Contract no. 248424

FP7 STREP Project

MADNESS

Methods for predictAble Design of heterogeNeous Embedded System with adaptivity and reliability Support

D6.2: First report on DSE for adaptive MPSoC

Due Date of Deliverable	1st January, 2012
Completion Date of Deliverable	1st January, 2012
Start Date of Project	1st January, 2010 – Duration 36 Months
Lead partner for Deliverable	UvA

Revision: v1

Project co-funded by the European Commission within the 7th Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other program participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Table of Contents

1 Introduction

This document reports on our initial efforts towards the design space exploration (DSE) of adaptive MPSoC systems. As a driver for adaptivity, we have chosen to address fault tolerance, as this perfectly aligns with the other system reliability research going in within the MADNESS project. The report will describe the steps we have taken to make our Sesame modeling and simulation environment as well as the DSE process using Sesame fault-tolerance aware. These steps are prerequisite for modeling, simulating and performing DSE of adaptive, fault-tolerant MPSoCs.

MPSoC design deals with many objectives. One of them is reliability. A MPSoC needs to be able to cope with soft and hard errors. Soft errors are transient errors that cause a temporal malfunction in the system. These soft errors are often called single upset event. This is due to one cause of soft errors: high energy neutrons resulting from cosmic rays colliding with particles in the atmosphere. More generally, 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 become more susceptible to soft errors [8]. Therefore, the MPSoC needs to be able to cope with these errors.

One software-based technique to deal with soft errors is active redundancy in space or time. If active redundancy is used in the space domain different resources are used to run the same tasks. Another possibility is to run the task multiple times on the same resource (space domain). A combination of both is also possible. 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 do fault detection and possible fault masking.

When mapping an application onto a MPSoC, there are already many ways resulting in results of different quality (for example with respect to performance or power). Taking the reliability into account, the number of possible MPSoC designs becomes even larger. There are many ways of applying active redundancy to an application. This active redundancy also affects the quality of the system (it becomes slower, consumes more energy). For that reason, we have extended the DSE to include the reliability of the system.

Moving further, we can also exploit the fault-tolerance by adjusting the MPSoC to the dynamic circumstances. Whenever the quality of the output drops, the MPSoC can change the mapping at runtime. In this way, also hard failures can be handled.

The contribution of this research is as follows: 1) Realtime semantics, 2) Checkpoint modeling, 3) High-level Active Redundancy modeling, 4) KPN Segregation to trade-off checkpointing overhead, and 5) Reliability-aware DSE in Sesame. In the rest of the report, we describe the fault-tolerant extension in more detail. The second section describes the Sesame environment. Next, the third section describes the real-time extension of Sesame. The fourth and fifth sections describe the fault-tolerance and the application segregation.

2 Sesame

In this work, we are using the high-level MPSoC simulation framework Sesame [6]. The advantage of high-level modeling is that it allows for a quick exploration 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. An application model describes an

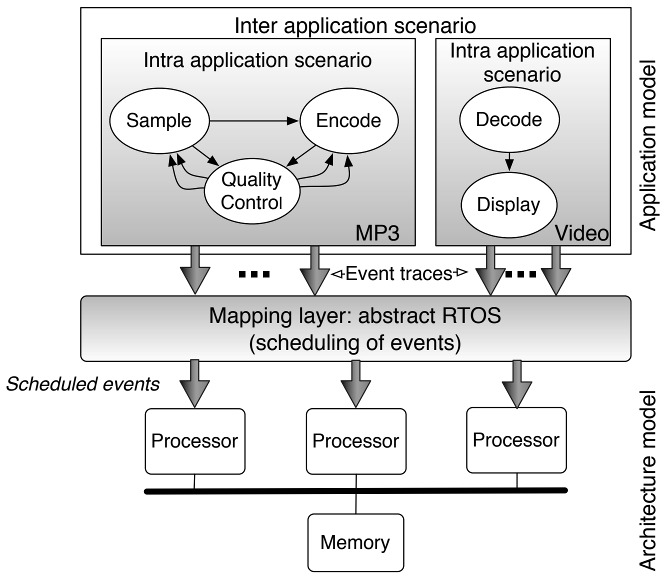
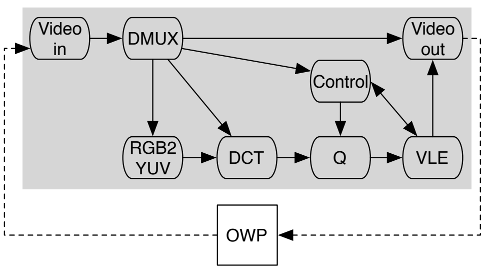


