Concepts for Autonomic Integrated Systems

Walter Stechele[1)], Oliver Bringmann[2)], Rolf Ernst[3)], Andreas Herkersdorf[1)], Katharina Hojenski[4)], Peter Janacik[4)], Franz Rammig[4)], Jürgen Teich[5)], Norbert Wehn[6)], Johannes Zeppenfeld[1)], Daniel Ziener[5)]
[1)] TU München, [2)] FZI Karlsruhe, [3)] TU Braunschweig, [4)] Uni Paderborn, [5)] Uni Erlangen, [6)] TU Kaiserslautern

Abstract

Future MPSoCs have to cope with unreliable functionality of their nanometer-scale internal components, mainly due to increasing sensitivity for technology parameter variations and natural radiation. While error correction is widely known in memory design, protection for arithmetic logic units within CPUs is an issue for future research. For on-chip communication resources there are many error protection techniques available, but a trade-off has to be found between techniques on various levels. In this paper we present our concept for Autonomic MPSoCs with capabilities for runtime detection and correction of sporadic errors, and adaptation of performance, power, and dependability on changing environmental conditions. Functional elements of the MPSoC are continuously monitored and controlled by autonomic elements. Re-distribution of tasks is supported by run-time performance analysis and an autonomic operating system. Life time dependability of the MPSoC is introduced in the design optimization process.

1. Introduction

In many complex systems, Multi Processor Systems on Chip (MPSoC) have proven to be efficient platforms to implement application solutions. They are used to control complex systems, like avionic or automotive. Future MPSoCs will face increasing challenges arising from nanoelectronic technology parameter variations, soft errors through environmental radiation, and sporadic timing errors. To cope with such challenges, we propose to design MPSoCs[1] with some degree of autonomic behavior, to give them capabilities to detect unwanted effects and to adjust themselves to maintain functionality and to optimize performance and power consumption. This requires introducing new self-organizing techniques in the hardware and software design process, on system level and component level.
To design reliable MPSoCs, with the capability to autonomously react on internal or external disturbances, requires a new paradigm in the design process. Not only performance, area, and power have to be in the focus of the designer, but also the online monitoring of the system's behavior and its reaction on disturbances. This means, that adequate sensors, evaluators, and actors inside the MPSoC have to register and analyze sporadic disturbances and trigger adequate reactions. In addition to error detection and correction on the hardware level, an autonomic operating system has to support self-organizing task migration between various computing elements.

Our proposed Autonomic MPSoCs will consist of three logical layers: A functional hardware layer, an autonomic hardware layer, and an autonomic operating system layer (fig. 1). The functional layer contains the usual IP components or Functional Elements (FEs). The autonomic hardware layer consists of Autonomic Elements (AEs) and an interconnect structure among various AEs. FEs are either general purpose CPUs, memories, on-chip busses, special purpose processing units (PUs) or system and network interfaces as in conventional, non-autonomous designs. AEs on the other hand contain any extensions necessary to improve the reliability of the FE and, thus, convert the FE-AE pairs into autonomous units. This FE/AE split represents only a logical structuring concept; in reality both FEs and AEs will be integrated on the same CMOS substrate. The autonomic operating system layer will enable the dynamic and self-optimizing distribution of operating system services among the MPSoC nodes.
Our Autonomic MPSoC architecture presents a smooth shift from present non-autonomous to autonomous designs. In fact, semiconductor companies can continue using existing IP-libraries by extending them with the proper autonomic elements. This will preserve the huge investments used to build those IP-libraries.
For Autonomic MPSoCs we will investigate many techniques from classical fault tolerance [Avi97] [Avr01], e.g. hardware redundancy, information redundancy, time redundancy, and mixed redundancy techniques. In contrast to classical fault tolerance, we will pursue different approaches as well: (1) We will investigate fault tolerance techniques on chip level with substantially lower overhead than classical hard-

[1] This work will be carried out in the edacentrum cluster research project "Autonomic Integrated Systems", funded by BMBF.

