_i > n_i^-(n_i^-+1)O_i \end{cases} \tag{2}$$

where O_i is a constant checkpointing overhead and C_i is the worst-case execution time of process P_i. The number of checkpoints is an integer, thus we use n_i^- (the floor) or n_i^+ (the ceiling) as a value. If $C_i \le n_i^-(n_i^-+1)O_i$, we select the floor value, since, according to [33], it will lead to a tighter worst-case response time. Otherwise, the ceiling value is used.

We have extended (2) to consider k faults and detailed checkpointing overheads χ_i and α_i for process P_i, when process P_i is considered in isolation (see [16] for the proof of the formula)

$$n_i^0 = \begin{cases} n_i^- = \lfloor\sqrt{\frac{kC_i}{\chi_i+\alpha_i}}\rfloor, & \text{if } C_i \le n_i^-(n_i^-+1)\frac{\chi_i+\alpha_i}{k} \\ n_i^+ = \lfloor\sqrt{\frac{kC_i}{\chi_i+\alpha_i}}\rfloor, & \text{if } C_i > n_i^-(n_i^-+1)\frac{\chi_i+\alpha_i}{k} \end{cases}. \tag{3}$$

Equation (3) allows us to calculate the optimal number of checkpoints for a certain process *considered in isolation*. However, calculating the number of checkpoints for each individual process will not produce a solution which is globally optimal for the whole application because processes share recovery slacks. In general, slack sharing leads to a smaller number of checkpoints associated to processes, or, at a maximum, this number is the same as indicated by the local optima. This is the case because the shared recovery slack, obviously, cannot be larger than the sum of individual recovery slacks of the processes that share it. Therefore, the globally optimal number of checkpoints is always less or equal to the locally optimal number of checkpoints obtained with (3). Thus, (3) provides us with an upper bound on the number of checkpoints associated to individual processes. We will use this equation in order to bound the number of checkpoints explored with the optimization algorithm presented in Fig. 10.

VII. Design Optimization Strategy

The design problem formulated in Section VI is NP complete (both the scheduling and the mapping problems, considered separately, are already NP-complete [12]). Our optimization strategy is outlined in Fig. 7 and has the following two steps.

1) In the first step (lines 1–2), we decide very quickly on an initial configuration ψ^0. This comprises an initial fault-tolerance policy assignment $\mathcal{F}^0$ and mapping $\mathcal{M}^0$. The initial mapping and fault-tolerance policy assignment algo-

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

```
RootScheduleGeneration(A, k, N, B, M)
1  RS = ∅
2  IntroduceOrdering(A) -- process ordering
3  for ∀Pi ∈ A do -- obtaining initial recovery slacks
4     s(Pi) = k × (Xi + μ) + (k – 1) × α
5  end for
6  -- adjusting recovery slacks
7  L = { RootNode(A) }
8  while L ≠ ∅ do
9     p = SelectProcess(L) -- select process from the ready list
10    -- the last scheduled process on the node where p is mapped
11    r = CurrentProcess(RS{M(p)})
12    ScheduleProcess(p, RS{M(p)}) -- scheduling of process p
13    b = start(p) - end(r) -- calculation of the idle time r and p
14    s(p) = max{s(p), s(r) - b} -- adjusting the recovery slack of p
15    -- schedule msgs. sent by p at the end of its recovery slack s
16    ScheduleOutgoingMessages(p, s(p), RS{M(p)})
17    Remove(p, L) -- remove p from the ready list
18    -- add successors of p to the ready list
19    for ∀Succ(p) do
20       if Succ(p) ∉ L then Add(Succ(p), L)
21    end for
22 end while
23 return RS
end RootScheduleGeneration
```

Fig. 8. Generation of root schedules.

rithm (InitialMPA line 1 in Fig. 7) assigns a checkpointing policy to each process in the application $\mathcal{A}$ and produces a mapping that tries to balance the utilization among nodes. The application is then scheduled using the list scheduling-based algorithm, presented in Fig. 8. If the application is schedulable the optimization strategy stops.

2) The second step consists of a Tabu search-based algorithm [38] TabuSearchMPA presented in Fig. 10.

Since we use a meta-heuristic (i.e., Tabu search) for the optimization, we are not guaranteed to find a schedule, even if one exists [38]. Therefore, if after these steps the application is unschedulable, we assume that no satisfactory implementation could be found with the available amount of resources. Moreover, although the transmission times on the bus are taken into account during scheduling, in this paper we do not optimize the access to the communication channel. We have addressed the issue of bus access optimization in [11].

A. Static Scheduling

Once a fault-tolerance policy and a mapping are decided, the processes and messages have to be scheduled. We use a static scheduling algorithm for building the schedule tables for the processes and messages.

Instead of generating all the contingency schedules, only a root schedule is obtained offline. The root schedule consists of start times of processes in the non-faulty scenario and sending times of messages. In addition, it has to provide idle times for process recovering, called recovery slacks. The root schedule is later used by the online scheduler for extracting the execution scenario corresponding to a particular fault occurrence. Such an approach significantly reduces the amount of memory required to store schedule tables.

The algorithm for the generation of root schedules is presented in Fig. 8 and takes as input the application $\mathcal{A}$, the number k of transient faults that have to be tolerated, the architecture consisting of computation nodes $\mathcal{N}$ and bus B, the mapping $\mathcal{M}$, and produces the root schedule $\mathcal{RS}$. We use a list-scheduling based algorithm to generate the root schedule. List scheduling heuristics are based on priority lists from which processes are extracted (line 9) in order to be scheduled (line 12) at certain moments. A process P_i is placed in the ready list $\mathcal{L}$ if all its predecessors have been already scheduled (lines 19–21). All ready processes from the list $\mathcal{L}$ are investigated, and that process P_i is selected for placement in the schedule which has the highest priority. Similarly, messages are scheduled on the bus. We use the "modified partial critical path" priority function (MPCP) presented in [11]. The list scheduling algorithm loops until the ready list $\mathcal{L}$ is empty (lines 8–22). Next, we will outline how the basic list-scheduling approach has been extended to generate the fault-tolerant schedules.