Figure 1: High level scenario-based MPSoC simulation

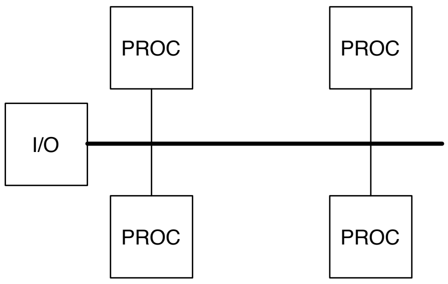
application using a Kahn Process Network (KPN) [3], while the architecture model models the MPSoC architecture in a cycle-approximate fashion. Subsequently, there is an explicit mapping of the application model(s) onto the architecture model, implemented using trace-driven co-simulation of the two aforementioned models. Mapping solves two aspects concurrently: 1) allocation and 2) binding. Allocation selects the architectural components used on 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 application workload at a high level of abstraction. These event traces are simulated by the architecture model to obtain non-functional metrics like execution time, energy consumption and cost.

The applications that need to be mapped on the MPSoC become more and more dynamic. Therefore, we are using scenario based design [5]. More precisely, we are using the scenario-based Sesame [9] that deploys workload scenarios [2] to model the dynamic applications. The scenario-based Sesame distinguishes two types of workload scenarios: intra and inter application scenarios. Intra-application scenarios describe the different behaviors, or operation modes, within an application. For example, an MP3 application can play music in mono or stereo sound. An inter-application scenario describes the behavior of multiple applications. In our case, it specifies which applications can run concurrently. In the example of Figure 1, the inter-application scenario describes simultaneous execution of the MP3 application and the video application. In order to fully describe what a system is doing, a complete application scenario bundles the possible intra-application scenarios of all the active applications. The set of active applications is described using an inter-application scenario, whereas each intra-application scenario specifies a particular operation mode of an individual application. An example application scenario in Figure 1 is that the MP3 application is playing music in mono sound, while the video is decoded at a low bitrate.

Important to notice is that between scenarios processes do not have any state. This makes the modeling of fault-tolerance and migration easier.



(a) Application layer



(b) Architecture layer

Figure 2: An example MPSoC model extended with I/O support

3 Realtime KPN applications

When performing fault-tolerant MPSoC design it is not sufficient to seek mappings that are as fast as possible. The total running time of the application is not relevant, but the timing of the perceived output by the user is. Take for example a multimedia application that displays a video. The user does not want to play the video as fast as possible, but he/she wants a timely display of the individual frames. Therefore, we need realtime behavior. In order to obtain realtime behavior, we have extended the scenario-based Sesame with two aspects: explicit I/O modeling and explicit frame barriers. Implementing explicit frame barriers is rather straightforward: an intra application scenario corresponds with a single frame. To communicate the frame barriers to the architecture special event types are introduced signaling the start and the end of a frame. With the use of these barriers, the frame rate can be determined. We should note that simulation cannot give any guarantees about the realtime behavior. However, this is also not yet required in our point in the design phase. Sesame aims to do DSE in the early stages. Therefore, it prunes the design space before more detailed (and probably more expensive) analysis is done in later design phases. Especially in the case of hard-realtime behavior, the modeling investment in later design phases is quite large. Therefore, it is useful to already leave out designs that are not likely to perform well.

Additionally, I/O is modeled explicitly. For this purpose, the model of Sesame is extended in both the application layer and the architecture layer. The application in the application layer is extended with a special Outside World Process (OWP). All output that is visible to the user must be communicated to the OWP. Additionally, external input is modeled by reading from the OWP. All of the other behavior that happens inside the application is invisible for the user. The MPSoC designer can program the OWP to model data that becomes available periodically. The end of an application frame is communicated to the OWP by the KPN node that receives the end-frame event. As a result, the OWP can determine the frame rate of each individual application during the simulation. Next, the architecture layer has an additional I/O component. The OWP in the application layer must be mapped onto one of the I/O components in the specific architecture.

