A SAFE Approach towards Early Design Space Exploration of Fault-tolerant Multimedia MPSoCs

Peter van Stralen, Andy Pimentel
Department of Computer Science
University of Amsterdam, The Netherlands
{p.vanstralen,a.d.pimentel}@uva.nl

ABSTRACT

With the reduction in feature size, transient errors start to play an important role in modern embedded systems. It is therefore important to make fault-tolerance a first-class citizen in embedded system design. Fault-tolerance patterns are techniques to make an application fault-tolerant. Not only do fault-tolerance patterns affect the quality of the embedded system (like performance, energy and cost), but there also are many ways of applying them. In this paper, we present the SAFE simulation framework that supports the early exploration of the different possibilities to apply fault-tolerance patterns to MPSoC-based embedded multimedia systems. The SAFE model incorporates fault injection, detection and correction. As a result, a Pareto front can be obtained that not only shows the trade-off between metrics like performance, energy, cost, but also captures reliability metrics like frame drops due to soft errors and the number of unresolvable faults.

Categories and Subject Descriptors

J.6 [**Computer-aided Engineering**]: Computer-aided design

Keywords

MPSoC, Design Space Exploration, Simulation, Fault-Tolerance

1. INTRODUCTION

The design of Multi-Processor System-on-Chip (MPSoC) based embedded systems deals with many objectives. One of them is reliability. An MPSoC design needs to be able to cope with soft and hard errors. Soft errors are transient errors that cause a temporal malfunction in the system. There are multiple sources of soft errors. Examples are single upset events (SUE) [23] caused by high energy neutrons resulting from cosmic rays colliding with particles in the atmosphere, and negative bias temparature instability (NBTI) [1]. In general, soft errors are failures in processor execution due to electrical noise or external radiation. Traditionally, soft errors were only an issue in electronic circuits used in space. However, due to the reduction in feature size and voltage levels, MPSoCs are becoming more susceptible to soft errors [23]. Therefore, an MPSoC-based embedded system needs to be able to cope with these errors.

Fault-tolerance patterns allow an MPSoC to deal with system errors. One of the possible fault-tolerance patterns is active redundancy in space and/or time. If active redundancy is used in the space domain, different resources are used to run the same tasks by means of task replication. Another possibility is to run the task multiple times on the same resource (time domain). The outcomes of the different runs are collected (no response within a certain time frame is also a response) and compared. Based on these results, majority voting can be applied to detect and handle faults.

When mapping an application onto an MPSoC, there are many ways of applying fault-tolerance patterns to the application. This procedure of applying fault-tolerance patterns to an application is called *patternization*. The type of patternization that is applied (like the granularity of the patterns and the number of task replicates used) directly influences the quality of the design in terms of performance, energy and cost. As the effect is hard to predict, it is important to be able to reason about the effects of patternization, right from the start of the design process. Therefore, the patternization should be part of the early design space exploration (DSE) of MPSoC-based embedded systems.

To explore different patternizations, we present the Sesame Automated Fault-tolerance Explorer (SAFE). SAFE provides a novel and generic framework for the early design space exploration of realtime multimedia applications where the effects of implementing fault-tolerance are taken into account. In other words, it aims at making fault-tolerance a first class citizen in early design space exploration. SAFE provides early feedback on the effects of making multimedia MPSoCs fault-tolerant, thereby enabling better decision taking as compared to an approach where fault-tolerance is only addressed in the later design phases. In the latter case, any fault-tolerance measure may invalidate all the decisions taken earlier. Moreover, SAFE provides a simulation-based evaluation, which allows for more detailed architecture models compared to the analytical models often deployed in this domain. As a result, SAFE can produce important metrics like frame drop ratio for dynamically scheduled applications for a wide range of (mixed) fault-tolerance patterns. In this paper, we only focus on active redundancy and soft errors, but SAFE also allows for deploying other types of

CODES+ISSS'12, October 7Ð12, 2012, Tampere, Finland.

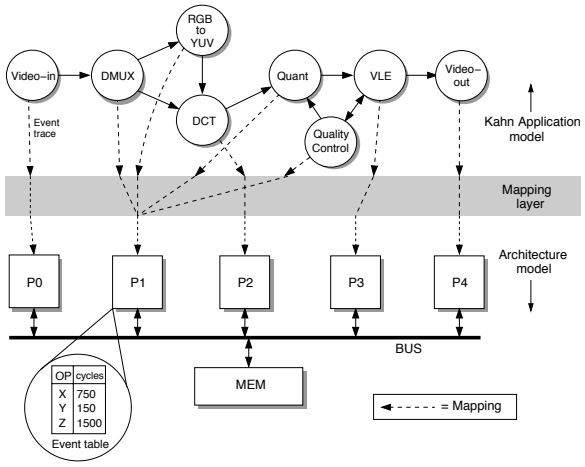


Figure 1: High level MPSoC simulation.

fault-tolerance patterns, like (low overhead) assertion based techniques, as well as for addressing permanent faults.

This paper is organized as follows: the next section describes Sesame, a high level framework on which SAFE is based. Next, the fault-tolerance integration in the MPSoC model is described. The extension of the Sesame simulation model is described in Section 4. Section 5 describes a number of experiments performed with SAFE. Finally, Section 6 describes related work and Section 7 concludes the paper.

2. THE SESAME ENVIRONMENT

For SAFE, we are building upon the Sesame framework for high-level simulation of multimedia MPSoCs [20]. The advantage of high-level modeling is that it allows for a quick pruning of the space of possible MPSoC designs. The Sesame framework, which is illustrated in Figure 1, enables fast performance evaluation using separate application and architectural models with a typical accuracy of 5% compared to a prototype implementation [17]. An application model is built using a Kahn Process Network (KPN) [13], while the architecture model models the MPSoC architecture in a cycle-approximate fashion. Subsequently, there is a mapping of the application model(s) onto the architecture model, implemented using trace-driven co-simulation of the two aforementioned models. Mapping solves two aspects concurrently: 1) *allocation* and 2) *binding*. Allocation selects the architectural components used in the MPSoC platform, whereas the binding defines on which architectural component the application tasks and communications are executed. During the evaluation of a mapping, each process in an application model generates a trace of application events, representing the workload at a high level of abstraction (like read/write a chunk of data, or execute a particular function). These event traces are simulated by the architecture model to obtain metrics like execution time and energy consumption.