At first, the order of process execution is introduced with the MPCP priority function (line 2). Initial recovery slacks for all processes $P_i \in \mathcal{A}$ are calculated as $s_0(P_i) = k \times (C_i + \mu) + (k-1) \times \alpha$ (lines 3–5). Then, recovery slacks of processes mapped on the same computation node are merged to reduce timing overhead.

The process graph $\mathcal{A}$ is traversed starting from the root node (line 7). Process p is selected from the ready list $\mathcal{L}$ according to the priority function (line 9). The last scheduled process r on the computation node, on which p is mapped, is extracted from the root schedule $\mathcal{S}$ (line 11). Process p is scheduled and its start time is recorded in the root schedule. Then, its recovery slack $s\,(p)$ is adjusted such that it can accommodate recovering of processes scheduled before process p on the same computation node (lines 13–14). The adjustment is performed in the following two steps.

1) The idle time b between process p and the last scheduled process r is calculated (line 13).
2) The recovery slack $s(p)$ of process p is changed, if recovery slack $s(r)$ of process r subtracted with the idle time b is larger than the initial slack $s_0(p)$. Otherwise, the initial slack $s_0(p)$ is preserved as $s(p)$ (line 14).

If no process is scheduled before p, the initial slack $s_0(p)$ is preserved as $s(p)$. Outgoing messages sent by process p are scheduled at the end of recovery slack $s(p)$ (line 16).

After the adjustment of the recovery slack, process p is removed from the ready list $\mathcal{L}$ (line 17) and its successors are added to the list (lines 19–21). After scheduling of all the processes in the application graph $\mathcal{A}$, the algorithm returns a root schedule $\mathcal{RS}$ with start times of processes, sending times of messages, and recovery slacks (line 23).

During list scheduling, the notion of "ready process" will be different in the case of processes waiting inputs from replicas. In that case, a successor process P_s of a replicated process P_i can be placed in the root schedule at the *earliest* time moment t, at which at least one valid message $m_{i(j)}$ can arrive from a replica $P_{i(j)}$ of process P_i.[5] We also include in the set of valid messages $m_{i(j)}$ the output from replica $P_{i(j)}$ to successor P_s passed through the shared memory (if replica $P_{i(j)}$ and successor P_s are mapped on the same computation node) [16].

The scheduling algorithm is responsible for deriving offline the root schedules. The contingency schedules are determined *online*, in linear time, by the scheduler in each node, based on the occurrences of faults. The overhead due to obtaining the new start times is considered at the schedule table construction [16].

[5]We consider the original process P_i as a first replica, denoted with $P_{i(1)}$.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

```
1  -- given a merged graph 𝒢, an architecture 𝒩 produces a policy
2  -- assignment ℱ and a mapping ℳ such that 𝒢 is fault-tolerant
     & schedulable
3  x^best = x^now = ψ; BestCost = ListScheduling(𝒢, 𝒩, x^best)
4  Tabu = ∅; Wait = ∅ -- The selective history is initially empty
5  while x^best not schedulable ∧ TerminationCond not satisfied do
6    -- Determine the neighboring solutions considering history
7    𝒞𝒫 = CriticalPath(𝒢); N^now = GenerateMoves(𝒞𝒫)
8    -- cut tabu moves if they are not better than best-so-far
9    N^tabu = {move(P_i) | ∀ P_i ∈ 𝒞𝒫 ∧ Tabu(P_i)=0 ∧
     Cost(move(P_i)) < BestCost}
10   N^non-tabu = N \ N^tabu
11   -- add diversification moves
12   N^waiting = {move(P_i) | ∀ P_i ∈ 𝒞𝒫 ∧ Wait(P_i) > |𝒢|}
13   N^now = N^non-tabu ∪ N^waiting
14   -- Select a solution based on aspiration criteria
15   x^now = SelectBest(N^now);
16   x^waiting = SelectBest(N^waiting); x^non-tabu = SelectBest(N^non-tabu)
17   if Cost(x^now) < BestCost then x = x^now -- select x^now if better
18     else if ∃ x^waiting then x = x^waiting -- otherwise diversify
19     else x = x^non-tabu -- if no better and no diversification,
       select best non-tabu
20   end if
21   -- Perform selected move
22   PerformMove(x); Cost = ListScheduling(𝒢, 𝒩, x)
23   -- Update the best-so-far solution and the history tables
24   If Cost < BestCost then x^best = x; BestCost = Cost end if
25   Update(Tabu); Update(Wait)
26 end while
27 return x^best
end TabuSearchMPA
```

Fig. 9. Tabu search algorithm for optimization of mapping and fault tolerance policy assignment.