An example of a Sesame model extended with I/O support is given in Figure 2. It is a MJPEG decoder (Figure 2a) with a simple four-processor architecture (Figure 2b). The Video-In process reads in the encoded frames from the OWP. Depending on the situation, all frames can be available instantaneously (i.e. the rate of incoming frames is only bound by the buffer sizes of the communication channels) or arrive with a certain interval. The MJPEG application will decode the frame and after a certain amount of simulated cycles the Video-Out process will send the decoded frame to the OWP. Since the processes know where the frame barriers are, there is no restriction on communication tokens. It can be done frame by frame, but it can also be done pixel by pixel. After the OWP received the complete frame, the delay since the previous frame can be calculated.

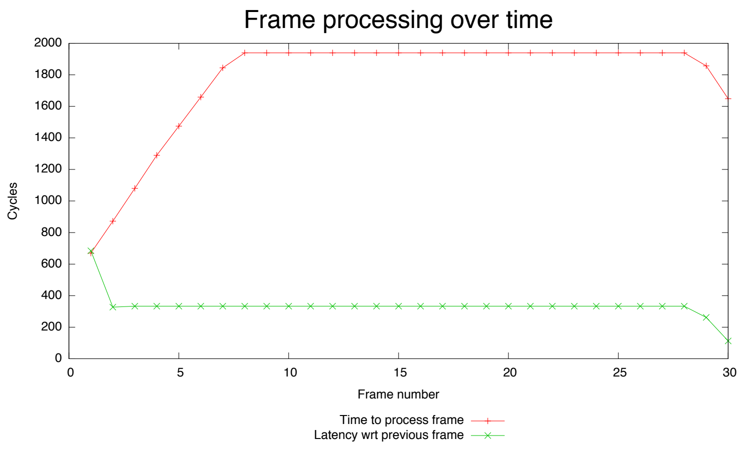


Figure 3: The frame processing time of a MJPEG application

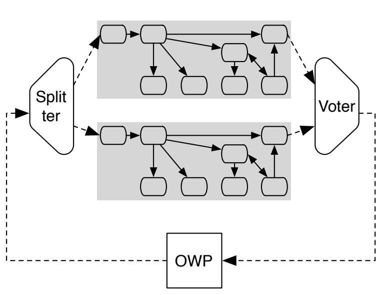
When performing such a simulation, one must take care of how the frame rate is determined. The nice aspect of a KPN-based MJPEG decoder (or any other pipelined application) is that different parts of the MJPEG application can work on different frames. Figure 3 shows some experimental results of the MJPEG application with an application mapping where the four processors are shared between all the nodes in the MJPEG applications. Two metrics are displayed: processing time and latency.

The processing time is the number of cycles an image is processed in the pipeline. More precisely, this is the elapsed time between the read of the frame in the Video-In process (all encoded frames are available instantaneously) and the point in time where the decoded frame is written to the OWP. Initially, the processing time is increasing (until frame 8). In this phase, there is a transition from an empty pipeline to a saturated pipeline. In the empty pipeline the processors are fully dedicated to a single frame, whereas in the saturated case the application is completely occupied with decoding multiple frames simultaneously. At the final frames the reverse behavior can be obtained. No more new frames are fed into the application and the pipeline becomes less saturated.

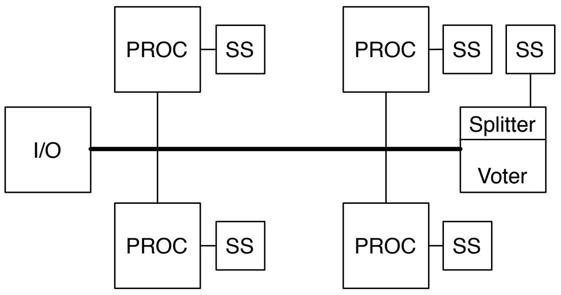
For the frame rate only the saturated case is realistic. The first frame has a rather large delay as the pipeline was completely empty. After that, every 333 cycles a new frame is received. For this simple example the first two frames need to be discarded. With this knowledge, based on 10kHz processors, a frame rate of 30 fps can be obtained. To calculate the real frame rate during the simulation, for arbitrary applications, an automatic warm-up procedure is used. As long as the processing time of a frame is increasing, the frames are discarded. After the first frame with a non-increasing processing time, the real quality metrics of the mapping are obtained.