Since the applications that need to be mapped on MPSoCs become more and more dynamic, SAFE uses the concept of scenario based design [19]. More precisely, SAFE is based on the scenario-based version of Sesame [24], deploying workload scenarios [8] to model dynamic application workloads. Where Sesame uses absolute metrics for an application workload, like total execution time and energy consumption, SAFE uses the notion of frames. As a result, SAFE considers metrics like frame rate and average power usage. For this purpose, we assume that a single scenario of an application is equivalent to the handling of a single frame. That is, frames are the elementary unit of work performed by an application. For a video application this is the decoding of a single image, whereas the frame of an MP3 application is a single block of sound. Important to notice is that between scenarios application processes do not have any implicit state. As will be explained later on, this simplifies the implementation and modeling of fault-tolerance.

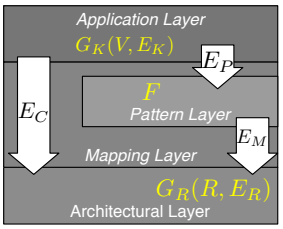


Figure 2: System model with integrated fault-tolerance patterns.

3. FAULT-TOLERANCE INTEGRATION

In this section, a description is given of how fault-tolerance is integrated in SAFE. The first subsection specifies the integration of the fault-tolerance patterns in the system model. Next, the mapping procedure is described. Finally, the fault-tolerance aware mapping process is illustrated using a motion-JPEG (MJPEG) application.

3.1 System model

Figure 2 shows the SAFE system model that integrates fault-tolerance. Apart from the layers that were already mentioned in Section 2 (application, mapping and architecture), the system model also contains a pattern layer. The pattern layer consists of all the possible fault-tolerance patterns that can be used to make an application fault-tolerant. In the following, we will describe the layers in more detail:

Application layer Applications are represented by a directed graph $G_K(V, E_k)$ with $V = V_{\mathrm{N}} \cup V_{\mathrm{OWP}}$ and $V_{\mathrm{N}} \cap V_{\mathrm{OWP}} = \emptyset$. Vertices represent the Kahn Process nodes. To make the external I/O with respect to the complete application explicit, a distinction is made between two type of nodes. Normal process nodes (V_{N}) do not have any interaction with the outside world, whereas *outside world processes* (V_{OWP}) exclusively model the interaction with the world externally to the application. Actions taken by normal process nodes should not affect anything externally to the application. This means that they also cannot write to memory that is accessible by components that are not part of the embedded system. As a result, normal process nodes can be replicated without any side-effects to the application output. Directed edges $E_k = V \times V$ represent communications links to pass messages between process nodes. It is not allowed for OWP processes to communicate directly with eachother:

$$\forall v_1, v_2 \in V_{\mathrm{OWP}} : (v_1, v_2) \notin E_k$$

Architecture layer The architecture is described by the undirected graph $G_R(R, E_R)$. R represents architectural re-

sources like processors, communication buses, splitters, majority voters and buffers. There are some special architectural elements that need to be mentioned specifically. Architectural processors $R_P \subset R$ are the elements capable of executing normal process nodes. OWP processes, on the other hand, are handled by I/O elements $R_I \subset R$. Majority voters $R_V \subset R$ are special architectural resources for implementing active redundancy. They split messages to send them to the different replicas and do a majority voting on outgoing messages. The edges in $E_R = R \times R$ describe the communication links in the architecture.

Pattern layer There are many approaches to make the application fault-tolerant. The collection F represents the possible fault-tolerance patterns. In this paper, we focus on active redundancy [6], but in principle more approaches are possible. A fault-tolerance pattern $f \in F$ contains a description of how the active redundancy is implemented. Examples are double modular redundancy (DMR) and triple modular redundancy (TMR). The fault-tolerance pattern also includes policies like what to do on the detection of a fault and the frequency of checkpoints (see Section 4.3). The function $n_{\text{proc}}(f)$ specifies the number of processors required for the fault-tolerance pattern $f \in F$. For the DMR pattern, this is two processors (possibly the same processor).

Mapping layer Patternization edges E_P describe the manner in which the application is made fault-tolerant. Each edge $(v, f) \in E_P$ represents a possible appliance of the fault-tolerance pattern $f \in F$ for the process node $v \in V_N$.

Mapping edges E_M assign architectural resources to the fault-tolerance pattern. More precisely, the edge $(f, r) \in E_M$ assigns processor $r \in R_P \cup R_V$ to pattern $f \in F$. Next, the I/O edges (E_{IO}) bind the I/O processes. An edge $c = (v, i)$ assigns the OWP processes $v \in V_{\text{OWP}}$ to an I/O component $i \in R_I$ in the architecture.

A final step in the mapping layer is the message dispatch. E.g., there may be multiple process replicas that generate different copies of the same data for a particular destination. Similarly, data may need to be sent to multiple processes Let $M = V_{\text{N}} \times V \times R$ be the set of messages. A message $m = (v_s, v_d, r) \in M$ is data sent from normal process node v_s from resource r such that $(v_s, v_d) \in E_k \vee (v_d, v_s) \in E_k$.

Dispatch edges E_D specify the target of the messages. When $(m, r) \in E_D$, message $m \in M$ needs to be sent to architectural resource $r \in R$.

3.2 Mapping procedure

During DSE, different design instances are created from the system model. To achieve this, a Y-chart approach [14] is used. Figure 3 shows the modification to the Y-chart to integrate fault-tolerance in the DSE. The inputs to our DSE are the applications, the architecture and the fault-tolerance patterns. With these inputs, the mapping procedure can synthesize a design instance. This instance is used in the SAFE simulation framework to obtain performance numbers, after which the DSE process uses these numbers to optimize the application, architecture or mapping.

The mapping procedure maps the application onto the fault-tolerant architecture and is divided into three steps: patternization, binding and message dispatch. Patternization selects the fault-tolerance patterns that will be used in the application. The binding step binds the application

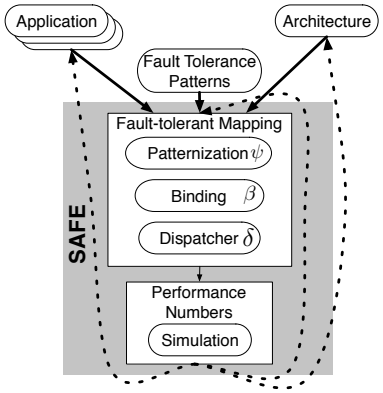


Figure 3: The modified Y-chart for synthesis with fault-tolerance support.

onto the architectural resources. Finally, the dispatch specifies the destination of the messages which are sent between the architectural resources. The complete procedure is as follows:

Patternization The first step in our fault-tolerant mapping is patternization ψ. The patternization ψ contains one edge for each node in V_{N} such that $\psi \subseteq E_P$:

$$\forall v \in V_{\text{N}} : |\{f|(v, f) \in \psi\}| = 1$$

The complete patternization ψ describes the used fault-tolerance patterns in the MPSoC design and the process nodes that are assigned to them. Processes that are connected to a pattern $f \in F$ are called a *fault-tolerant subnetwork* (G_f):

$$G_f := \{v|(v, f) \in \psi\}$$

Computational Binding Computational binding β_x maps the fault-tolerant subnetworks onto the architecture such that $\beta_x \in E_M$. In case a fault-tolerant pattern is used it must be assigned to a voter and a set of processors:

$$\forall f \in F : G_f = \emptyset \vee (|\{r|(f, r) \in \beta_x \wedge r \in R_V\}| = 1 \wedge |\{p|(f, p) \in \beta_x \wedge p \in R_P\}| = n_{\text{proc}}(f))$$

The implication of this constraint is that all the processes in a replica of a subnetwork are mapped onto the same processor. This is useful for minimizing the overhead of explicit checkpoints (as described in Section 4.3.2) as the checkpoint can be taken locally at the processor. Each voter may only be used by one pattern:

$$\forall r \in R_V : |\{f|(f, r) \in \beta_x\}| \leq 1$$

I/O binding I/O binding β_{io} maps the OWP processes onto the architecture such that $\beta_{io} \in E_{IO}$:

$$\forall v \in V_{\text{OWP}} : |\{i|(v, i) \in \beta_{io}\}| = 1$$

Each OWP process must be mapped on exactly one I/O component.

Message Dispatch After binding, a message dispatch δ must be generated. The dispatch δ defines the target of each message such that $\delta \in E_D$. Additionally, the routing between the processors, voters and I/O elements is determined. Currently, this routing is fixed and is not part of the mapping procedure.

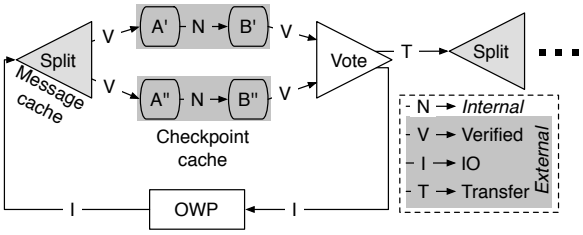


Figure 4: The different types of communication and the required caching for enabling restarting.

In order for the dispatch to be valid, two requirements must be met: 1) the process must be mapped on the resource the message is sent from and 2) each message has at most one destination resource.

$$\forall((v_s, v_d, r), r_d) \in \delta : \exists f \in F \mid ((v_s, f) \in \psi \wedge (f, r) \in \beta_x)$$
$$\forall m \in M : |\{r|(m, r) \in \delta\}| \leq 1$$

The dispatch must also be complete. For each link $e \in E_k$, the message dispatch must specify the destination resource of all the messages:

$$\forall(v_s, v_d) \in E_k : \begin{cases} Intern(v_s, v_d, f) & \exists f \in F : v_s, v_d \in G_f \\ Extern(v_s, v_d, f) & \nexists f \in F : v_s, v_d \in G_f \end{cases}$$

There are two types of communication: internal and external communication. Figure 4 shows the communication types.

Internal communication is communication between two process nodes in the same fault-tolerance subnetwork (in the example between processes A and B). This communication is handled internally by the processor and is unverified (recall that the computational binding enforces that processes on the same subnetwork are mapped onto the same processor). For each of the replicated processes, a message dispatch entry must be present:

$$Intern(v_s, v_d, f) := |\{r|((v_s, v_d, r), r) \in \delta \wedge r \in R_P\}| = n_{\text{proc}}(f)$$

If the processes of the link are not in the same subnetwork, the communication is external. *External communication* passes the majority voter and consists of two steps:

$$Extern(v_s, v_d, f) := \text{V}(v_s, v_d, f) \wedge \begin{cases} IO(v_s, v_d) & v_d \in V_{\text{OWP}} \\ T(v_s, v_d) & v_d \in V_N \end{cases}$$

The first step is *verified communication* (V). This is communication from the replicated process to the majority voter. The majority voter will collect the messages of the different replicas of v_s and will verify the message that is passed on to the destination process v_d:

$$V(v_s, v_d, f) := |\{r|((v_s, v_d, r), r_d) \in \delta \wedge r \in R_P \wedge r_d \in R_V \wedge (f, r_d) \in \beta_x\}| = n_{\text{proc}}(f)$$

The second step of external communication depends on the type of the process node v_d. If the destination process is a normal process node ($v_d \in V_{\text{N}}$), there will be a *transfer* (T) of the message to the subnetwork of v_d. In the case of an OWP process ($v_d \in V_{\text{OWP}}$), IO is performed:

$$IO(v_s, v_d) := |\{r|((v_s, v_d, r), r_d) \in \delta \wedge r \in R_V \wedge (v_d, r_d) \in \beta_{io} \wedge r_d \in R_I\}| = 1$$
$$T(v_s, v_d) := |\{r|((v_s, v_d, r), r_d) \in \delta \wedge r, r_d \in R_V \wedge (v_d, f_d) \in \psi \wedge (f_d, r_d) \in \beta_x\}| = 1$$

This second step only needs to be done once. For a transfer, the destination resource must be the voter on which the subnetwork of v_d is mapped. The message will be split by a splitter (which is physically part of a voter component, see Figure 5b) for use by the replicated processes of v_d. In case of I/O, the destination resource must be the I/O element on which process v_d is mapped.

Besides writing data, read requests are issued by process nodes. These read requests are passed on to a voter element or an I/O element. The completeness requirements are similar to those of writing data, except that the reading node is the sending process and the writing node is the receiving process:

$$\forall(v_s, v_d) \in E_k : \begin{cases} Intern(v_d, v_s, f) & \exists f \in F : v_s, v_d \in G_f \\ Extern(v_d, v_s, f) & \nexists f \in F : v_s, v_d \in G_f \end{cases}$$

A design instance is feasible if and only if all the constraints during the patternization, computational binding, I/O binding and message dispatch are fulfilled.

3.3 An MJPEG Example

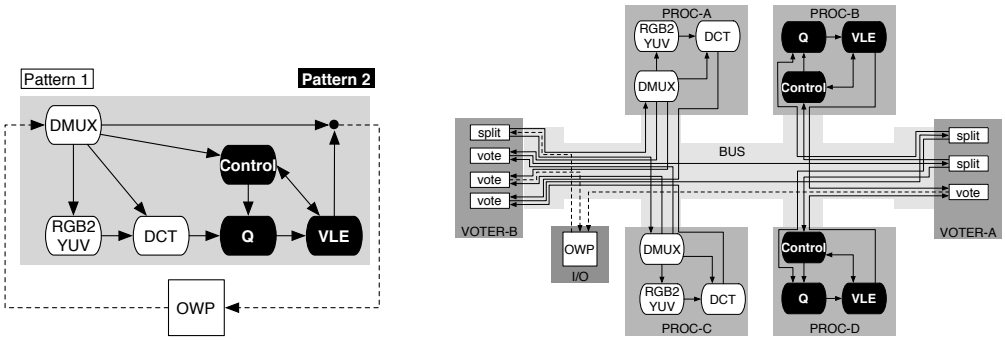
To illustrate the fault-tolerance aware mapping in SAFE, an example mapping of an MJPEG encoder is given in Figure 5. First, the application model is illustrated in Figure 5a. All the processes are normal (V_{N}), apart from the OWP process. As discussed in the previous section, the OWP is part of V_{OWP} to make those actions explicit that can have side-effects external to the specific application. As the OWP is aware of the arrival times of frames, as well as their output times, we can determine if a certain frame rate is met.