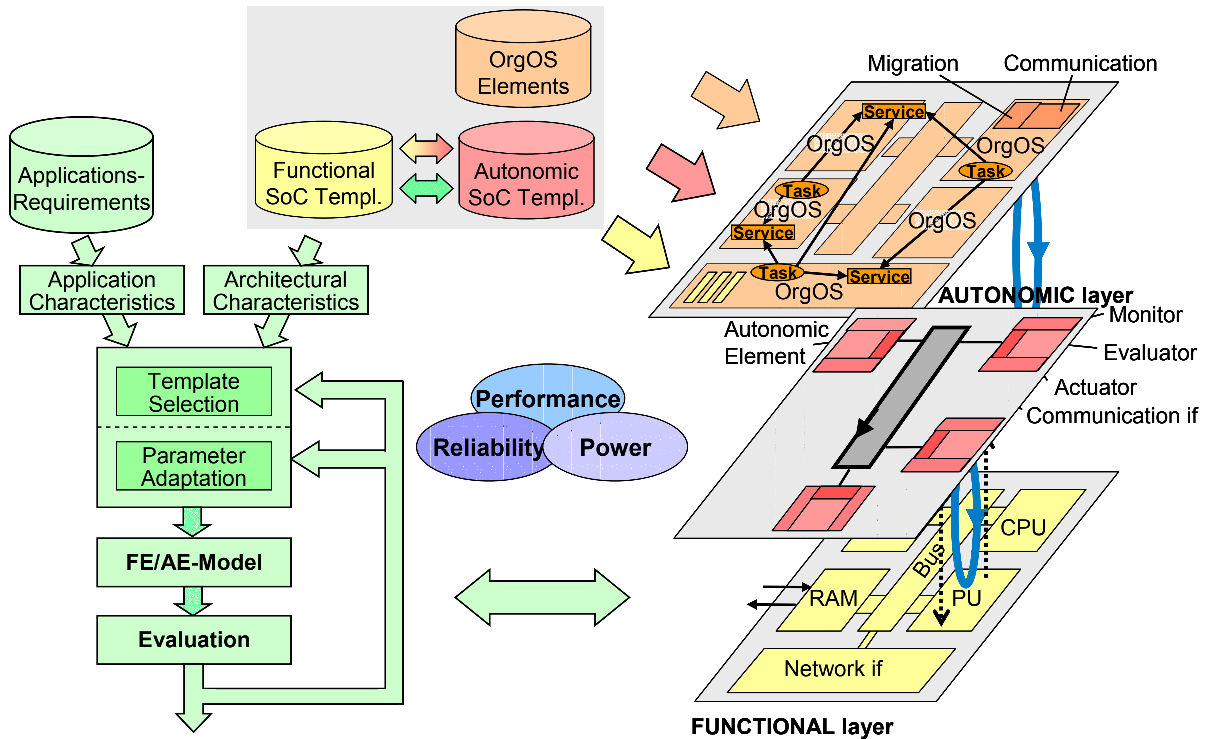

Figure 1: Autonomic MPSoC (right) with functional hardware layer (right bottom), autonomic hardware layer (right middle), autonomic operating system layer (right top), and related design flow (left).

ware redundancy techniques. (2) We will investigate online learning capabilities of the MPSoC to achieve emergent behavior that was not completely determined at design time. (3) We will focus on reuse of existing IP-libraries. (4) We will target not only fault tolerant systems, but also contribute to improve yield for general MPSoCs by giving them the capability to "learn to live with defects".

In the following section we will present system-level techniques, i.e. design methods, autonomic operating systems, and performance analysis. Section 3 will cover component-level techniques, including CPU datapath, controlpath, and communication resources.

2. System-level techniques

2.1 Design methods

In order to be able to design MPSoCs which can cope with the challenges that are posed by future CMOS technology nodes, a major shift in the applied SoC design methods is necessary. While performance and power dissipation are already taken into account as optimization goals at system level, the inclusion of reliability is still an open problem.

A new reliability-aware system level design methodology should target the following objectives: (1) Optimized incorporation of mechanisms that allow an autonomous reaction of the system in case of an error to maintain functionality. (2) Usage of reliability as design parameter beside the traditional area/performance metrics. (3) Fault state exploration during design time by using transaction level fault injection strategies combined with dynamic analysis of error propagation and masking effects. (4) Designing an architecture that is well-suited to cope with thermal and mechanical stresses and time-dependent variability.

