Hierarchical Self-repair in Heterogenous Multi-core Systems by Means of a Software-based Reconfiguration

Sebastian Müller, Mario Schölzel, Heinrich Theodor Vierhaus

Institut für Informatik
BTU-Cottbus
Walther-Pauer-Str. 2
03046 Cottbus
\{smuelle2 \textbar mas \textbar htv\}@informatik.tu-cottbus.de

Abstract: This paper deals with the problem of a software-based self-repair in a statically scheduled multi-core system in the presence of permanent faults. The basic idea is to adapt the application in a way that the use of faulty components is avoided. This goal is achieved by re-compiling the program-code via an off-line repair process in the field. The repair process is organized in a hierarchical manner. At the beginning a local repair is applied considering only one core. If the local repair fails, a retry at a higher system-level is performed. For that purpose, the local repair techniques are re-used in a global context. The repair at a global system level results in some specific system constraints and properties, which are investigated in this work. Due to the use of pure software-based methods one gains the possibility to repair defects in different components (even multi errors) or defects in spare components. The presented approach is not bounded to a concrete architecture and is therefore adaptable to systems like MPSoCs or NoCs, if these systems provide some basic functionality.

1 Introduction

The on-going downscaling of the feature-size for integrated circuits (currently 32nm) provides the possibility to combine a higher amount of transistors with a decreased switching time and a decreased energy consumption. But these positive effects also come along with negative ones, e.g. a higher vulnerability to faults and early life failures due to wear-out effects. The reasons are, on the one hand, higher stress density [BGM04] and, on the other hand, deviations in the production [MG04]. To lower the production costs and to guarantee certain emergency properties it is necessary to develop fault tolerant systems, which can cope with such permanent faults. Due to the possibility of integrating an increasing amount of transistor in an steadily decreasing chip area, we can observe the trend to multi-core systems [OH05]. To design such systems as fault tolerant, it can be helpful to combine techniques, which were developed isolated, in a hierarchical approach.

Statically scheduled VLIW-processors (very long instruction word) are easily scalable and therefore well adaptable to different requirements of certain applications. In addition, they provide a feasible degree of performance, due to the super-scalar architecture, for signal-
and video-processing in an automotive or avionic environment. A crucial advantage of a VLIW-architecture is the small controlling logic. In contrast, the control logic in dynamically scheduled processors makes up a major portion of the overall gate count. These dynamic scheduled processors are therefore hardly to repair.

2 Related Work

In the past decades, several hardware-based as well as software-based self-repair approaches have been proposed for processor based systems. An established approach is the re-configuration of a FPGA in the presence of permanent defects [MHS+04]. In [MHS+04] the re-configuration is done by a micro-controller which is implemented in a second FPGA. The disadvantage of this approach is the increase of the necessary memory size (configuration data) and the fact that an FPGA-implementation, in contrast to an ASIC-implementation, tends to have a higher consumption of area and energy.

To replace logic blocks in non-FPGA systems it is possible to use switches, which can shutdown blocks and activates spare-blocks [KSV09]. Such an approach is only feasible for large logic blocks, since the administration of the repair requires additional hardware which is error prone itself. To gain an actual improvement of such a fault-tolerant system, it is necessary that the portion of the administrative logic is relatively small compared with the overall chip area [KV11].

There exist other approaches, which focus on a software-based self-repair. In [KKP00] it is proposed to calculate all necessary schedules for every fault situation in advance. These schedules are stored in the memory and the appropriate schedule is executed at run-time, according to the detected faults. The obvious disadvantage of this method is the extra cost in memory for storing all these additional schedules.

Another approach is presented in [MS08]. It is proposed to implement most of the hardware operations in a software routine. If an operator is defective, its function is assumed by the associated software implementation. In case of an error the performance degradation might be substantial due the overhead of the implementation in software.

The presented solutions so far are all bounded to the repair of at least one single core or only a few components in a single core. But just with the upcoming of multi-core systems further techniques for handling permanent faults in such an environment have been proposed. In [ARJS07] and [RS08] similar approaches are presented. The idea is that a defective core can borrow resources from other cores in the system; e.g. a core $C_1$ can use the execution stage of a different core $C_2$ to finish the execution of one of its operations. The major disadvantage of this approach can be found in the administration. For this purpose it is necessary to extend the control logic and to enlarge the connection between the cores.