DMR-A and DMR-B are the fault-tolerance patterns that are used in our example application. DMR is a type of *active redundancy* [6] that uses two processors (not necessarily different). The active redundancy subnetwork is duplicated in time or space ($n_{\text{proc}}(\text{DMR}) = 2$) among the processor(s) to be able to compare the different outgoing messages. An architectural majority voter (VOTER-A or VOTER-B) takes care of the splitting of incoming messages to the replicas and the verification of outgoing messages using majority voting. The complete description of the mapping procedure is as follows:

The first step of the mapping is the patternization. Most fault-tolerant exploration frameworks are fixated on a single fault-tolerance pattern: replicate the complete application or replicate each single process. The SAFE model is, however, not limited to a fixed pattern. In our example (Figure 5a), the MJPEG encoder is split into two fault-tolerant subnetworks: $G_{f,\text{DMR-B}} = \{\text{DMUX}, \text{RGB2YUV}, \text{DCT}\}$ and $G_{f,\text{DMR-A}} = \{\text{Control}, \text{Q}, \text{VLE}\}$.

$$\psi = \{(\text{DMUX}, \text{DMR-B}), (\text{Control}, \text{DMR-A}), (\text{RGB2YUV}, \text{DMR-B}), (\text{DCT}, \text{DMR-B}), (\text{Q}, \text{DMR-A}), (\text{VLE}, \text{DMR-A})\}$$

A next step is to perform the binding (see Figure 5b). During the communication binding, the OWP process is



(a) Patternized Application (b) Synthesized design with two DMRs

Figure 5: MJPEG application synthesized with two active redundancy networks.

bound onto the I/O module. As the DMR pattern of subnetwork $G_{f,\text{DMR-A}}$ requires two processors, subnetwork $G_{f,\text{DMR-A}}$ is bound to processors PROC-B and PROC-D. Similarly, subnetwork $G_{f,\text{DMR-B}}$ is bound to PROC-A and PROC-C. In this example, all the replicas are mapped to different processors. The SAFE model does not require this and all the replicas could e.g. also have been mapped to PROC-A.

$$\beta_{io} = \{(OWP, I/O)\}$$
$$\beta_x = \{(\text{DMR-A}, \text{PROC-B}), (\text{DMR-A}, \text{PROC-D}), (\text{DMR-A}, \text{VOTER-A}), (\text{DMR-B}, \text{PROC-A}) (\text{DMR-B}, \text{PROC-C}), (\text{DMR-B}, \text{VOTER-B})\}$$

The final step is generating the dispatch of the messages. In this example, we will discuss messages from four links: (VLE, OWP), (Q, VLE), (DCT, Q) and (OWP, DMUX). For each link, multiple messages need to be dispatched: one message per replicated process and possibly a verified message after majority voting. For the replicated processes, the communication is internal or verified. (Q, VLE) is internal because Q and VLE are on the same subnetwork $G_{\text{DMR-A}}$. That is why Q on PROC-B can directly write a message to VLE on PROC-B without any dependency on the Q process that is running on PROC-D.

For (VLE, OWP), (DCT, Q) and (OWP, DMUX) the communications (read/write) are external. For the DMR pattern, this means that for written data the messages of all the replicas must be compared before a message is passed on. Take for example (DCT, Q). Both the DCT processes on PROC-A and PROC-C must write a message after which they are processed (i.e., voting) and a single message is passed on. As the destination of the message is located in another subnetwork ($G_{f,\text{DMR-A}}$), the resulting messages are transferred to VOTER-B. Whenever one of the Q processes does a read request on (DCT, Q), it receives a copy of the message from VOTER-B. The OWP process is passive. Incoming frames for the MJPEG application, i.e. (OWP, DMUX), are handled by sending the data to the voter belonging to the subnetwork of the target process (VOTER-B for (OWP, DMUX)). Whenever a voter has a message for an OWP process (outgoing I/O) it will be sent to the I/O element.

4. SIMULATION MODEL

After a design instance is generated using the mapping procedure, a SAFE simulation model is used to obtain its performance numbers. This section describes the different aspects of this fault-tolerant aware model. The first subsection discusses the fault injection, followed by a description of the fault detection. To facilitate fault correction, restarting and checkpointing is introduced in the simulation model. This is elaborated in the third subsection. Finally, the last subsection describes the obtained performance metrics.

4.1 Fault Injection

To be able to reason about the trade-off between objectives like reliability and performance, faults must be simulated in our model. First, we make the assumption that the communication network is fault-tolerant. Therefore, the only source of error is at the processor level. Potentially, all the different processor components can be affected by an error [3]. Examples are the register file, the logical units and the on-chip memories. As SAFE models a processor at a high level of abstraction, we only know what function an application is executing and when it reads or writes data. Therefore, we use the SoftWare Initiated Fault Injection (SWIFI) method [22]. In principle, both permanent and transient errors can be modeled. In this paper, however, we limit ourselves to transient errors.

The processor models the occurrence of faults using an exponential random distribution. An exponential distribution describes the time between events in a Poisson process, which occur continuously and independently at a constant average rate. This is very suitable for modeling fault injection times [7] as transient errors are infrequent and independent of earlier errors. During the simulation, the processor model iteratively injects transient faults based on the exponential distribution. Here, it depends if the transient error affects the execution of the application. The fault does not have any consequences when no process is active at the time the fault occurs. However, in case a process is active, it will invalidate all future output of the process. In reality, it may be the case that the fault does not affect the output, but in our high-level approach we cannot know the exact effect. Therefore, we take the most pessimistic assumption.

4.2 Fault Detection

A fault injected into an application will be propagated until it is detected by one of the fault-tolerant subnetworks. When a fault-tolerant subnetwork detects an error, there are two possible responses: fault masking and fault correc-

tion. *Fault masking* is applied when there is a clear majority for the data values among the different replicates. The execution can be continued by passing on the data value of the majority. The corrupt replicate is allowed to continue its execution. As there is no implicit state between frames in our KPN applications (see Section 2), the next frame is processed with clean data again.