The foundation for the design process is build by an executable transaction level system model written in SystemC, that allows the exploration of design decisions/parameter sets with respect to performance, power dissipation and reliability. Using these design decisions and parameter sets, the behavior of the system can be influenced.

The model has to reflect the most important properties of the three logical system layers, like e.g. the capability of the operating system to migrate tasks, the policies for power reduction, that are implemented inside the hardware units or the process variability, that occurs during the manufacturing of integrated semiconductor devices, as accurately as possible.

For a selection of the AEs consisting of monitors, actuators and evaluators regarding their suitability for the realization of the principles of self-organization, self-healing, self-optimization and self-protection in the corresponding system components, these have to be modeled and rated.

As actuators, units can be considered that perform dynamic voltage and frequency scaling (DVFS) or that apply a packet re-routing to on-chip networks. As evaluators, e.g. finite automatons, but also autonomous concepts like Learning Classifier Systems (LCS) can be used. The derivation of appropriate LCS rules, that restrict the system in a way, that application specific boundary conditions can be kept during run-time, is part of the optimization process.

An approach to detect errors during run-time, consists in the inclusion of failure rate registers at certain locations of a module together with error monitors or error correcting codes (ECC). In the registers the number of errors, that are detected by the monitors during a certain time interval, is stored. Based on these numbers, a reliability measure can be calculated by using the technique presented in [BerB06].

When the reliability measure takes an unacceptable value, actuators are triggered and/or the corresponding information is sent to the next higher block in the module hierarchy. If for example large number of errors are counted over multiple intervals, the evaluator possibly concludes that an advanced aging of the component is responsible for this. When this information is propagated in the system in an appropriate way, this may lead to the migration of tasks carried out by the operating system.

The issue, how the faults detected by the monitors on the chip can be propagated and aggregated during run-time, in order to make conclusions about the entire system and its subsystems respectively, has also to be dealt with during the system design process. In doing so, it has to be taken into account, that occurring errors are possibly not visible on system level due to a potential error masking.

With the help of an assumed error propagation in the model the effects of different faults on system level have to be examined. This, e.g. can be done by injecting errors into the system.

2.2 Autonomic operating systems

Real-Time operating systems like VxWorks or DREAMS [Boe03] provide fine-granular architectures which enable customized configuration of the OS to the requirements of the applications. The ability of customizing the OS services makes these operating systems to some extent applicable for resource-restricted architectures. However, these operating systems are not able to cope with dynamically changing system behavior and requirements of the applications as well as environmental changes. The MPSoC is composed of computing elements which are too restricted in terms of resources to each run a complete operating system providing the needed complexity and functionality. Therefore, the main idea of the proposed operating system is to distribute the services provided by it among the computing elements of the MPSoC. The set of services required by an application will be supplied by a cluster of the computing elements. A cluster of computing elements is defined as a subset of the computing elements of the MPSoC. Naturally, there are also some services which must be available at each node (i.e. computing element) in order to provide a minimum functionality. These fundamental services are, for example, communication and migration.

The distribution of the OS services among the nodes aims to efficiently exploit the restricted resources of the system. The operating system autonomously manages the distribution of the services depending on the current workload. It continuously optimizes an objective function. To realize this, it uses distributed optimization algorithms from the area of swarm intelligence. We will apply ant algorithms to enable the dynamic and self-optimizing distribution of the OS services among the MPSoC nodes. More concretely, it is realized employing the foraging behavior of ants: services are considered as food sources, the application requesting a particular service, as the formicary. The request itself is modeled as an ant that travels towards a particular service. According to its natural model, an ant representing a request message leaves behind pheromones on its route to the target service. The position of a service is optimized by migrating that service to that neighboring node with the highest pheromone value.

Given the possibility of node failure (e.g. incurred by external attacks), the operating system generates a number of replicas of each service and distributes them over the network of nodes. Combined with the property of continuous self-optimization of the distribution of services over the MPSoC, this redundancy of system services leads to the self-healing property of the operating system. This property includes the ability of the system to autonomously detect software and hardware failures and to trigger appropriate actions in order to preserve as much of the functionality of the entire system as possible and to provide graceful degradation.