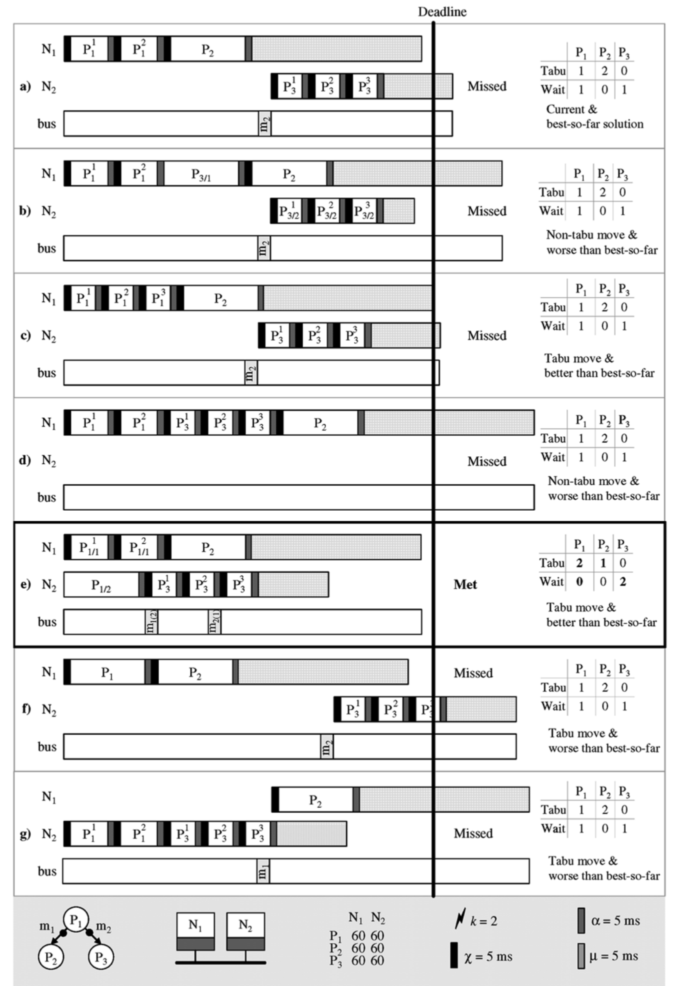


Fig. 10. Moves and Tabu History. Considering (a) as the current solution, the neighborhood is formed by (b)–(e).

B. Mapping and Fault-Policy Assignment

For the optimization of process mapping and fault-policy assignment we are using a tabu search-based approach, TabuSearchMPA (see Fig. 9).

The tabu search takes as an input the merged application graph $\mathcal{G}$ (see Section IV), the architecture $\mathcal{N}$ and the current implementation ψ, and produces a schedulable and fault-tolerant implementation x^{best}. The tabu search is based on a neighborhood search technique, and thus in each iteration it generates the set of moves N^{now} that can be reached from the current solution x^{now} (line 7 in Fig. 9). In our implementation, we only consider changing the mapping or fault-tolerance policy assignment of the processes on the critical path, corresponding to the current solution, denoted with $\mathcal{CP}$ in Fig. 9. We define the critical path as the path through the merged graph $\mathcal{G}$ which corresponds to the longest delay in the schedule table. For example, in Fig. 10(a), the critical path is P_1, m_2, and P_3.

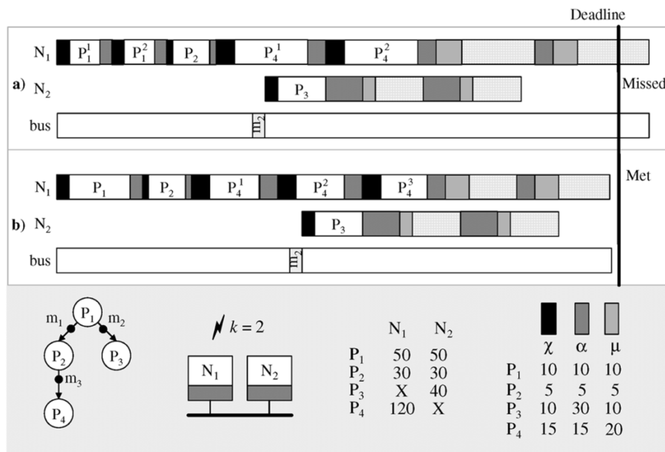
Tabu search uses design transformations (moves) to change a design such that the critical path is reduced. Let us consider the example in Fig. 10, where we have an application of four processes that has to tolerate one fault, mapped on an architecture of two nodes. Let us assume that the current solution is the one depicted in Fig. 10(a). In order to generate neighboring solutions, we consider three types of moves: 1) remapping moves; 2) policy-assignment moves; and 3) checkpointing-change moves.

1) Each remapping move is the remapping of a process P_i (or its replica), mapped on a computation node N_k, to another computation node $N_j \neq N_k$. The number of checkpoints is not changed by the remapping move.
2) Each policy-assignment move applied to a process P_i changes the combination of replication/checkpointing required to tolerate the faults. This move will not change the number of checkpoints assigned to a process.
3) Each checkpointing move changes the number of checkpoints for a process P_i from n_i^k to $n_i^j \neq n_i^k$, while the mapping is not affected. We bound the maximum number of checkpoints that can be assigned to a process P_i to n_i^0, the locally-optimal number calculated with (3).

Let us illustrate how the neighbor solutions are generated. In Fig. 10(a), we use only recovery with checkpointing: P_1 has two checkpoints, P_2 one, and P_3 has three. Starting from this solution, the generated neighbor solutions are presented in Fig. 10(b)–(g): in (b) process P_3 is replicated, thus its recovery slack on node N_2 can be reduced; in (c) an additional checkpoint is introduced in P_1; in (d) P_3 is mapped on N_1; in (e) P_1 is replicated, thus P_3, its descendant, can receive m_2 directly from P_1's replica in case no error occurs; in (f) we remove a checkpoint from P_1; finally, in (g) P_1 is mapped on N_2.

During the generation of neighbor solutions, we try to eliminate moves that change the number of checkpoints if it is clear that they do not lead to better results. Consider the example in Fig. 11 where we have four processes, P_1 to P_4 mapped on two

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.