If there is no majority, then only *fault detection* can be performed. On the detection of a non-maskable fault, there are two options: *skipping* and *restarting*. In case of skipping, the current frame is dropped and the processing of the next frame is started.

4.3 Fault Correction

If the number of skipped frames becomes too large, then the quality of service (QoS) of the application significantly deteriorates. In this case, restarting the frame processing can improve the QoS. To facilitate such restarting, *checkpointing* must be part of the SAFE model.

4.3.1 Checkpoint budget

As illustrated in Figure 6, there are two types of checkpoints: implicit and explicit. *Implicit checkpoints* are located at the frame barriers. In the example of Figure 6 checkpoints $C_{A.3.0}$ and $C_{B.3.0}$ form the implicit checkpoint at frame 3. Notice that an implicit checkpoint is not taken at once. Process A reaches the barrier of frame 3 much earlier than process B. The complete implicit checkpoint is available once every process in the subnetwork has reached the specific frame barrier. A complete implicit checkpoint does not require storage (as there is no implicit state between frame barriers), but it allows us to perform message cleanup as will be discussed in Section 4.3.2.

As will also be detailed in the next subsection, the restart from an implicit checkpoint is trivial because no state needs to be restored for the processes. Still, restarting from an implicit checkpoint has some disadvantages. Not only needs the complete frame to be recalculated, but it can also be the case that the application is not started at its full capacity. Take the simple application in Figure 6. On a restart from an implicit checkpoint at the end of frame 2, both processes start at the barrier of frame 2. In this case process B needs to wait for output of process A before it can do any work.

To resolve this, explicit checkpoints can be taken during the lifetime of an application. *Explicit checkpoints* are initiated by the voter and store the state of all the processes in the active redundancy network and their internal communication channels at a specific point in time. In contrast

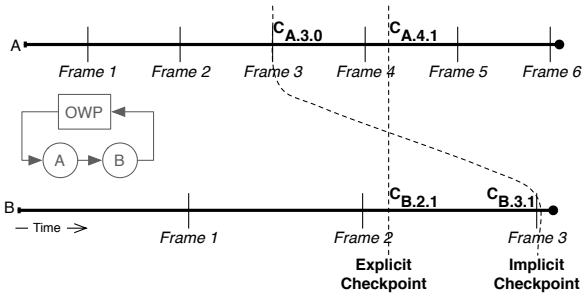


Figure 6: Implicit versus explicit checkpoints as illustrated by a timeline of a simple application.

to implicit checkpoints, the complete subnetwork is halted to capture the state of the current processes and the internal communication channels. To obtain a consistent explicit checkpoint, the voter will ensure that all the replicas stop at the same point in the application. One of the processes in the subnetwork will be responsible for collecting and sending the checkpoint. This results in a checkpoint for each replica, which will be compared by the voter. In Figure 6, $C_{A.4.1}$ and $C_{B.2.1}$ are illustrations of an explicit checkpoint.

Implicit and explicit checkpoints can also enhance each other. Take for example the checkpoints in Figure 6. If there is a restart just after the checkpoint of $C_{B.3.1}$, the restart can take place from a combination of the explicit checkpoint $C_{*.*.1}$ and the implicit checkpoint $C_{*.3.*}$. This means that process A starts from $C_{A.4.1}$ and process B starts from $C_{B.3.1}$. Without the presence of explicit checkpoints, process A should have processed frame 3 again. Similarly, with only explicit checkpoints, process B would have been required to process frame 2 again.

For each fault-tolerant subnetwork, a *checkpoint budget* is defined. The checkpoint budget determines the checkpointing granularity by describing the number of explicit checkpoints per frame (possibly 0). The size, and thus the overhead, of the checkpoint is dependent on the application processes and the amount of data in the internal communication channels. This also means that the voting time of a checkpoint is variable. That is, the explicit checkpoints of the different replicas in a subnetwork must be verified against each other to ensure that on a restart the process state is valid. After verification by the voter, the explicit checkpoint is stored locally at the processor.

Explicit checkpoints are not only useful to minimize the amount of work that has to be redone, but they also allow to implement fault-tolerant techniques like Roll Forwarding Checkpointing Schemes [21] or assertion based techniques [11] where a fault is corrected by reprocessing the frame. This is, however, beyond the scope of the current paper.

4.3.2 Restart budget

Fault correction is not simply the case of enabling or disabling the possibility to restart the processing of a corrupted frame. As restarting requires some overhead, it cannot be done unlimitedly. Therefore, a restarting budget is specified. The restarting budget defines the number of times a restart may be done during the processing of a frame. In case the restart budget is zero, no restarting is performed at all and the fault-tolerant subnetwork will skip corrupted frames.

When restarting is enabled, some data must be cached (see Figure 4). The explicit checkpoints must be stored locally at each of the processors on which the fault-tolerant subnetwork is mapped. Moreover, messages need to be cached at the splitter. Without caching the incoming messages of the subnetwork, it cannot be guaranteed that input data is still available. On top of that, it must be possible to restart subnetworks without the need of restarting other networks to regenerate the incoming data. The caches can be protected from errors in several ways, but as SAFE is currently only targeted towards transient errors a memory with ECC should be sufficient.

The splitter model incorporated in SAFE models a limited amount of cache capacity. Therefore, a policy is required for *message cleanup*. There are two moments for cleaning up messages: during explicit checkpoints and during implicit

checkpoints. During explicit checkpoints all the messages in the cache may be discarded. At implicit checkpoints, all the input messages of earlier frames may be removed.

If a corrupted frame is encountered then there are two possibilities. When the restart budget is empty, the current frame is dropped. Otherwise, the restart budget is decreased by one and the subnetwork is restarted from the most recent checkpoint. Two things need to be done during a restart: 1) extract the process states from the checkpoint (not necessary in the case of an implicit checkpoint) and 2) restoring the externally incoming messages at the splitter.

To conclude, checkpointing involves overhead (in space and time), but it tries to avoid skipping frames on faulty processors. The usage of checkpoints depends on the application and the failure rate of the processors. The higher the failure rate of processors, the higher the fraction of skipped frames will be. Additionally, the type of application determines the fraction of skipped frames that can be tolerated. For a safety critical application, none of the frames may be skipped. Multimedia applications typically allow for a small fraction of the frames to be skipped, as long as it is unnoticeable to the user. All these trade-offs can be explored automatically by SAFE.

4.4 Performance metrics

All the techniques described in the previous subsections have been implemented in an extension of the Sesame environment [20]. As the external I/O is made explicit by the use of OWP processes, SAFE allows for obtaining metrics like frame rate in the presence of transient faults. Moreover, since the modeled architectural elements include fault-tolerance components such as majority voters, the overhead of applying active redundancy (or any other fault-tolerance implementation technique) is part of the analysis. Besides the frame rate, other metrics can be obtained as well, such as frame miss ratio, the number of unrecoverable faults and the number of frames that are skipped because the incoming I/O is not available anymore.