4 Fault-tolerant KPN Model

Our fault-tolerant KPN model provides fault-tolerance at the computation level. For the communication network, it is assumed that it is implemented in a fault-tolerant manner [4]. In order to achieve a fault-tolerant KPN there are two required aspects that need to be modeled. First, the active redundancy must be implemented. The active redundancy network is implemented using replicates, a splitter and a voter. The replicates do the processing, the splitter and voter handle the external communication. This is visualized in Figure 4. Looking at the application model (Figure 4a), the incoming data from the OWP must be split for the replicates. All of the replicates will process the data. This data is sent to a majority voter. The majority voter compares the different



(a) Application layer



(b) Architecture layer

Figure 4: An example MPSoC model extended with majority voting

outcomes and in case of a majority it can send the verified output to the OWP[1]. As a consequence of the use of majority voting, we only support deterministic processes. This means that, given error-free execution, that the same input will always generate one unique outcome. These splitters and voters do not only need to be modeled in the application layer, but also in the architecture layer (Figure 4b).

Having a majority voting mechanism is not enough. Take the example in Figure 4, which introduces fault-tolerance using two replicates. As long as the execution is error-free, the voter always observes a majority. However, with the occurrence of a transient failure, the error is detected and not masked. In the case of fault masking the transient failure can be masked and is thus invisible to the external world. Fault detection is the case where the fault is detected without the possibility to mask it. In that case, there are two options: restarting and skipping. In the case of skipping, the frame is discarded and (hopefully) the next frame will be available again. Without too many skipped frames, the user can still have a descent quality. In the case of a restart, the decoding of the frame is retried. In order to guarantee that the data of incoming I/O is still present, it must be temporarily stored in stable storage (SS). As the example in Figure 4b shows, both the splitter and the voter have a shared access to the stable storage. When receiving a frame, it is stored in stable storage by the splitter, and once the frame is verified the voter removes it from stable storage again.

Restarting in such a way is fairly coarse grained. After a failure, all the work performed on future frames must be redone. This does not only introduce a lot of restart overhead, but it also results in an empty pipeline that needs to be refilled. Remember the experiment from Figure 3 where we showed that it takes time before a pipelined application is running at full capacity. Therefore, we also implemented a checkpointing model. The checkpointing model can ensure a restart with a full pipeline (e.g. all the processes have work to do, not only the sink node). Additionally, less work needs to be redone. Finally, it also allows us to use fault-tolerant techniques like Roll Forwarding Checking Schemes (RFCS) where the voter activates a spare replica when there is no majority. The work of [10] already discussed the implementation of fault-tolerance techniques in KPNs. However, this work misses the discussion of checkpointing, which is required for the RFCS pattern. This section is structured as follows: The first subsection gives a more detailed description of the implementation of the majority voting and the second subsection describes the checkpoint model.

4.1 Majority voting

A first prerequisite in order to simulate transient failures is a way too incorporate them in the simulation model. As we provide fault-tolerance on a computation level, the only sources of errors are the processors. The processor models the occurrence of failures using an exponential random

[1] Fault-tolerant is not the same as fault free: there is a slight probability that a majority has the same incorrect answer.

distribution. An exponential distribution describes the time between events in a Poisson process. Events in a Poisson process occur continuously and independently at a constant average rate. This is true for transient failures in the form of SUEs (Special Uncorrectable Error). SUE errors are infrequent and the occurrence of it is independent of the previous one. During the simulation, the processor will iteratively 'schedule' transient failures by picking a random exponential number. It depends whether or not the transient error affects the execution of the application. If no process is active at the time the transient failure occurs, the failure does not have any consequences. However, in case a process is active, it will invalidate all future output of the process. In reality, it may the case that the failure does not affect the output, but in our high level approach we cannot know the exact effect. Therefore, we take the most pessimistic assumption.

Processes enclosed in the active redundancy network have two types of communication: internal and external. Internal communication is communication between two processes in the same active redundant network. This communication is unchecked and there is no interaction with the splitter or the voter. External communication must pass via the splitter or the voter. In our fault tolerant KPN model all computation is guarded. This means that of all computation there must be at least two replicates. Rationale is that unguarded computation is not desirable. We take an imperfect architecture as a starting point, hence we need to check all computation on it to be able to obtain correct results.