	N_1	N_2
P_1	50	50
P_2	30	30
P_3	X	40
P_4	120	X

	χ	α	μ
P_1	10	10	10
P_2	5	5	5
P_3	10	30	10
P_4	15	15	20

Fig. 11. Restricting the moves for setting the number of checkpoints. We will not increase the checkpoints for P_2 nor reduce for P_4, since they clearly do not lead to better solutions.

nodes, N_1 and N_2. The worst-case execution times of processes and their fault-tolerance overheads are also given in the figure, and we can have at most two faults. The number of checkpoints calculated using (3) are: $n_1^0 = 2, n_2^0 = 2, n_3^0 = 1$ and $n_4^0 = 3$. Let us assume that our current solution is the one depicted in Fig. 11(a), where we have $\mathcal{X}(P_1) = 2, \mathcal{X}(P_2) = 1, \mathcal{X}(P_3) = 1$ and $\mathcal{X}(P_4) = 2$. Given a process P_i, with a current number of checkpoints $\mathcal{X}(P_i)$, our tabu search approach will generate moves with all possible checkpoints starting from 1, up to n_i^0. Thus, starting from the solution depicted in Fig. 11(a), we can have the following moves that modify the number of checkpoints: 1) decrease the number of checkpoints for P_1 to 1; 2) increase the number of checkpoints for P_2 to 2; 3) increase the number of checkpoints for P_4 to 3; and 4) decrease the number of checkpoints for P_4 to 1. Moves (1) and (3) will lead to the optimal number of checkpoints depicted in Fig. 11(b).

In order to reduce optimization time, our heuristic will not try moves (2) and (4), since they cannot lead to a shorter critical path, and, thus, a better root schedule. Regarding move (2), by increasing the number of checkpoints for P_2 we can reduce its recovery slack. However, P_2 shares its recovery slack with P_1 and segments of P_4, which have a larger execution time, and thus even if the necessary recovery slack for P_2 is reduced, it will not affect the size of the shared slack (and implicitly, of the root schedule) which is given by the largest process (or process segment) that shares the slack. Regarding move (4), we notice that by decreasing for P_4 the number of checkpoints to 1, we increase the recovery slack, which, in turn, increases the length of the root schedule.

The key feature of a tabu search is that the neighborhood solutions are modified based on a selective history of the states encountered during the search. This means that the neighborhood of x^{now} is not a static set, but rather a set that can change according to the history of the search. The selective history is implemented in our case through the use of two vectors, *Tabu* and *Wait*. Basically, the purpose of the *Tabu* vector is to prevent certain solutions (i.e., *tabu-active* solutions) from the recent past to be revisited, while the purpose of the *Wait* vector is to diversify the search by applying moves to a process that has waited for a long time. Thus, each process has an entry in these tables. Initially, all entries are set to 0. Let us consider that the current solution is the one in Fig. 10(a), and that the search so far has led to the values in the *Tabu* and *Wait* tables as depicted in Fig. 10(a). If $\text{Tabu}(P_i)$ is non-zero, it means that the process is "tabu", i.e., should not be selected for generating moves. This is the case with processes P_1 and P_2, while any moves concerning P_3 are not tabu. However, tabu moves are accepted if they are better than the best-so-far solution. Such situations are presented in Figs. 10(c) and (e), where moves concerning P_1 are applied because they lead to better solutions, although P_1 is tabu. Regarding the *Wait* vector, if *Wait* (P_i) is greater than the number of processes in the graph $|\mathcal{G}|$ (a value determined experimentally), we consider that the process has waited a long time and should be selected for diversification.[6] Let us consider the algorithm in Fig. 9.

- A move will be removed from the neighborhood solutions if it is tabu (lines 9 and 10 of the algorithm):
- In lines 12–13, the search is diversified with moves which have waited a long time without being selected.
- In lines 14–20, we select the best one out of the neighborhood solutions. We prefer a solution that is better than the best-so-far x^{best} (line 17). If such a solution does not exist, then we choose to diversify. If there are no diversification moves, we simply choose the best solution found in this iteration, even if it is not better than x^{best}.
- To further improve diversification, we generate neighbors using a different objective function. Thus, every time we have performed more than $|\mathcal{V} + \mathcal{V}_R| \times |\mathcal{N}|/2$ moves (the number of processes multiplied with the number of resources and divided by two—a heuristic value determined experimentally), we generate solutions using an objective criterion which tries to balance the utilization of resources. This is achieved by visiting each process, and assigning it to the least-utilized processor so far.
- The algorithm updates the best-so-far solution, and the selective history tables *Tabu* and *Wait*. The *Tabu* entries are decremented if the corresponding tabu move has not been currently selected, and incremented if it was selected (a tabu move can be selected if it is better than the best-so-far). For example, starting from Fig. 10(a) to we generate the solution in Fig. 10(e) by introducing replication for P_1. Thus, we increment $\text{Tabu}(P_1)$ from 1 to 2 and we decrement $\text{Tabu}(P_2)$ from 2 to 1, because the change does not refer to P_2. The *Wait* entries are incremented if moves referring to them are not applied, and set to 0 otherwise. For the same example, we reset $Wait(P_1)$ and increment $\text{Wait}(P_3)$ from 1 to 2. Note that $\text{Wait}(P_2)$ is not incremented because P_2 is tabu and hence will not be chosen for diversification anyway.
- Finally, the algorithm ends when a schedulable solution has been found, or an imposed termination condition has been satisfied (e.g., if a time-limit has been reached).

Fig. 10 illustrates how the algorithm works. Let us consider that the current solution x^{now} is the one presented in Fig. 10(a), with the corresponding selective history presented to its right, and that this is also the best-so-far solution, denoted with x^{best}.

[6]Diversification moves are driving the exploration into yet unexplored neighborhoods.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

No solution is removed from the set of considered solutions because none is simultaneously tabu and worst than x^{best}. Thus, all solutions (b)–(g) are evaluated in the current iteration. Out of these, the solutions in Fig. 10(c) and (e) are considered because although they are tabu, they are better than x^{best}. Out of these two, the solution in Fig. 10(e) is selected because it meets the deadline. The table is updated as depicted to the right of Fig. 10(e) in bold and the iterations continue with solution (e) as the current solution.