We should note, however, that our simulation-based approach is not able to provide guarantees for the realtime behavior. Its goal is not to make the system reliable, but to identify the most reliable design in the design space taking all the other system objectives into account. SAFE is thus able to prune the design space such that only a limited number of designs need to be studied in more detail during later design phases.

5. EXPERIMENTS

In this section, a use case for fault-tolerant DSE with SAFE will be presented. For this purpose, three different applications are used: an MJPEG encoder, a sobel edge detector (to detect edges for each image in a video stream) and an MP3 decoder. All of these applications are individually mapped onto an MPSoC, which somewhat resembles the architecture in Figure 5b. The only difference is that for our experiments, 6 voters are available. To reduce the probability of a communication bottleneck, there are two buses to which all the processors are connected. The voters, however, are distributed: 3 voters are connected per bus.

All the processors have the same failure rate: 10^{-6} FIT. This is a rather conservative choice based on logic implemented in a 180nm technology [23]. Our focus will be on exploring the trade-offs of implementing fault-tolerance. Therefore, the computational binding will be fixed: processors A, B and C will be used to for the first, second and third instance of each replicated process.

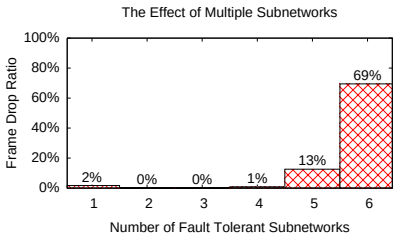


Figure 8: Different subnetworks and their effect on frame drop ratio of the MJPEG application.

We have performed two experiments. In the first experiment, the MJPEG application is revisited. Here, we will scrutinize the process of patternization, and in particular, the identification of several optimal patternizations. Subsequently, SAFE is used for a fault-tolerant DSE on the three test applications individually.

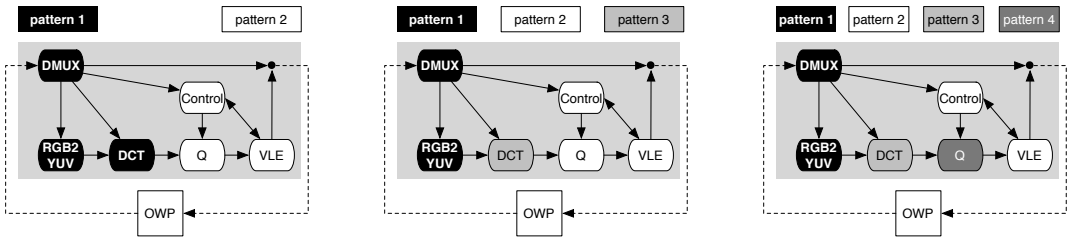
5.1 MJPEG Patternization

To study the process of patternization, we performed an exhaustive patternization for the MJPEG application and an MPSoC where only triple modular redundancy (TMR) patterns are used. There are multiple instances of the TMR in the set of fault-tolerance patterns, involving different choices for the restart budget and the checkpoint budget.

Figure 7 shows the optimal patternization for a different number of fault-tolerant subnetworks. In this case, we have taken frame drop ratio as a primary objective and power consumption as a secondary objective. A first observation is that in this design space the patternization is incremental. By adding an additional subnetwork, one of the processes is moved into the new subnetwork. In the case of two subnetworks, the application is split into two equally sized subnetworks. Not only are these subnetworks equally sized, but also the number of external communication channels is kept minimal. Apart from the I/O communication channels (which are external by definition), only the channels (DCT, Q) and (DMUX, Control) are external. Due to this minimum of external channels, the amount of majority voting (only done on external communication) is minimized.

When increasing the number of subnetworks to three, the DCT process is put into a separate subnetwork. The rationale is not the minimization of external communication, but the guarding of the compute intensive tasks. As the DCT is the most computationally expensive operation, it is beneficial to ensure that verified data is used in the computation. If it would have been unverified, it can be the case that unnecessary computation will be done. The same is true for the optimal patternization with four subnetworks. In this case, the quantization (Q) process is separated, being the second most compute intensive operation in MJPEG.

Having more fault-tolerant subnetworks may increase the quality of the application (with respect to frame drop ratio). However, it also increases overhead. This can be seen in Figure 8. Up to four subnetworks, the frame drop ratio reduces to 0 percent. With five or more subnetworks, the frame drop ratio quickly climbs up to 69 percent when each process is placed in a separate subnetwork.



(a) 2 subnetworks (b) 3 subnetworks (c) 4 subnetworks

Figure 7: Optimal patterns for a different number of fault-tolerance subnetworks.

5.2 Fault-tolerant DSE

To further illustrate the potentials of SAFE, three test applications are mapped using a fault-tolerant DSE: MJPEG, sobel and MP3. To keep the experiment feasible, we performed a partial search of the design space. To achieve this, we explored all the mappings with at most six fault-tolerant subnetworks. Each of the subnetworks is implemented using the same type of fault-tolerance pattern. The explored fault-tolerance types are DMR and TMR, and for the restart budget and the checkpoint budget (only relevant when the restart budget is nonzero) the complete range between 0 and 6 is covered. Our model is only able to make explicit checkpoints between communication events. Because the MP3 application is too coarse grained as it has at most two communication events per frame, the fault tolerance patterns with an explicit checkpoint budget larger than 0 are not explored for the MP3 application.

5.2.1 Power and Frame Drop Ratio

Figure 9a shows the frame drop ratio and power consumption trade-off for all of the design points. The design points are differentiated based on the type of fault-tolerant pattern: 1) DMR or TMR and 2) restarting enabled or not. A first and straightforward observation is that for our experiment DMR takes less power than TMR. The overhead of executing an additional replica, is clearly larger than the restarting overhead, since DMR with restarting takes less power than TMR without restarting.

Therefore, for this experiment, it depends on the application if it makes sense to use TMR. Figure 9a also shows the Pareto front of non-dominated solutions. For MJPEG and MP3, DMR dominates all the design instances with TMR. The reason is that DMR is already capable of achieving a frame drop ratio close to or equal to zero. However, for the sobel application, DMR cannot obtain a frame drop ratio below 5 percent. With TMR, sobel can achieve a frame drop ratio of 1.4 percent. Similarly, it depends on the application if restarting is required. The MP3 application is quite lightweight and can already achieve a frame drop ratio of 0.1 percent without restarting. Heavier applications like sobel and MJPEG cannot go below a frame drop ratio of 16 and 31 percent respectively without restarting.