The splitter is responsible of delivering data to the replicates. As discussed before, this occurs in two steps. At first, the data is temporarily stored in stable storage. This ensures that in case of a restart the data is still available. After the data has been temporarily stored, the data is delivered to the nodes in the replicates that are expecting the data. At this point in time the data is assumed to be correct. The cached data at the splitter may be removed if the frame to which the data corresponds is completely handled and the voter has verified all outgoing data. The caching of data required for restarting introduces another degree of freedom in the design of the fault-tolerant MPSoC. When restarting is enabled (the designer can also choose to leave out the possibility of restarting and save the overhead of caching the data and the silicon cost of the stable storage), the simulation also analyses the amount of stable storage that is required at the splitter. There are two options during simulation: no limitation on the amount of data that can be stored in stable storage (in that case the simulation learns us how much data is required) and a limited amount of storage. With a limited amount of storage, the splitter will limit the number of frames that can be processed simultaneously. Obviously, the stable storage must at least have the size to store the input data of a single frame.

Outgoing external communication always passes the voter. The voter will perform majority voting on the data and in case of a successful voting (e.g. there is a majority), data will be sent to its destination. In case the voting fails (e.g. there is no majority or data comes in too late) it depends on the setting on what the consequence will be. Each voter has a restarting budget. The restarting budget determines the number of times the voter restarts during a frame. If there is still budget for restarting, the connected nodes and splitter will restart from the last checkpoint (more about checkpointing in the next subsection). It can also be the case that the voter has already used its restarting budget for the current frame. In that situation, the output data is flagged on its possible inconsistency and sent out. Voting is done with best effort for the remaining tokens in the frame. The skip flag will be set, irrespective of the existence of a majority during the voting. We have chosen to flag the output because in some applications it can be the case that still something can be done with the possible incorrect output. In case the output communication directly goes to I/O, the frame will be skipped.

4.2 Checkpoint modeling

Earlier, we already motivated the use of checkpoints. A checkpoint involves overhead, but it allows the reduction of skipped frames on faulty processors. The usage of checkpoints depends on the

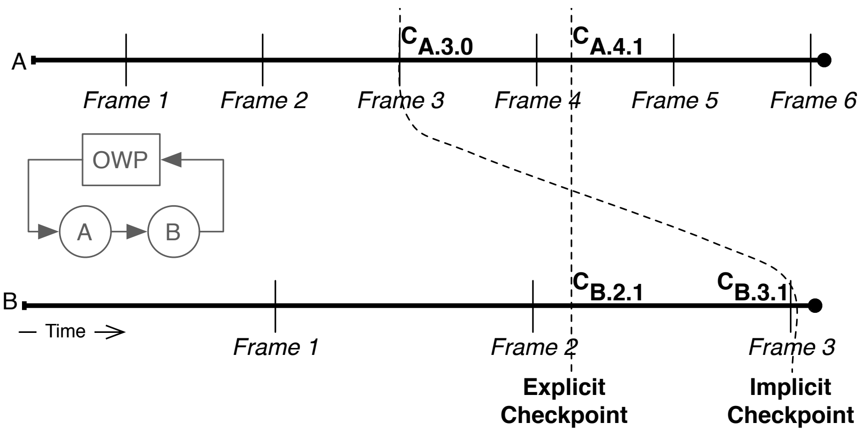


Figure 5: The difference between an implicit and an explicit checkpoint as illustrated by a timeline of a simple application.

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 errors that can be tolerated. Clearly, for a safety critical application, no error can be tolerated. Multimedia applications typically allow for a part of the frames to be skipped, but this fraction must be small enough to be sure the user would not notice it.

As illustrated in Figure 5, there are two types of checkpoints: implicit and explicit. Apart from the cached data in the splitter, implicit checkpoints do not take any overhead. Implicit checkpoints are exactly at the frame barriers. Since there is not state between frames (see Section 2), restarting the application from this execution point is trivial. No state between frames means that the internal communication channels are empty (no state means no internal messages left) and the application is only triggered by incoming external communication. This incoming external communication is already cached by the splitter. Restarting from an implicit checkpoint involves the restart of a process and reading the cached external data from stable storage into the splitter.