VIII. EXPERIMENTAL RESULTS

Unless otherwise stated, we have used the following experimental setup for the evaluation of our algorithms. We have used synthetic applications of 20, 40, 60, 80, and 100 processes (all unmapped and with no fault-tolerance policy assigned) implemented on architectures consisting of 3, 4, 5, 6, and 7 nodes, respectively. We have generated both graphs with random structure and graphs based on more regular structures, such as trees, and groups of chains. Execution times and message lengths were assigned randomly using both uniform and exponential distribution within the 10 to 100 ms, and 1 to 4 bytes ranges, respectively. We have varied the number of maximum tolerated faults depending on the architecture size, considering 4, 5, 6, 7, and 8 faults for each architecture dimension, respectively. The recovery overhead μ has been set to 5 ms. We have also varied the fault tolerance overheads (checkpointing and error detection) for each process, from 1% of its worst-case execution time up to 30%. Fifteen examples were randomly generated for each application dimension, thus a total of 75 applications were used for each experiment presented next. During the experiments, we have derived the shortest schedule within an imposed time limit for optimization: 1 minute for 20 processes, 10 for 40, 30 for 60, 2 h and 30 min for 80 and 6 h for 100 processes. We have also used a case study consisting of a cruise controller. The experiments were performed on Sun Fire V250 computers.

We were interested to evaluate the proposed optimization strategy with regard to the overheads, in terms of schedule length, introduced due to fault-tolerance. For this, we have implemented each application without any fault-tolerance concerns. This non-fault-tolerant implementation, NFT, has been obtained using an approach similar to the algorithm in Fig. 7 but without any fault-tolerance considerations. Then, we have implemented each application on its corresponding architecture, using the MPAOptimizationStrategy from Fig. 7. In the first set of experiments, we have considered a single checkpoint per process, which is similar to simple re-execution. Let us call this optimization strategy MXR.

The first results are presented in Table I. In the three last columns, we present maximum, average, and minimum time overheads introduced by MXR compared to NFT. Let δ_{MXR} and δ_{NFT} be the schedule lengths obtained using MXR and NFT. The overhead due to introduced fault tolerance is defined as $100 \times (\delta_{\text{MXR}} - \delta_{\text{NFT}})/\delta_{\text{NFT}}$. We can see that the fault tolerance overheads grow with the application size. The MXR approach can offer fault tolerance within the constraints of the architecture at an average time overhead of approximately 100%. However, even for applications of 60 processes, there are cases where the overhead is as low as 52%.

TABLE I
FAULT TOLERANCE OVERHEADS DUE TO MXR (COMPARED TO NFT)

Number of processes	k	% maximum	% average	% minimum
20	3	98.36	70.67	48.87
40	4	116.77	84.78	47.30
60	5	142.63	99.59	51.90
80	6	177.95	120.55	90.70
100	7	215.83	149.47	100.37

TABLE II
FAULT TOLERANCE OVERHEADS DUE TO MXR FOR DIFFERENT NUMBER OF FAULTS

k	% maximum	% average	% minimum
2	52.44	32.72	19.52
4	110.22	76.81	46.67
6	162.09	118.58	81.69
8	250.55	174.07	117.84
10	292.11	219.79	154.93

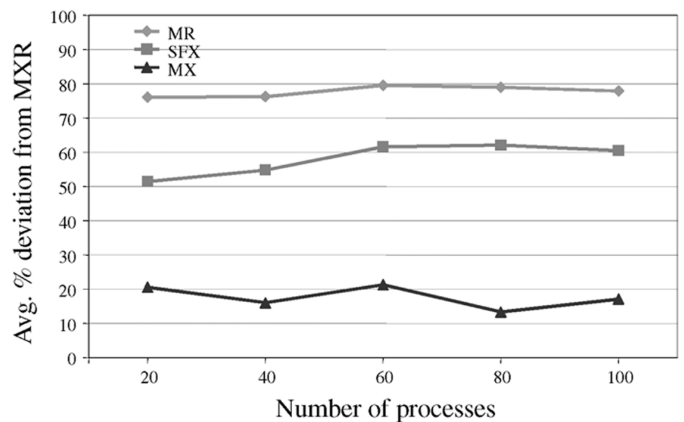


Fig. 12. Comparing MXR with MX, MR, and SFX.

We were also interested to evaluate our MXR approach in the case of different number of faults, while the application size and the number of computation nodes were fixed. We have considered applications with 60 processes mapped on four computation nodes, with the number k of faults being 2, 4, 6, 8, or 10. Table II shows that the time overheads due to fault tolerance increase with the number of tolerated faults. This is expected, since we need more replicas and/or re-executions if there are more faults.

Next, we were interested to evaluate the quality of our MXR optimization approach. Together with the MXR approach we have also evaluated two extreme approaches: MX that considers only one checkpoint (similar to re-execution), and MR which relies only on replication for tolerating faults. MX and MR use the same optimization approach as MRX, but, for fault tolerance, all processes are assigned only with re-execution or replication, respectively. In Fig. 12, we present the average percentage deviations of the MX and MR from MXR in terms of overhead. We can see that by optimizing the combination of re-execution and replication, MXR performs much better compared to both MX