The architecture of the autonomous operating system is composed of a functional layer and an autonomic layer. The functional layer includes the basic operating system functions: (1) Hardware abstraction services, encapsulating the lower-layer hardware. (2) Services for the communication of data, code, state and monitoring information. (3) Services for migration.

These basic operating system services constitute the so-called “ZeroOS” which provides a basis for the self-organizing operating system. Each node of the MPSoC executes one instance of the “ZeroOS.

The autonomic layer consists of elements that support the self-organization functions of the operating system. In particular, the autonomic elements realize the

operating system's self-healing capabilities employing a reflection service and a replication service.
The main responsibility of the reflection service is to monitor the system state and, if required, to invoke methods that adjust the system state with the objective of self-healing, self-optimizing or adjusting the quality of a service. The replication service is responsible for replicating services in order to increase redundancy, to improve the reliability and the self-healing capabilities of the system in the case of a failure. The services of the autonomic layer use the services provided by the functional layer.

2.3 Performance analysis

Changes in the FEs due to failures or reduced performance will affect the overall system performance. Any modification of a configuration in case of static faults and any recovery step in case of transient faults affects the overall system performance and can lead to system faults, e.g., overload situations, buffer over- or underflows, or missed deadlines in case of real-time systems. The result can be a system failure or, at least, reduced system quality. The system must take these effects into account and reroute communication or remap tasks.
Based on previous work in analyzing and optimizing heterogeneous embedded systems which where implemented in the tool, SymTA/S [BS1], an analysis function will be developed as a basis for optimization. First, the system shall be analyzed for the fault free case. That alone is so complicated for heterogeneous systems that it can only be approximated for realistic systems. Component data such as worst case run times and context switch cost are needed for that purpose as well as process activation and communication frequencies. They are considered part of the system description. Results of this analysis are global performance data, such as system response times, transient load situations and required buffer memory. In a second step, fault modeling is used to determine the effects of a fault on system performance. The fault model must include transient and static faults. Single transient faults are typically covered by retransmission or roll-back, i.e. extra run or communication time, and can be included in system timing. More complicated in analysis is the system response to frequent transient and static faults which are treated by a change in system configuration. Now, analysis not only capture the effects of faults but must also predict re-routing and remapping. For that purpose, analysis needs input from the operating system and the rerouting algorithms. To control computation time, the model used in prediction will be simplified and limited to single changes of configuration. Result of the second step will be a parametric fault model that provides the system performance for a given fault situation. For each potential fault that is given by the global design method, the influence on load, buffering or end-to-end timing can now be analyzed. That kind of system-level fault analysis is only possible because system performance analysis is extremely fast compared to simulation.
Once such a sophisticated fault model will be available, it can be used in system robustness optimization. The goal of robustness optimization is to minimize system sensitivity to changes, in this case to failures. In previous work [BS2], we have already shown that scheduling or traffic shaping can be effectively adapted to improve robustness. Such an improvement will be particularly helpful to mitigate the global influence of transient faults which appears necessary to control system design under the assumption of faults.

3. Component-level techniques

3.1 CPU Datapath

In order to detect and correct errors in a CPU datapath, we need to look at fault models first, which formally describe the nature of the errors under investigation. In [Avi04] a fault model classification was presented, classifying transient errors, timing errors, permanent errors, and design errors. Based on this fault models, we presented a study on existing fault tolerance techniques in [BouB06]. Each technique was assessed according to the class of errors it detects and the corresponding associated overheads. A key point in this study concerned IP-reuse, i.e. the capability to build a fault-tolerant CPU starting from actual existing non-fault-tolerant CPUs by adding cost-effective error monitoring and correction techniques.

In [BouZ06] we presented a new scheme for a self-healing CPU datapath with protection against transient and timing errors. Simulations proved that error correction was achieved with a penalty of only 2 clock cycles, no matter where the error occurs within the pipeline. Preliminary synthesis results showed a modest hardware overhead of 23% in a Xilinx Virtex-II Pro FPGA.
Another technique based on hardware and time redundancy called DIVA [Aus01] achieves high fault coverage (permanent, transient, timing and design errors) with low overheads and with the possibility of IP-reuse. The weakness of DIVA is that the checker unit is considered as fully reliable, which does not hold in deep submicron technologies. In [BouB06] we suggested a modified version of DIVA in which both the DIVA core and checker re-execute an errant instruction, so that the checker no longer needs to be reliable.
In future work, we want to extend the CPU datapath pipeline protection to permanent errors, due to long

term degradation effects, to design errors, and to multi-bit faults.