The downside of an implicit checkpoint is that with restarting from a specific frame all the work that is done on newer frames is lost. Take the simple application in Figure 5. On a restart from an implicit checkpoint at the end of frame 2, both processes start at the barrier of frame 2. Before any new frame can be delivered both process A and process B need to do some processing. Moreover, process B needs to wait on output of process A before it can do any work. In this time this process is only idling. To resolve this, explicit checkpoints can be taken during the lifetime of an application. Explicit checkpoints are initiated by one of the processes and store the state of all the processes in the active redundancy network and their internal communication channels at a specific point in time.

First, let us discuss the initiation of an explicit checkpoint. For each process, the designer can specify a checkpoint interval. The checkpoint interval decides the number of read events after which the process will request a checkpoint. If before the end of this interval an explicit checkpoint is already made, the interval is reset. The explicit checkpoint is requested at the voter. The voter will determine if it is granted. A reason to dismiss the explicit checkpoint is that the voter is currently handling an earlier error by starting up a restart or skip procedure. If the checkpoint is granted, the voter will ask the splitter to perform a checkpoint procedure. In this way, the voter can handle other votes in the meanwhile. To obtain a consistent checkpoint, all the replicas must stop at the same point in the program. Therefore, each process is queried about its state and based on that a halting point will be determined and communicated to the processes. The processes will execute up to the specific point and make a checkpoint. One of the processes in the active redundancy network will be responsible for collecting the checkpoint. This process will send the checkpoint to the voter. The voter will compare the checkpoints of the different replicas. In case they are unequal, the checkpoint is discarded. Before there can be voted on the checkpoint all the voting on other external output must be completed. In case any of this voting fails, the checkpointing procedure is aborted and the restart or skip procedure is started.

The size of the checkpoint is dependent on the process and the amount of data in the internal communication channels. To reduce the checkpointing overhead, all the processes of the same replica must be running on the same processor. In the example of Figure 5, this can for example mean that processes A and B of replica index 0 are running on processor p1 and processes A and B of replica index 1 are running on processor p3. The checkpoint of each replica is stored in the local stable storage of the specific processor. Simulation will analyze how large this storage is required to be. The voting time of a checkpoint is also dependent on the size of the checkpoint. This is partly due to the contents of the internal communication channels that are checked in order to be sure about correct data once the program is started from the explicit checkpoint.

Implicit and explicit checkpoints can also enhance each other. Take for example the checkpoints in Figure 5. In this Figure a timeline shows the checkpoints. A checkpoint link $C_{B.2.1}$ is a checkpoint state of process B after 2 implicit checkpoints and 1 explicit checkpoint. If there is a restart just after the implicit checkpoint of frame 3, the system can be restarted from a combination of the explicit and the implicit checkpoints. First, the application is restarted to the state of the explicit checkpoint. Next, process B can be shifted to the implicit checkpoint by discarding the messages of frame 3 in the internal communication channel.

5 Fault-tolerant KPN Segregation

Until now, only the complete replication of an application has been discussed. However, this is only one alternative in applying active redundancy. The two extreme methods are: replicate the entire application or replicate each individual process. In the individual case, each process has its own splitter and voter and no internal communication is present. However, there are many other ways of applying active redundancy.

In the general case, the application can be segregated into a number of sub-networks. The active redundancy pattern is applied to these sub-networks. The sub-network should be connected (for the connectivity analysis, the communication channels are considered to be undirected). In case it is not, the sub-network is split in multiple sub-networks in such a way that all the sub-networks are connected.

Figure 6 gives an example of a possible segregation of an application. In this case, there are three sub-networks: A, BC and D. The choice of how to implement the active redundancy is independent for each sub-network. It can be the case that A is implemented using Triple Modular Redundancy (TMR), whereas BC only has Double Modular Redundancy (DMR). Additionally, A can be instrumented to do checkpointing, whereas in the case of D checkpointing is completely disabled.

From the example in Figure 6, it also becomes clear why the sub-networks need to be connected. If we would have taken AD and BC as a sub-network, A and D are not connected but still in the same active redundancy network. As a consequence, D will be restarted together with A in the case there is an error in the output of A. However, D is not connected to A and it is perfectly possible to continue execution of D because it only has verified input from the splitter.