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

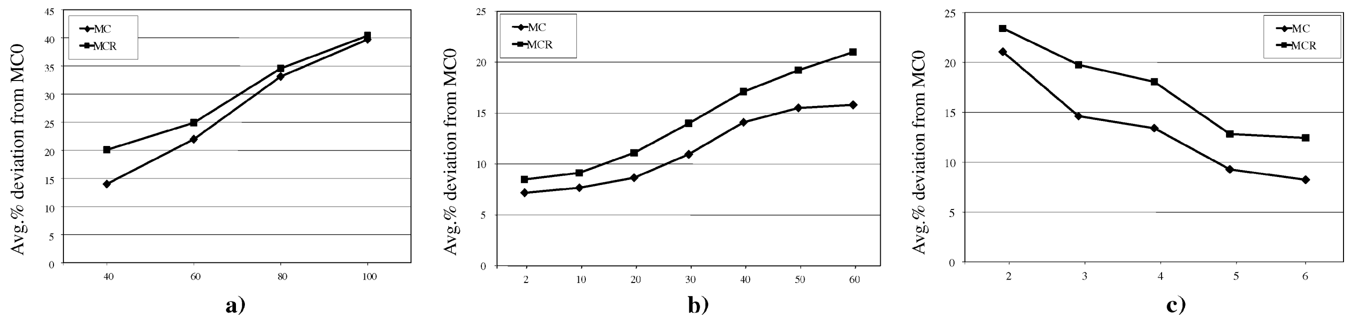


Fig. 13. MCR and MC compared to MC0. (a) Number of processes. (b) Checkpointing overheads. (c) Number of transient faults.

and MR. On average, MXR is 77% and 17.6% better than MR and MX, respectively. This shows that considering two redundancy techniques simultaneously (in this case, re-execution at the same time with replication) can lead to significant improvements.

In Fig. 12, we have also presented a straightforward strategy SFX, which first derives a mapping without fault-tolerance considerations (using MXR without fault-tolerance moves) and then applies re-execution. The re-execution is similar to a single-checkpoint scheme. The main idea here is that SFX separates the mapping and re-execution assignment steps (which are done simultaneously in MX; MXR introduces also redundancy). Thus, although re-execution is applied in an efficient way, by sharing the recovery slacks and thus trying to minimize the schedule length, the mapping will not be changed. This is a solution that can be obtained by a designer without the help of our fault-tolerance optimization tools. We can see that the overheads thus obtained are very large compared to MXR, up to 58% on average. We can also notice that, despite the fact that both SFX and MX use only re-execution, MX is much better. This confirms that the optimization of the fault-tolerance policy assignment has to be addressed at the same time with the mapping of functionality.

In the second set of experiments we have concentrated on evaluating the checkpoint optimization. We were interested to compare the quality of MCR to MC0 and MC. In Fig. 13, we show the average percentage deviation of overheads obtained with MCR and MC from the baseline represented by MC0 (larger deviation means smaller overhead). From Figs. 13(a) and (b), we can see that by optimizing the combination of checkpointing and replication MCR performs much better compared to MC and MC0. This shows that considering checkpointing at the same time with replication can lead to significant improvements. Moreover, by considering the global optimization of the number of checkpoints, with MC, significant improvements can be gained over MC0 which computes the optimal number of checkpoints for each process in isolation.

In Fig. 13(a), we consider 4 processors, 3 faults, and vary the application size from 40 to 100 processes. As the amount of available resources per application decreases, the improvement due to replication (part of MCR) will diminish, leading to a result comparable to MC.

In Fig. 13(b), we were interested to evaluate our MCR approach in the case the checkpointing overheads (i.e., $\chi + \alpha$) are varied. We have considered applications with 40 processes mapped on four processors, and we have varied the checkpointing overheads from 2% of the worst-case execution time of a process up to 60%. We can see that as the amount of checkpointing overheads increases, our optimization approaches are able to find increasingly better quality solutions compared to MC0.

We have also evaluated the MCR and MC approaches in the case the maximum number of transient faults to be tolerated increases. We have considered applications with 40 processes mapped on 4 processors, and varied k from 2 to 6, see Fig. 13(c). As the number of faults increase, the improvement, compared to MC0 decreases, and will stabilize to about 10% improvement (e.g., for $k = 10$ the improvement due to MC is 8.30%, while MCR improves with 10.29%).

The experiments demonstrate that it is very important to consider checkpointing at the same time with replication, and to optimize the number of checkpoints globally.

A. Case Study

A typical safety critical application with hard real-time constraints, to be implemented on a distributed architecture, is a vehicle cruise controller (CC). We have considered a CC system derived from a specification provided by the industry.

The CC described in this specification delivers the following functionality: 1) it maintains a constant speed for speeds over 35 km/h and under 200 km/h; 2) offers an interface (buttons) to increase or decrease the reference speed; 3) is able to resume its operation at the previous reference speed; and 4) the CC operation is suspended when the driver presses the brake pedal.

The process graph that models the CC has 32 processes, and is described in [32]. The CC was mapped on an architecture consisting of three nodes: electronic throttle module (ETM), anti-lock breaking system (ABS), and transmission control module (TCM). We have considered a deadline of 260 ms, $k = 2$ faults and the checkpointing overheads are 10% of the worst-case execution time of the processes.

In this setting, the MCR produced a schedulable fault-tolerant implementation with a worst-case system delay of 230 ms, and with an overhead compared to NFT (which produces a non-fault-tolerant schedule of length 136 ms) of 69%. If we globally optimize the number of checkpoints using MC, we obtain a schedulable implementation with a delay of 256 ms, compared to 276 ms produced by MC0 which is larger than the deadline. If replication only is used, as in the case of MR, the delay is 320 ms, which is greater than the deadline.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

IX. Conclusion

In this paper, we have considered hard real-time systems, where the hardware architecture consists of a set of heterogeneous computation nodes connected to a communication channel. The real-time application is represented as a set of processes communicating by messages. The processes are scheduled by static cyclic scheduling. To provide fault tolerance against transient faults, processes are assigned with replication or recovery with checkpointing.

Scheduling: We have proposed a scheduling approach for fault-tolerant embedded systems in the presence of multiple transient faults, based on list-scheduling. The scheduler on each node determines the new start times considering the root schedule and the current fault occurrence scenario.