There exist further publications, in which the repair of permanent faults is done with the help of virtualization in homogenous multi-core systems [Jos06, CASP10]. In [Jos06] a hypervisor is introduced between the hardware and the operating system. The hypervisor can e.g. emulate operators in software, and it can shift threads from a faulty core to
a non-fault one. In [CASP10] an additional layer is introduced, which maps defective components of a certain core to a fault-free core.

All these methods are bounded to homogenous multi-core processors. Moreover, all the presented methods have a low system performance in case that an error is present. This performance degradation arises from the virtualization and therefore an additional runtime due to the necessary software layers.

3 Description of the Basic System

The multi-core system, depicted in fig. 1, consists of several VLIW-Processors, the necessary program memories, one common used data memory, and a connection network. All components are clocked synchronous via a global clock. The processors can be heterogeneous, if they belong to a family of processors. To every processor $C_i$ exists one program memory $P_i$. The cores are not connected directly to their ROM. Instead, they are connected to the connection network, which implements the external interface for every processor. Via the connection network every access to the program and data memories is handled as well as the communication between the cores. The controller of the connection network guarantees a mutual exclusion regarding the access to resources of the system.

![Figure 1: Scalable MPSoC with several cores, ROMs and one shared memory](image)

A single VLIW-core has no information about the global design of the system. Every core addresses a local program memory and a common used data memory, which seems to be exclusive for every core. Furthermore, there are I/O-ports available for every core to communicate with external components. The cores are implemented in a VLIW-architecture. The advantage of such an architecture is its scalability. With such architecture a varying range of several heterogeneous systems can be configured in an easy manner. A VLIW-core consists typically of multiple execution slots. The calculation in one slot is done
by an FU (functional unit), whereas those can perform different operations. A statically scheduled instruction word controls in every cycle, which operation is calculated in which slot. There is no dynamic scheduling within the processor. The scheduling is done at the compile time of the system. The given processor has a 4-stage pipeline (FE, DE, EXE and WB) as well as a bypass network to avoid hazards.

In the program memory of every core exists a program for a self-test. The software-based self-test is executed at the system start-up or during short breaks of the system runtime. Even the spare cores executes this self-test. The results of the self-test are stored in a dedicated part of the data memory. In the presence of any error an appropriate self-repair routine is executed. The self-repair is software-based and can cover local repair techniques as well as global ones. The major goal of the repair is to re-configure the application in a way that the use of faulty components is avoided.

4 Local Software-based Self-repair

This section presents two repairing techniques which can be applied as a local repair in a VLIW processor. The first method implements a re-binding, whereas the second one executes a re-scheduling based on entire basic blocks. Both techniques re-configure the application to avoid the use of defective components. This adaption is done as an off-line repair, and the repair process itself is a pure software-based solution.

The software-based re-binding, developed in [Sch09, SM10], binds the operations of an instruction word to a different hardware resource in the case that the original binding plans the use of a defective component. The algorithm iterates over the program memory and transfers in every step initially one instruction into the data memory. After that, every operation is bound to an unused non-defective resource. Then the new created instruction word replaces the old instruction in the program memory. The software-based re-binding is a fast technique in respect to the execution time, which can only handle single cycle operations.

A more powerful technique is called Scoreboarding, which can handle multi-errors for any number of components. The method is described in detail in [SM10]. The basic idea is the following. The program code of an application is re-compiled stepwise at the granularity of basic blocks. The algorithm transfers at the beginning an entire basic block from the program memory into the data memory. The machine code is then inserted into a priority list. Based on the priority list, a re-scheduling is executed, which generates a new schedule for the basic block. For re-scheduling a simple list-scheduling algorithm is used. The schedule is converted into machine code and written back into the program memory, where it replaces the old machine code. The Scoreboarding algorithm takes into account that several components can be faulty. Due to that a fine grained repair of the system is possible with this technique. The repairable defects can be located in different system components like single operators, FUs, register of the pipeline, regular registers or read- and write-ports. The presented method is a pure software-based technique, which is executed at the system start-up.
5 Global Self-repair

If a repair at the local level fails, the next objective is to apply a global repair strategy. A local repair can fail, if there are too many defects present or if a critical component (e.g. all components of the control path) fails. This can lead to severe problems during the repair process. A first scenario could be that the repair algorithm is not able to determine a valid schedule. In another scenario, it is possible that the software-based self-repair itself is not correctly executable. This problem can arise if the repair algorithm uses a defective component (e.g. a faulty branch unit). In the first case, a spare core $C_s$ shall overtake the task of the defective core. In the latter case it is sufficient to execute an appropriate repair algorithm, which does not use any defective component. This can be achieved by e.g. repairing the repair algorithm itself.