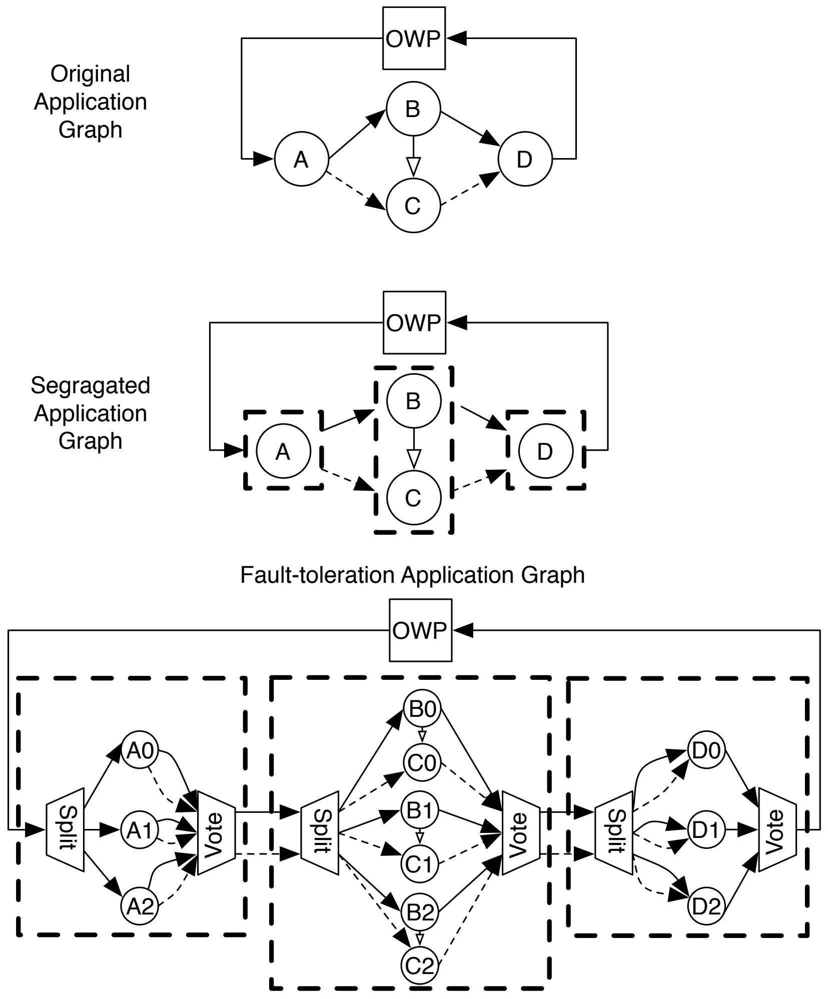


Figure 6: An example of application segregation in order apply active redundancy on parts of the application instead of applying it on the complete application.

The benefit of exploring the application segregation is the possibility to leverage between fault-tolerance overhead and performance. The more communication is done within a sub-network, the less checking has to be done by the voter. However, larger sub-networks also involve larger overheads for restarting and checkpointing. On top of that, computational intensive parts of an application can be implemented using TMR (less probability that it needs to be restarted) while the less computationally parts can be put in DMR (restarting is not expensive, and with DMR voting is easier and cheaper). There are many ways to segregate an application with unpredictable consequences on the quality of the resulting MPSoC. Therefore, an automated Design Space Exploration (DSE) is required, which is fault-tolerance aware.

6 Fault-tolerant aware DSE

The fault-tolerant MPSoCs as we presented so far have many degrees of freedom. Therefore, there are many ways of implementing an MPSoC. To guide this complex process, we need to perform a DSE. The DSE is performed using a python module called DEAP [7], deploying NSGA2 [1] to search the design space. The degrees of freedom that are encoded in the chromosome are as follows:

- Per Process
  - Sub-network id
  - Explicit Checkpoint Interval
- Per Sub-network
  - Used type of active redundancy (DMR / TMR)
  - Architectural processors selection
  - Max Restarts per frame

For each process in the applications, a voter is selected and an interval for the request of an explicit checkpoint. In case this interval is zero, it will not request explicit checkpoints. The application segregation will be determined based on which processes are assigned to the same voter. If the sub-network assigned to the voter is not connected, the chromosome is repaired by splitting the sub-network and assigning the new sub-networks to other (unused) voters.