Mapping and Fault Tolerance Policy Assignment: Regarding fault tolerance policy assignment, we decide on which fault tolerance technique or which combination of techniques to assign to a certain process in the application. The fault tolerance technique can be either rollback recovery, which provides time-redundancy, or active replication, which provides space-redundancy. We have implemented a tabu search-based optimization approach that decides the mapping of processes to the nodes in the architecture and the assignment of fault-tolerance policies to processes.

Checkpoint Optimization: We have also addressed the problem of checkpoint optimization. We have shown that by globally optimizing the number of checkpoints, as opposed to the approach when processes were considered in isolation, significant improvements can be achieved. We have also integrated checkpoint optimization into a fault tolerance policy assignment and mapping optimization strategy, and an optimization algorithm based on tabu-search has been implemented.

All proposed algorithms have been implemented and evaluated on numerous synthetic applications and a real-life example from automotive electronics. The results obtained have shown the efficiency of the proposed approaches and methods.

References

[1] K. Ahn, J. Kim, and S. Hong, "Fault-tolerant real-time scheduling using passive replicas," in *Proc. Pacific Rim Int. Symp. Fault-Tolerant Syst.*, 1997, pp. 98–103.

[2] R. Al-Omari, A. K. Somani, and G. Manimaran, "A new fault-tolerant technique for improving schedulability in multiprocessor real-time systems," in *Proc. 15th Int. Parallel Distrib. Process. Symp.*, 2001, pp. 23–27.

[3] A. Bertossi and L. Mancini, "Scheduling algorithms for fault-tolerance in hard-real time systems," *Real Time Syst.*, vol. 7, no. 3, pp. 229–256, 1994.

[4] A. Burns, R. I. Davis, and S. Punnekkat, "Feasibility analysis for fault-tolerant real-time task sets," in *Proc. Euromicro Workshop Real-Time Syst.*, 1996, pp. 29–33.

[5] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[6] P. Chevochot and I. Puaut, "Scheduling fault-tolerant distributed hard-real time tasks independently of the replication strategies," in *Proc. Real-Time Comput. Syst. Appl. Conf.*, 1999, pp. 356–363.

[7] V. Claeson, S. Poldena, and J. Söderberg, "The XBW model for dependable real-time systems," in *Proc. Parallel Distrib. Syst. Conf.*, 1998, pp. 130–138.

[8] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[9] J. Conner *et al.*, "FD-HGAC: A hybrid heuristic/genetic algorithm hardware/software co-synthesis framework with fault detection," in *Proc. Asia South Pacific Des. Autom. Conf. (ASP-DAC)*, 2005, pp. 709–712.

[10] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[11] P. Eles, A. Doboli, P. Pop, and Z. Peng, "Scheduling with bus access optimization for distributed embedded systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 5, pp. 472–491, Sep. 2000.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 2003.

[13] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel, "An algorithm for automatically obtaining distributed and fault-tolerant static schedules," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2003, pp. 159–168.

[14] C. C. Han, K. G. Shin, and J. Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *IEEE Trans. Computers*, vol. 52, no. 3, pp. 362–372, Mar. 2003.

[15] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai, "Impact of CMOS process scaling and SOI on the soft error rates of logic processes," in *Proc. Symp. VLSI Technol.*, 2001, pp. 73–74.

[16] V. Izosimov, "Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems," Licentiate Thesis No. 1277, Dept. Comput. Inf. Sci., Linköping University, Linköping, Sweden, 2006.

[17] J.-J. Han and Q.-H. Li, "Dynamic power-aware scheduling algorithms for real-time task sets with fault-tolerance in parallel and distributed computing environment," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2005, pp. 6–16.

[18] J. Xu and B. Randell, "Roll-forward error recovery in embedded real-time systems," in *Proc. Int. Conf. Parallel Distrib. Syst.*, 1996, pp. 414–421.

[19] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Trans. Computers*, vol. 52, no. 2, pp. 113–125, Feb. 2003.

[20] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Dependable communication synthesis for distributed embedded systems," in *Proc. Comput. Safety, Reliab. Security Conf.*, 2003, pp. 275–288.

[21] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The mars approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, Jan. 1989.

[22] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating transient faults in MARS," in *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, 1990, pp. 466–473.

[23] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proc. IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.

[24] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri, "From a federated to an integrated architecture for dependable embedded real-time systems," Technische Univ. Wien, Vienna, Austria, Tech. Rep. 22, 2004.

[25] C. M. Krishna and A. D. Singh, "Reliability of checkpointed real-time systems using time redundancy," *IEEE Trans. Reliab.*, vol. 42, no. 3, pp. 427–435, Sep. 1993.

[26] H. Lee, H. Shin, and S.-L. Min, "Worst case timing requirement of real-time tasks with time redundancy," in *Proc. 6th Int. Conf. Real-Time Comput. Syst. Appl.*, 1999, pp. 410–414.

[27] F. Liberato, R. Melhem, and D. Mosse, "Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems," *IEEE Trans. Computers*, vol. 49, no. 9, pp. 906–914, Sep. 2000.

[28] A. Maheshwari, W. Burleson, and R. Tessier, "Trading off transient fault tolerance and power consumption in deep submicron (DSM) VLSI circuits," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 3, pp. 299–311, Mar. 2004.

[29] A. Orailoglu and R. Karri, "Coactive scheduling and checkpoint determination during high level synthesis of self-recovering microarchitectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 3, pp. 304–311, Sep. 1994.

[30] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications," in *Proc. Des. Autom. Test Eur. Conf.*, 2004, pp. 1164–1169.

[31] S. Poldena, *Fault Tolerant Systems—The Problem of Replica Determinism*. Norwell, MA: Kluwer, 1996.