More unexpected are the two clusters of design instances for the MJPEG application. For each experiment instance (DMR, TMR, with and without restarting), there are roughly two clusters: a large drop ratio and a low one. Further investigation showed that once the frame drop ratio is above a certain threshold ($\pm 50\%$) the MJPEG application cannot keep up pace with the incoming frames from the OWP process. As a result, many frames are skipped (e.g., not processed at all) due to deadline misses and the frame drop ratio is drastically increased. The higher number of unprocessed input images has a side effect: less power is required.

Also interesting to notice is the shape of the front of the MP3 application. Increasing power consumption of the system (e.g., making it more fault-tolerant), makes the frame drop ratio even higher. This is caused by the fault tolerance overhead as will be discussed in the next subsection.

5.2.2 Breakdown of Frame Drop Ratio

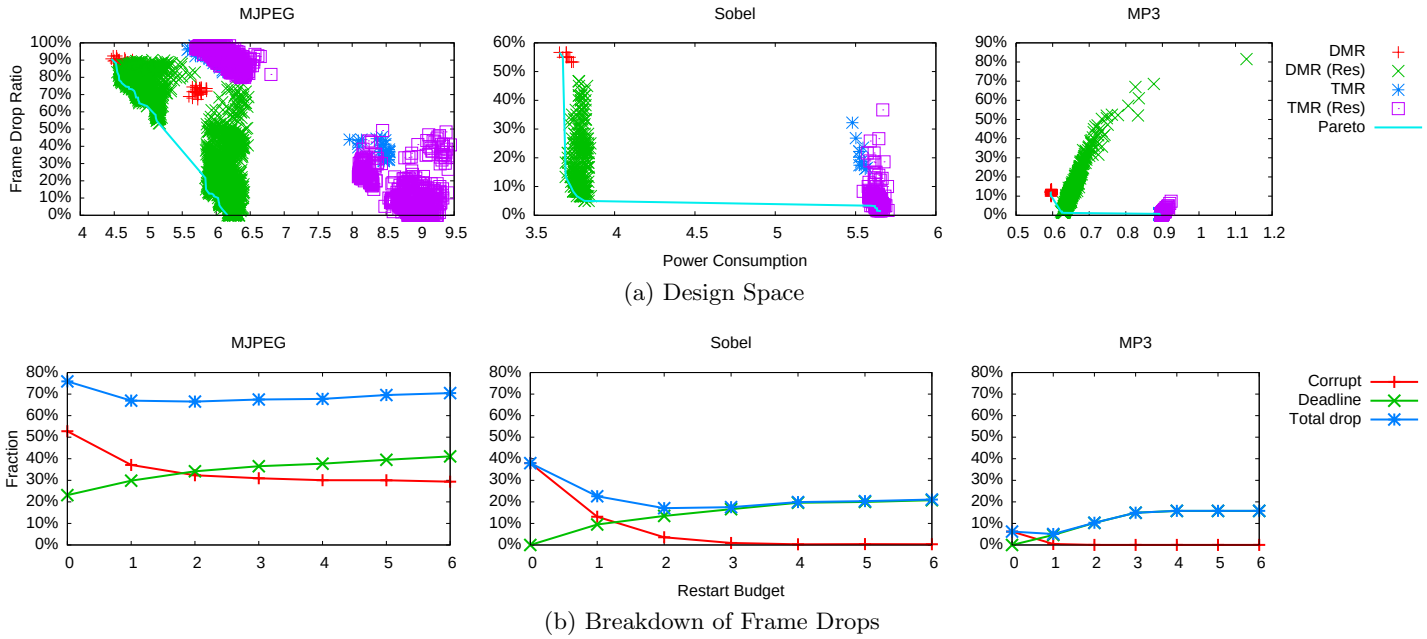
Dropping a frame can have several causes, which is illustrated in Figure 9b. First, there are corrupted frames. In these frames, a fault is detected (due to a transient error), but the restart budget was too small to correct these faults. Second, there are deadline misses. In these cases, the application is too late to retrieve or deliver a frame from/to the OWP process. For the experiment in Figure 9b, we have taken all the design points of the previous experiments where the explicit checkpoint budget was 0. These design points are differentiated by restart budget and for each category the average frame drop ratio is given. Not only the total drop ratio is given, but also the fractions that are due to corrupt frames and deadline misses.

A larger restart budget can both improve the frame drop ratio (less corrupted frames) and degrade the frame drop ratio (more deadline misses). The more effort is put in fault correction the lower the number of corrupted frames. However, the effort has a negative effect on the number of deadline misses. The optimal restart budget is thus application dependent. For all our applications, the gain in the reduction of corrupted frames is overshadowed by the increase of deadline misses at about 1 or 2 restarts per frame. However, the MP3 application is not influenced anymore once the restart budget is above 3. In this case, the MP3 application is already able to circumvent corrupted frames and the additional restarts will not be used.

5.2.3 Buffer Requirements

We have already shown that restarting and checkpointing have a positive effect on the frame drop ratio. A next question is what the effect of the additional checkpointing is on the buffer requirements. For this purpose, we studied the MJPEG and the sobel applications and selected the best design points for a checkpoint budget between 0 and 6. The results are illustrated in Figure 10. The left vertical axis represents the normalized buffer size, whereas the right vertical axis shows the frame drop ratio.

The checkpoint budget has a positive effect on the buffer



(a) Design Space

(b) Breakdown of Frame Drops

Figure 9: The results of the fault-tolerant DSE of the 3 test applications: MJPEG, sobel and MP3.

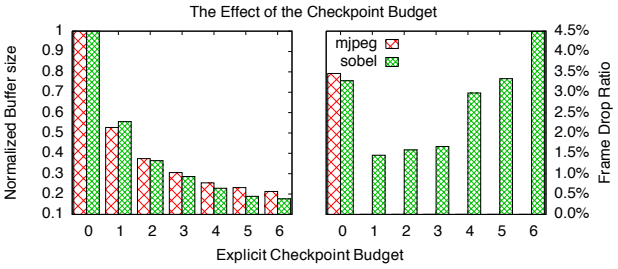


Figure 10: Varying the checkpoint budget of an MJPEG application with restarting enabled.

requirements. When the checkpoint budget increases, the required size of the message cache decreases. The reason is that the message cache can be flushed whenever an explicit checkpoint is taken (see Section 4.3.2). The more often a checkpoint is taken, the fewer messages are simultaneously cached. The storage requirements of the explicit checkpoints, however, are less influenced by the frequency of checkpointing (not shown in the graph).

Still, a checkpoint has its overhead. It depends on the application if it can be tolerated or not. The sobel application cannot tolerate the overhead and with a checkpoint budget above 1, the frame drop ratio is increasing due a larger number of deadline misses. The MJPEG application, on the other hand, can take up to 6 checkpoints per frame without any deadline misses.