To make a fixed length chromosome, the maximal number of sub-networks is described in the chromosome. This maximal number of sub-networks is equal to the number of processes. Each sub-network has a different implementation of the active redundancy network (2 or 3 replicas). Additionally, the architectural processors are selected that are used to map the replicated processes on. The number of processors that is selected is equal to the number of replicas and can contain duplicated entries. In case of a duplicated entry, redundancy in time is modeled. Different entries indicate redundancy in space. The maximal number of restarts indicates the maximal number of restarts within a frame before it is skipped. In case the maximal number of restarts is 0, the checkpointing of the sub-network is completely disabled.

The DSE performs a multi-objective optimization. The objectives that we are using are as follows:

- Missed deadlines
- Energy
- Skipped frames
- Used Stable Storage

7 Adaptive System perspective

As a next step in the coming year, we plan to incorporate real adaptivity. The adaptivity allows the MPSoC not only to tolerate failures, but also to adapt the system to maintain a certain Quality of Service. For this purpose, the DSE is extended to search for optimal mappings for multiple so-called architecture scenarios. Architecture scenarios are run-time scenarios describing the state of the system (which processors are active and which are not). In case of errors, the adaptive MPSoC can re-map the processes that are executing on the defective processor and restart them on a different processor, which is facilitated by the checkpointing mechanism that has been described in this report. To this end, the re-mapping process uses information from the design-time DSE and, based on the state of the system, applies the mapping of the "nearest" architecture scenario to provide graceful degradation on the MPSoC.

Important to notice is that remapping may be expensive, but that this is not an issue in our case. Remapping will be done in case of processors having an unexpected high failure rate or are completely broken down. This is not expected to happen regularly. Therefore, it is acceptable to take some time. However, a complete DSE on chip is in most cases unfeasible. Therefore, some preprocessed information must be available in order to take the right decisions.

8 Interactions with other MADNESS partners and WPs

The work described in this document has been aligned to the work performed by USI in WP 5. We should note that WP 5 is mainly focusing on permanent failures, whereas in our case transient failures (and thus checkpointing) are also taken into account. This, for example, allows us to reason about the amount of skipped frames, which is relevant for multimedia applications that can tolerate some missing frames. So far, however, we did not yet have the opportunity (as our work has started only recently) to arrange physical meetings with USI for more detailed harmonization of our research. This is planned for the first quarter of 2012. Moreover, meetings with Leiden University are also planned to align our work with their work on the support for run-time management of MPSoCs within WP 6.

Summary

In this report, we provided an overview of the work that has been performed by the University of Amsterdam in WP 6 during the second year of the MADNESS project. More specifically, we described the steps we have taken to make our Sesame modeling and simulation environment as well as the DSE process using Sesame fault-tolerance aware. These steps are prerequisite for modeling, simulating and performing DSE of adaptive, fault-tolerant MPSoCs.

References

[1] K. Deb et al. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. on Evolutionary Computation, 6(2):182–197, April 2002.

[2] Stefan Valentin Gheorghita et al. System-scenario-based design of dynamic embedded systems. ACM Transactions on Design Automation of Electronic Systems, 14(1):1–45, 2009.

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

[4] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C.R. Das. Exploring fault-tolerant network-on-chip architectures. In International Conference on Dependable Systems and Networks (DSN 2006), pages 93–104, June 2006.

[5] JoAnn M. Paul, Donald E. Thomas, and Alex Bobrek. Scenario-oriented design for single-chip heterogeneous multiprocessors. IEEE Trans. on VLSI Syst., 14(8):868–880, August 2006.

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

[7] F. M. De Rainville, F. A. Fortin, C. Gagne, and M. Parizeau. Evolutionary algorithms in python. http://deap.googlecode.com, October 2011.

[8] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In Proceedings of the International Conference on Dependable Systems and Networks (DSN'02), pages 389 – 398, 2002.

[9] Peter van Stralen and Andy 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.

[10] Onur Derin, Erkan Diken, and Leandro Fiorin, "A Middleware Approach to Achieving Fault Tolerance of Kahn Process Networks on Networks on Chips", International Journal of Reconfigurable Computing, vol. 2011, Article ID 295385, 15 pages, 2011. doi:10.1155/2011/295385