[32] P. Pop, P. Eles, and Z. Peng, *Analysis and Synthesis of Distributed Real-Time Embedded Systems*. Norwell, MA: Kluwer, 2004.

[33] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Syst. J.*, vol. 20, no. 1, pp. 83–102, 2001.

[34] P. Puschner and A. Burns, "A review of worst-case execution-time analyses," *Real-Time Syst. J.*, vol. 18, no. 2/3, May 2000.

[35] J. Sosnowski, "Transient fault tolerance in digital systems," *IEEE Micro*, vol. 14, no. 1, pp. 24–35, Jan. 1994.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.

[36] S. Srinivasan and N. K. Jha, "Hardware-software co-synthesis of fault-tolerant real-time distributed embedded systems," in *Proc. Eur. Des. Autom. Conf.*, 1995, pp. 334–339.

[37] B. Strauss, M. G. Morgan, J. Apt, and D. D. Stancil, "Unsafe at any airspeed?," *IEEE Spectrum*, vol. 43, no. 3, pp. 44–49, Mar. 2006.

[38] C. R. Reevs*, Modern Heuristic Techniques for Combinatorial Problems*. Oxford, U.K.: Blackwell, 1993.

[39] D. Rossi, M. Omaha, F. Toma, and C. Metra, "Multiple transient faults in logic: An issue for next generation ICs?," in *Proc. 20th IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, 2005, pp. 352–360.

[40] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Reliability-aware co-synthesis for embedded systems," in *Proc. 15th IEEE Int. Conf. Appl.-Specific Syst., Arch. Processors*, 2004, pp. 41–50.

[41] Y. Zhang, R. Dick, and K. Chakrabarty, "Energy-aware deterministic fault tolerance in distributed real-time embedded systems," in *Proc. 42nd Des. Autom. Conf.*, 2004, pp. 550–555.

[42] Y. Zhang and K. Chakrabarty, "A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, no. 1, pp. 111–125, Jan. 2006.

[43] D. Zhu, R. Melhem, and D. Mossé, "Energy efficient configuration for QoS in reliable parallel servers," *Lecture Notes Comput. Sci.*, vol. 3463, pp. 122–139, 2005.

Paul Pop (M'99) received the Ph.D. degree in computer systems from Linköping University, Linköping, Sweden, in 2003.

He is an Associate Professor with the Informatics and Mathematical Modelling Department, Technical University of Denmark, Copenhagen, Denmark. He is active in the area of analysis and design of real-time embedded systems, where he has published extensively and coauthored several book chapters and one book.

Prof. Pop was a recipient of the Best Paper Award from the Design, Automation, and Test in Europe Conference (DATE 2005) and from the Real-Time in Sweden Conference (RTiS 2007), and was nominated for the Best Paper Award from the Design Automation Conference (DAC 2001).

Viacheslav Izosimov (S'03) received the Qualified Engineer degree in computer science (with honor) from St. Petersburg State University of Telecommunications, St. Petersburg, Russian Federation, in 2002, the M.Sc. degree in information processing and telecommunications from Lappeenranta University of Technology, Lappeenranta, Finland, in 2003 (IMPIT-program), and the Licentiate degree in computer systems from Linköping University (LiU), Linköping, Sweden, in 2006, where he is currently pursuing the Ph.D. degree in computer and information science.

His research deals with design optimization and scheduling of fault-tolerant distributed embedded systems.

Mr. Izosimov was a recipient of the Best Paper Award from the Design, Automation and Test in Europe Conference (DATE 2005).

Petru Eles (M'99) received the Ph.D. degree in computer science from the Politehnica University of Bucharest, Bucharest, Romania, in 1993.

He is currently a Professor with the Department of Computer and Information Science, Linköping University, Linköping, Sweden. His research interests include embedded systems design, hardware-software codesign, real-time systems, system specification and testing, and CAD for digital systems. He has published extensively in these areas and coauthored several books.

Prof. Eles was a corecipient of the Best Paper Awards from the European Design Automation Conference in 1992 and 1994, and from the Design Automation and Test in Europe Conference in 2005, and of the Best Presentation Award from the 2003 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and of the *IEE Proceedings—Computers and Digital Techniques*. He has served as a Program Committee member for numerous international conferences in the areas of Design Automation, Embedded Systems, and Real-Time Systems, and as a TPC chair and General chair of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis.

Zebo Peng (M'91–SM'02) received the B.Sc. degree in computer engineering from the South China Institute of Technology, China, in 1982, and the Licentiate of Engineering and Ph.D. degrees in computer science from Linköping University, Linköping, Sweden, in 1985 and 1987, respectively.

He is currently a Full Professor of Computer Systems, Director of the Embedded Systems Laboratory, and Chairman of the Division for Software and Systems with the Department of Computer Science, Linköping University. He is also the Director of the National Graduate School of Computer Science in Sweden. His current research interests include design and test of embedded systems, electronic design automation, SoC testing, design for testability, hardware/software co-design, and real-time systems. He has published more than 200 technical papers, and coauthored books.

Prof. Peng was a corecipient of two Best Paper Awards from the European Design Automation Conferences (1992 and 1994), a Best Paper Award from the IEEE Asian Test Symposium (2002), a Best Paper Award from the Design Automation and Test in Europe Conference (2005), and a Best Presentation Award from the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (2003). He serves currently as Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, *VLSI Design Journal*, and the *EURASIP Journal on Embedded Systems*. He has served each year on the program committee of a dozen international conferences and workshops, including ATS, DATE, DDECS, DFT, DTIS, ETS, ITSW, LATW, MEMOCDE, and VLSI-SOC. He was the Program Chair of DATE'08. He is the Chair of the IEEE European Test Technology Technical Council (ETTTC), and has been a Golden Core Member of the IEEE Computer Society since 2005.

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on October 20, 2009 at 11:03 from IEEE Xplore. Restrictions apply.