**Case 1: Compilation to a spare core**

Figure 2 shows the corresponding process for the global repair strategy. In this example it is assumed that core $C_x$ is faulty and its task shall be overtaken by core $C_s$. To achieve this it is necessary to transfer all relevant parts of the program memory of $C_x$ into the memory of $C_s$. Meanwhile it is possible to replace the program code of $C_x$ with nop instructions, so that $C_x$ is denied from any writing access to system resources. Due to the fact that $C_x$ might differ in its architecture from $C_s$, it can be necessary to adapt the program code to the architecture of $C_s$. For that the scheduling method, which was described in the last chapter, will be applied. With the help of the Scoreboarding algorithm the application of $C_x$ is re-compiled to the architecture of $C_s$, whereas the algorithm consideres parameters like FU amount and operator configuration. After completing the re-compilation of the application the repair process is done and the regular system execution can be started.

**Figure 2:** (a) Transferring the program code from $C_x$ into the ROM of $C_s$; (b) Overwriting the ROM of $C_x$ with NOP-Instructions and adapting the program code to $C_s$

**Case 2: Repair of the Repair Algorithm**

This section presents two strategies of how a repair algorithm can be adapted to a certain defect situation. The first method adapts the program code of the repair algorithm at the
compile time of the system. In comparison, a second method will execute the adaption in
the field as a off-line repair procedure.

In the first approach the decision, which version of a repair algorithm has to be executed, is
based on differently compiled versions of this algorithm. At system compile time, several
versions $v_i$ of the same algorithm are generated. Every version $v_i$ uses only operators of
an corresponding FU $f_i$. As long as there exist on fault-free FU $f_j$ in the datapath, it also
exists a working version $v_j$ of the repair algorithm. An external core $C_e$ can determine
at repair time an appropriate algorithm version, based on the results of the self-test. After
determining a version $v$, it is necessary to start the execution of $v$ on the defective core.
To do so $C_e$ manipulates a jump instruction in a way that the target of the instruction is
the start address of $v$. Figure 3 shows the procedure of launching an appropriate algorithm
version.

![Diagram](image)

Figure 3: Launching an appropriate algorithm version by changing a jump address

In the second alternative, we propose to adapt the repair algorithm in the field in an off-
line repair process. The advantage is that no additional schedules have to be stored in the
memory. This approach is especially effective, if defects in the regular registers or the
read- and write-ports of the register file can occur. In such a case, it is not practicable
to calculate several schedules in advance for all possible combinations of faults. For that
purpose it is more feasible to adapt the local repair algorithm in the field to the current fault
situation. Thereby the procedure is quit similar to the procedure in the first alternative. Thedifference is that no spare core is necessary, and instead only the repair algorithm in the
ROM of a defective core $C_x$ is changed. The residue application of $C_x$ is self-repaired by
$C_x$ after its repair algorithm is re-compiled. To support this repair strategy, it is helpful to
compile the local repair algorithm only with a few or even with no parallelism at instruction
word level. This can be done by the compiler at the compile time of the system.

## 6 Extension of the System

The proposed repair strategy has some consequences, which affect the system architecture.
The first issue to mention is that it is necessary to organize and administrate the repair in
an appropriate manner. A next important point is that the requirements concerning the
connection network has changed; e.g. access to the program memories. Other points are the extensions regarding the spare core and creating necessary meta-data for supporting the repair. These meta-data can be generated and gathered during the compile time of the system.

6.1 Organisation and Administration of the Repair

The system start-up is divided into two phases. At the end of a phase, every core has to be in a valid state. In the first phase, every core (even spare cores) execute a software-based self-test followed by a local software-based self-repair if necessary. At the end of every sub phase, the result (e.g. fault detected, repair successful) is written into the shared memory. For coordinating the organisation and administration of the repair, a designated core of the system (referenced as $C_M$) is determined. The core which finishes first successfully the first phase of the system start-up declares itself as master core $C_M$. After that $C_M$ informs the other cores about this fact via the shared program variable $mID$. The evaluation of all of the results generated in the first phase is done by $C_M$. The possible situations are the following:

- Case 1: No defects respective successfully repaired
- Case 2: One or more cores are faulty but not repaired yet (Repair failed or did not stop)
- Case 3: One or more cores are without result (Self-test did not stop)

After determining the master $C_M$, the second phase is started. First of all $C_M$ waits until the necessary execution time of the several self-tests is over. Thereafter $C_M$ knows if a local self-repair is executed and waits for the end of it as well. Now $C_M$ can determine, which of the latter described cases is present. If case one is present, the regular system execution can be started. The other two cases will be handled as follows:

**Case 2:**
In this case a faulty core $C_x$ could successfully determine its fault situation. However, the local repair was not able to generate a valid schedule. $C_M$ determines, based on the results in the shared memory, that the execution of the self-test was finished successfully, but that the self-repair failed. There are two reasons why a local self-repair might fail:

- 1. If the local self-repair algorithm uses defective components it can not be executed correctly. Therefore the repair algorithm might not stop (e.g. defective branch unit) or it calculates wrong results (e.g. defect in a register).
- 2. Due to many faults or a lack of resources, it is not possible to determine a valid schedule. A possible scenario might be that all operators of a certain type are faulty. In certain circumstances it is also thinkable that undetected errors influence the repair algorithm.
can in the first case adapt the local repair algorithm with the strategy described in case 2 of the global repair. After that \( C_x \) could re-executed the local repair algorithm with the appropriate algorithm. Alternatively \( C_M \) can execute the local repair and re-schedule the application of \( C_x \). As already mentioned, the second strategy is useful, if registers are faulty.

In the second case, core \( C_x \) has to be replaced by a spare core due to many or critical faults. It is assumed that undetected errors can occur. Therefore an external software-based test is executed to re-investigate \( C_x \) for undetected errors before a spare core is used in the end. If any differences between the results are identified, the self-repair is re-executed with the new information.

The last thing to mention is how \( C_M \) can decide whether a local repair algorithm uses a faulty component or not. It would normally imply that every instruction word has to be checked for a certain binding to a faulty component. As solution it is proposed that the compiler (at system compile time) compiles the algorithm in a way that only the first slot is used. If a fault in the first slot is present, \( C_M \) now knows that the local repair algorithm is not executable. A similar agreement can be made regarding the registers by declaring a fixed range of register as usable.

**Case 3:**
In this case, no result of the self-test of core \( C_x \) is present. \( C_M \) can check this by evaluating the shared memory. In this scenario \( C_M \) externally re-tests \( C_x \). After that \( C_x \) will be repaired with respect to the detected faults. This can be done externally or locally. It is marked that \( C_x \) has been successfully tested. If now a repair fails, the situation changes into the second case, which was discussed before.

### 6.2 Requirements for the Connection Network

The access to the program memories is managed by the connection network. Every core can communicate with the connection network via certain I/O-ports. Three different ports are necessary. The first port specifies the address in the program memory, the second port declares the address in the data memory, and the third port specifies the required command word. In the command word is the access mode (read or write) and the target core coded. The connection network ensures a mutual exclusion during the access to the program memories. Every core, which requests a further access during a granted access, will be set into a stall-mode.

During system start-up no write access to the program memories will be granted by the network. Therefore it is avoided that a faulty core accidently writes to system resources. At the global system repair phase, write access has to be explicitly activated by sending a certain command via an I/O-port to the controller of the connection network.

In the last section it was described how the core \( C_M \) is determined. After \( C_M \) has declared it self as master, this fact is communicated to the other cores via the global variable \( mID \). The access to this variable has to take place exclusively. Therefore the system is extended with a global instrument for synchronisation. To do so, a dedicated port \( p_m \) is declared,
which has to be accessed before an access to $mID$ is issued. The synchronisation via port $p_m$ is managed by the controller of the connection network. After an access to $p_m$, no further access will be granted for the next 10 clock cycles. If the controller registers a further access within this 10 cycles, the corresponding core will be stalled.

6.3 Metadata for the Administration

For coordinating the repair, it is necessary to provide some information about the system to the core $C_M$. These data are stored in a data structure, to which every core has access during the runtime of the system. The necessary data are the amount and id of the cores, the runtime of the self-tests, start addresses and size of the repair algorithm, memory address of the self-test results, and addresses of the jump instructions into the local self-repair routines. The determination of the metadata takes place at the compile time of the system. This task can be done automated by the compiler. The runtimes of the local self-tests can be determined by simulation and profiling of the algorithm. At the end, all data are gathered in one data structure. This data structure is stored in the shared memory.