3.2 CPU Controlpath

Complementary and in addition to autonomous error handling in data paths, it is our goal to ensure and/or increase the reliability of control paths by suitable hardware and software measures. This will be achieved by autonomous control elements which recognize, evaluate, and correct errors in the program execution of processors as well as the control logic of hardware modules. Memory errors such as "soft errors" or specific local attacks on control logic units of CPUs are in our particular research focus. This contains the correct condition transfer as well as the correct condition storage. Methods are investigated which supervise the control path and in the case of an error, execute reversible and not-reversible measures, for example the forced transfer into "secure" states or, if possible, perform autonomic error corrections. These methods are provided by autonomous control elements which function as a bridge between the functional and the software layer by offering an interface for the operating system.

Reliable functioning of the control path inside CPUs and IP cores is essential. In control paths, two types of errors may occur: errors in the correct storage of the state and errors in the calculation of the correct next state. In both cases, the errors should be recognized and fixed at run time. Methods to monitor and to handle permanent and temporal errors in memory structures are well known, e.g., maskable memory regions or error correcting codes. These methods can be used for control paths of CPUs or as well as IP cores, if the state memory is organized like a RAM. Unfortunately, state of CPUs and IP cores are mainly stored in registers, which are distributed over the whole component rather than in memories.

Far fewer systematic approaches exist to detect and to cope with incorrect state transitions and achieve a correct functional action. Partly, the methods are too unspecified and ineffective for checking the correctness of state transitions. Methods using hardware redundancy, like Concurrent Error Detection (CED) and Triple mode redundancy (TMR) usually require a high amount of additional hardware, or, they are not taking benefit of the specific structure of the components which are monitored.

Future research has to concentrate on techniques which exploit specific properties and the structure of the control logic to be monitored. These techniques should guarantee a higher reliability at a lower hardware overhead considering the control path of a CPU. The actual state of execution of a program is, in general, given by the value of the program counter, the register values, as well as the stack. Usually, the next instruction is computed by incrementing the program counter, but also branches and jumps may occur. In order to determinate correct branch or jump destinations, a complex control logic must be implemented affecting many pipeline steps, which may be influenced by accidental sporadic or permanent errors or intended errors caused by a local attack in order to manipulate the program execution. Errors may also be caused by pure software effects like buffer overflows. Here, a wrong jump destination or a wrong return address from a subroutine can cause an execution of infiltrated code.

In MPSoCs, especially in embedded applications, the update rate of the software is low compared to a general purpose computer. Most program execution time of an MPSoC is consumed only on few specified subroutines. Now, these specific programs or subroutines can be analyzed at compile time for branches and jumps of the program flow. This information can be extracted and an internal control unit of the CPU can use this information to monitor the correct execution of a program. We intend to investigate new program path checking methods, to detect and compensate for hardware errors and software errors. In the case of an error, the correct program state should be restored. Also, the erroneous behavior is logged into failure rate registers (see Section 2.1) to notify the operating system.

The main tasks of our contribution are: (1) Concepts and models for error detection and compensation in CPU control paths. (2) Analysis and evaluation of cost overheads for supervisor units. (3) Integration of concepts and test together with autonomous data path units. (4) Demonstration: Finally, the integration and combination with other autonomic elements and the evaluation in our demonstrator platform with a LEON 2 processor will be demonstrated.

3.3 Communication resources

Increasing the reliability of the communication between the computing elements is as important as the reliability of the computing elements itself. In many MPSoCs, communication is already the bottleneck with respect to bandwidth, performance, power and reliability. Time-dependant dynamic variations in voltage and power density, electromagnetic inference and cross talk, synchronization failures, and soft-errors make the communication on the electrical level less and less reliable resulting in transient timing errors or even complete information loss during transmission.