6. RELATED WORK

A lot of research effort has been spent on the early DSE of embedded systems [10]. Recently, workload scenarios [8] have been introduced to make the embedded system design scenario aware [19, 18, 24]. Our work is based on [24], and extends this into the direction of fault-tolerance aware DSE. This is because reliability becomes a major design objective [23, 16] and has a major effect on other objectives. It is therefore important that this objective is explored as a separate design objective [9] and not hierarchically.

Many of the reliability aware DSE environments [2, 5, 12, 11, 6] are based on a static scheduling of application tasks. COFTA [5] aims at minimizing the overhead of fault-tolerance measures using replication and assertion. The analysis only takes the performance and failure rate of the final application into account. It provides exploration of the placement of the assertion and replication patterns. COFTA misses the ability to reason about handling multiple frames and their throughput. The methods in [12, 11] provide fault-tolerance in realtime embedded systems using re-execution of tasks. For this purpose, a sanity check determines if the task is executed correctly. In case of a fault, a special time slot can be used to re-execute a task. As all of these approaches use static scheduling, they are based on worst case execution time. The method in [2] tries to resolve this by using a hyperperiod instead of a fixed time. Still, it does not allow for modeling the full dynamism within applications.

Bolchini [4] provides an analysis framework that performs reliability analysis for an embedded system with dynamic scheduling. The framework injects faults into the system during execution and classifies the effect on the system (like silent, detected and failure).

The novelty of SAFE is that it not only provides exploration on the placement of fault-tolerance patterns, but it is also capable to fully simulate realtime embedded systems with dynamic scheduling at a high abstraction level. On top of that, it provides a unique fine-grained and parametrized checkpointing model. Most fault-tolerant aware DSE frameworks only take implicit checkpoints into account [15] that are strictly taken in between frames. As a result, SAFE provides more detailed insight into the consequences of fault-tolerant design. By using the patternization approach, different fault-tolerance patterns can be applied to different

parts of the same application. Finally, the explicit modeling of transient errors allows us to not only model the effects of the (mixed) fault-tolerance patterns on the system performance, but also to gain insight into the number of missed frames (due to a late arrival or an unrecoverable fault).

7. CONCLUSION

In this paper, we have argued that fault-tolerance should become a first class citizen in the early design space exploration of MPSoCs. To this end, we have presented SAFE, a simulation environment to explore the different implementation strategies for realizing fault-tolerance in MPSoC-based embedded multimedia systems.

Our experiments show that it is beneficial to explore all the possible ways of implementing fault-tolerance. A custom patternization allows to leverage fault handling capabilities and reducing performance overhead by minimizing the amount of verified communication and to assure that the input of compute intensive tasks is correct. As a consequence, an automated way of exploring all these possible fault-tolerant implementation variants is crucial to make the correct decisions during the early design stages.

Future work will focus on large-scale DSE in which mixed techniques (not only active redundancy, but also assertion-based techniques) are explored for fault-tolerant MPSoC-based embedded systems.

8. REFERENCES

[1] M. Agostinelli et al. Random charge effects for PMOS NBTI in ultra-small gate area devices. In *IEEE Int. Symp. on Reliability Physics*, pages 529–532, 2005.

[2] P. Axer, M. Sebastian, and R. Ernst. Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints. In *Proc. of the 9th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 149–158, October 2011.

[3] C. Bolchini et al. A model of soft error effects in generic IP processors. In *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE Int. Symposium on*, pages 334 – 342, October 2005.

[4] C. Bolchini and A. Miele. An application-level dependability analysis framework for embedded systems. In *IEEE Int. Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 171–178, October 2011.

[5] B. Dave and N. Jha. COFTA: hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance. *IEEE Trans. on Computers*, 48(4):417–441, April 1999.

[6] O. Derin, E. Diken, and L. Fiorin. A middleware approach to achieving fault tolerance of kahn process networks on networks on chips. *International Journal of Reconfigurable Computing*, 2011:15, 2011.

[7] F. Faure et al. Single-event-upset-like fault injection: a comprehensive framework. In *IEEE Transactions on Nuclear Science*, volume 52, pages 2205–2209, December 2005.

[8] S. V. Gheorghita et al. System-scenario-based design of dynamic embedded systems. *ACM Trans. on Design Automation of Electronic Systems*, 14(1):1–45, 2009.

[9] M. Glass et al. Reliability-aware system synthesis. In *Design, Automation Test in Europe Conference Exhibition 2007 (DATE '07)*, April 2007.

[10] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.

[11] J. Huang et al. Reliability-aware design optimization for multiprocessor embedded systems. In *14th Euromicro Conf. on Digital System Design (DSD)*, pages 239–246, September 2011.

[12] V. Izosimov et al. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *Proc. of the conf. on Design, Automation and Test in Europe (DATE)*, pages 864–869, 2005.

[13] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[14] B. Kienhuis et al. A methodology to design programmable embedded systems: the Y-chart approach. *LNCS - Embedded Processor Design Challenges*, 2268:18–37, 2002.

[15] K. Lee et al. Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach. In *Proc. of the 16th ACM int. conf. on Multimedia*, pages 319–328, 2008.

[16] S. Mitra et al. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, February 2005.

[17] H. Nikolov et al. Daedalus: toward composable multimedia MP-SoC design. In *Proc. of the Design Automation Conf. (DAC)*, pages 574–579, 2008.

[18] G. Palermo, C. Silvano, and V. Zaccaria. Robust optimization of SoC architectures: A multi-scenario approach. In *Proceedings of ESTIMedia 2008 - IEEE Workshop on Embedded Systems for Real-Time Multimedia. Atlanta, Georgia, USA*, October 2008.

[19] J. M. Paul, D. E. Thomas, and A. Bobrek. Scenario-oriented design for single-chip heterogeneous multiprocessors. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(8):868–880, August 2006.

[20] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. on Computers*, 55(2):99–112, 2006.

[21] D. Pradhan and N. Vaidya. Roll-forward checkpointing scheme: a novel fault-tolerant architecture. *IEEE Trans. on Computers*, 43(10):1163–1174, October 1994.

[22] A. Rohani and H. Kerkhoff. A technique for accelerating injection of transient faults in complex SoCs. In *14th Euromicro Conf. on Digital System Design (DSD)*, pages 213–220, September 2011.

[23] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. of the Int. Conference on Dependable Systems and Networks (DSN'02)*, pages 389 – 398, 2002.

[24] P. van Stralen and A. D. Pimentel. A trace-based scenario database for high-level simulation of multimedia MP-SoCs. In *Proc. of the Int. Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS '10)*, Samos, Greece, July 2010.