7 Results

The presented multi-core system has been implemented in VHDL (and synthesized) and, furthermore, a scalable multi-core simulation environment has been developed in C++. The simulator emulates for every VLIW-core in the MPSoC one instruction set simulator. The proposed software-based self-repair algorithms have been developed in assembler, translated into binary code, and tested on the VHDL-model and in the simulation environment.

7.1 Synthese Results

Two VHDL-Implementations have been developed. The first implementation is the non fault-tolerant system with four cores. In a next step this system was extended with a spare core and the necessary functionality for our fault-tolerant strategies. Both systems were synthesized using a 45nm library. The delay on the longest critical path amounts to 2.9 ns. Therefore the resulting system clock is about 340 MHz. Table 1 shows additional results of synthesize for some selected components and its area dimensions. The major portion of the overall chip area belongs to the data path, whereas the control path makes up only a few percent. This is the major advantage of such a VLIW-Architecture. The probability that an error in the control path occurs, is decisively lower compared to the occurrence of an error in the data path. Furthermore our repair strategy concentrates on components like the FUs, registers and ports, which define the major part of the data path. The necessary extensions of the connection network which are crucial for our software-based repair are,
compared to the overall chip area, relatively small with only 2.8 percent. This is an another
advantage in contrast to a fault-tolerant hardware solution.

Table 1: Synthese results (relative area dimensions) of the original system (4 cores) and the fault-
tolerant system (4 + 1 cores) - values in Nand2-equivalents

<table>
<thead>
<tr>
<th>Component</th>
<th>Original System</th>
<th>Fault-tolerant System</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPSoC overall</td>
<td>92240</td>
<td>118510</td>
</tr>
<tr>
<td>Connection network</td>
<td>1274</td>
<td>2844</td>
</tr>
<tr>
<td>1 core</td>
<td>22869</td>
<td></td>
</tr>
<tr>
<td>1 slot</td>
<td>965</td>
<td></td>
</tr>
<tr>
<td>1 FU</td>
<td>748</td>
<td></td>
</tr>
<tr>
<td>Register file</td>
<td>14835</td>
<td></td>
</tr>
<tr>
<td>Control path</td>
<td>270</td>
<td></td>
</tr>
</tbody>
</table>

7.2 Runtime of the Software-based Re-binding

The following scenario (Case 2 of the global repair) presents the necessary execution time
of a global repair strategy if the local repair algorithm is adapted to the current fault situ-
ation. The program code of a core $C_x$ is adapted by the help of a software-based Re-binding
executed on core $C_M$. The code size, which has to be repaired, amounts to about 500 in-
struction words. The Re-binding algorithm reads in every step one instruction word from
the program memory, disassembles it, creates a new binding, assembles it, and overwrites
the old instruction word in the program memory with the newly created one. The neces-
sary overall execution time for repairing a single instruction word is 306 cycles. To adapt
the Scoreboarding algorithm (around 500 instruction words), the repair will need 0.72 ms
of execution time in respect to a system clock of 340 MHz.

7.3 System Reliability

Based on the synthesis results the overall system reliability of the original system and the
fault-tolerant system has been determined. Figure 4 compares the reliabilities of both
systems against each other. The reliability was calculated in respect to a constant rate of
failure. For the calculation of the reliability of the original system, the reliabilities of the
four cores and the non-fault tolerant connection network have been used. For the fault-
tolerant system, the reliabilities of the spare core and of the extended connection network
were used. The fault-tolerant system is modelled as 4-out-of-5 system due to the fact
that a failing of an entire core can be tolerated. Figure 4 shows that the system with a
software-based repair (Rft) provides a higher system reliability then the original system.
We presented a fault-tolerant heterogeneous multi-core system. The applied repair strategies are organized in a hierarchical manner, and the main focus lays on a software-based self-repair. The proposed method abstracts from a certain architecture and can be employed on a varying set of platforms (MPSoCs, NoCs). A prototypical system has been implemented and synthesized in VHDL. The results of the synthesis process were used to determine the reliability of different system configurations. Further investigations have been done with a scalable simulation environment.

The main advantage of combining different repair techniques in one hierarchical strategy is that a local fine-grained repair is possible as well as a repair at system level. A system stays functional even if critical components of a core are faulty. Due to the hierarchical approach the amount of critical components, which causes a total breakdown of a multicore system, is further reduced.

References