A well known approach to tackle this problem is the use of static worst case design techniques. Examples for these techniques are shielding, special spacing rules and repeater insertion. However these techniques result in over dimensioned circuits and sys-

tems for the typical case, but worst case scenarios typically take place infrequently. Moreover all worst case scenarios have to be known at design time, i.e. the system can't react dynamically on new situations at run-time. The trend towards faster and low-power communication will further decrease the reliability at the physical and electrical level. Hence, correct by construction design techniques at the physical level will no longer be feasible. As a consequence, the physical layer has to be considered as unreliable digital link, where the probability of bit errors is small but not negligible. Furthermore, reliability can be traded-off for energy.

To solve this problem we have to use higher layers in the communication stack, i.e. data-link, transport and network layer, and the application layer, to improve the system reliability.

One of the most widely used techniques to improve reliability consists of adding some redundancy to the original information, for instance, by appropriate coding techniques. The data-link can be used to detect and correct errors with such coding techniques (this technique is exploited very successfully for many years in wireless communication systems in which the transmission channel is always unreliable). Several coding techniques are published: low power codes, crosstalk avoidance codes and error correction codes. It is often argued that these techniques are technology and implementation independent. However this is only partially true since the efficiency of a coding technique strongly depends on the underlying fault and bit error rate models which are up to the physical layer.

Another approach for reliable communication consists of self-calibrating data-link design techniques. Self calibrating circuits adjust their operating parameters to run-time actual conditions and allow a trade-off between energy, performance and reliability. Self calibrating and coding techniques can be combined to coding adaptation which consists of dynamically changing the coding strength at run-time dependent on the current context.

If the communication is implemented by a network-on-chip architecture we can apply stochastic communication on the network and transport layer to improve reliability. Such a paradigm is based on probabilistic flooding which is an effective fault-tolerant technique because if a path exists to destination, a message will almost certainly arrive.

Moreover we can exploit the application knowledge to improve the reliability. Many applications have some inherent fault-tolerance which can be exploited to partially tolerate errors in the communication.

Obviously, many techniques exist on the various communication layers to make the unreliable physical link more reliable from a system point of view. The challenge consists of finding the best trade-off of all these techniques with respect to performance, energy and reliability. This trade-off strongly depends on the underlying hardware platform and the application.

Conclusions

Future nanoelectronic systems will be increasingly sensible to technology parameter variations, natural radiation, and changing environmental conditions. The classical worst-case design paradigm is not feasible for these challenges. We propose to investigate Autonomic MPSoCs with the capability to detect and correct sporadic errors and to adapt themselves to changing conditions, in order to achieve reliable system behavior with unreliable components. In this paper, we presented our concept of Autonomic MPSoCs, the major problems to be solved, and some first approaches to be followed.

References

[Avi97] A. Avizienis: "Toward Systematic Design of Fault-Tolerant Systems", IEEE Computer, April 1997

[Avi04] A. Avizienis et al.: "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Trans. on Dependable and Secure Computing, 1(1) (2004)

[Avr01] D. Avreski et al.: "Fault-Tolerant Embedded Systems", IEEE Micro, Sept.-Oct. 2001

[BerB06] A. Bernauer et al.: "An Architecture for Runtime Evaluation of SoC Reliability", GI-Edition - Lecture Notes in Informatics, pages 177-185. Köllen Verlag, September 2006

[Boe03] C. Boeke, M. Goetz, T. Heimfarth, D. Kebbe, F. Rammig, S. Rips. (Re-)configurable Real-time Operating Systems and their Applications. IEEE International Workshop on Object-Oriented Real-Time Dependable Systems , 2003.

[BouB06] A. Bouajila et al.: "Error Detection Techniques Applicable in an Architecture Framework and Design Methodology for Autonomic SoCs", Biologically Inspired Cooperative Computing (BICC 2006), pages 107-113. Springer, August 2006

[BouZ06] A. Bouajila et al.: "Organic Computing at the System on Chip Level", VLSI-SoC, Nice, France, Oct. 2006

[BS1] R. Henia et al.: "System Level Performance Analysis - the SymTA/S Approach", IEE Proceedings Computers and Digital Techniques, 2005

[BS2] A. Hamann, et al.: „A Formal Approach to Robustness Maximization of Complex Heterogeneous Embedded Systems." CODES, Seoul, Korea, October 2006.