Microelectronics Reliability 80 (2018) 176–183

Contents lists available at ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel

Handling of transient and permanent faults in dynamically scheduled super-scalar processors

Felix Mühlbauer[a,*], Lukas Schröder[a], Mario Schölzel[a,b]

[a] University of Potsdam, Germany
[b] IHP, Frankfurt(Oder), Germany

ARTICLE INFO

Keywords:
Fault tolerance
Fail-safe
Dynamically scheduled processor

ABSTRACT

This article describes architectural extensions for a dynamically scheduled processor to enable three different operation modes, ranging from high-performance, to high-reliability. With minor extensions of the control path, the resources of the super-scalar data-path can be used either for high-performance execution, fail-safe-operation, or fault-tolerant-operation. Furthermore, the online error-correction capabilities are combined with reconfiguration techniques for permanent fault handling. This reconfiguration can take defective components out of operation permanently, and can be triggered on-demand during runtime, depending on the frequency of online corrected faults. A comprehensive fault simulation was carried out in order to evaluate hardware overhead, fault coverage and performance penalties of the proposed approach. Moreover, the impact of the permanent reconfiguration regarding the reliability and performance is investigated.

1. Introduction

Within the past decade dynamically scheduled processor architectures have taken root in the domain of embedded systems. For example, ARM-based processors are heavily used. High performance of these processors is typically achieved by instruction level parallelism via multiple functional units and supporting out-of-order and speculative execution. Moreover, the performance greatly benefited from the shrinking feature size in the CMOS technology that enabled the integration of more functional units. But manufacturing in nanoscale dimensions increases the probability for temporary faults affecting the device at runtime and aging effects causing permanent faults in such processors [1].

On the other hand, new application domains, like electronic control units for automated or assisted driving, increase the demand for high-performance processors, while at the same time very high reliability in some use cases is required. The demand for reliability could have different flavors: In some cases, a fail-safe mode is sufficient, if, in case of a detected error (transient or permanent one), the system can switch into a fail-safe state, where it cannot cause any harm and it is acceptable that the system stops to perform the desired function. In other cases, a fault-tolerant mode is required, if the safety-critical function must be provided reliable despite of occurred temporary or permanent faults.

The architecture of dynamically scheduled processors already offers mechanisms that can be used for fail-safe or fault-tolerant operation. For example, hardware-redundancy is inherently available as it is typically employed in fault-tolerant system design. But, also roll-back techniques are available which are typically used to recover from the speculative execution of miss-predicted branches.

Several approaches for integrating fault tolerance were presented during the recent years for dynamically scheduled processors. These techniques are either based on online error detection and correction, or on reconfiguration. Reconfiguration allows only for handling permanent faults, while online error handling could handle temporary and possibly permanent faults during runtime. Online fault handling of permanent faults is usually accomplished by fault masking and/or recomputation, both causing a performance loss, because operations are executed multiple times simultaneously, or a time consuming roll-back is required for error correction. For permanent faults, this performance loss may be reduced by reconfiguration means, where a defective component is excluded from being used during the execution. By this, at least a repeated error detection and recovery of permanent faults can be avoided at runtime. Although for both techniques several approaches were presented in the past, there is to the best knowledge of the authors no approach, where online error correction and reconfiguration are combined into a single dynamically scheduled processor architecture.

The contribution of this article is the extension of a dynamically scheduled processor to detect temporary and permanent faults at runtime and to recover from them, and the integration of a reconfiguration technique for handling permanent faults. This yields a processor

* Corresponding author.
E-mail addresses: muehlbauer@cs.uni-potsdam.de (F. Mühlbauer), luschroe@uni-potsdam.de (L. Schröder), schoelzel@ihp-microelectronics.com (M. Schölzel).

https://doi.org/10.1016/j.microrel.2017.11.021
Received 4 August 2017; Received in revised form 6 November 2017; Accepted 24 November 2017

architecture with the following features:

- The processor can switch dynamically between three different execution modes: fail-operational (errors are detected and corrected online), fail-safe (only online error detection), and high-performance (neither error detection nor recovery).
- Frequently detected errors can trigger the reconfiguration mechanism that can exclude defective components of the processor from being used during the execution.
- Information of dynamically detected errors is propagated to a diagnostic test routine, which is used during reconfiguration.

Because online error detection does not deliver detailed diagnostic information about the site of the fault, the reconfiguration mechanism is coupled with a diagnostic software-based self-test, such that the fault site can be localized with the same granularity as the reconfiguration mechanism works.

The reminder of this article is organized as follows. Section 2 reviews existing work regarding fault handling and reconfiguration mechanisms for dynamically scheduled processors. Section 3 describes the baseline architecture of our dynamically scheduled processor. In Sections 4—6 our offline and online fault handling extensions are introduced. Finally, Section 7 provides results regarding hardware overhead as well as performance penalties and fault coverage.

2. Related work

For tolerating faults always some kind of redundancy is necessary [2]. In dynamically scheduled architectures hardware redundancy is inherently available to achieve high-performance. A lot of research was done on managing this redundancy for fault tolerance purposes and can be grouped into two categories. First, graceful degradation: A faulty component is permanently disabled and operations are distributed to other available components. This resource reduction is accompanied by performance degradation. Such approaches are suitable for permanent faults. Second, concurrent execution: Operations are executed multiple times and errors could be detected or even masked. Not only transient faults could be handled online but also permanent faults in some cases. However, defective components remain active and permanent faults will repeatedly trigger the correction mechanism.

A very simple approach concerning graceful degradation in dynamically scheduled processors is to mark defective functional units as occupied [3,4]. In [5] this method was extended to array-like structures which are present in many control tables of dynamically scheduled processors, like reservation stations or the reorder buffer. On this more fine-grained level, single table entries could be disabled by declaring them as occupied. However, the fault diagnosis, i. e. the online localization of faults, is only discussed briefly. In [6] a diagnostic software-based self-test (SBST) that can detect defective components offline was proposed for that purpose.

In order to handle transient and permanent faults, which occur at runtime, the error checking must be done online, e.g. duplicating operations, distributing them to different functional units and compare the results afterwards. This can be done at different levels between hardware and software. For example, in [7] the simultaneous multithreading capabilities of superscalar processors are exploited for executing a thread twice. However, the error detection latency of software is usually high and recovery is difficult. This can be a problem especially for systems with real-time constraints.

For dynamically scheduled processors, concurrent error checking can also be done at instruction level. A first discussion of suitable techniques was done by Franklin [3]. In [4] he proposed the duplication of operations by the scheduler. Stall cycles during the execution of the program are used to re-execute already executed operations, which reduces the performance penalty. In [8] the duplication is used for error detection. For the duplicated operations two consecutive reorder buffer entries are reserved. This makes the comparison of the results easy, but increases the size of the reorder buffer. Nevertheless, these works do not propose any recovery scheme.

In [9], recovery is postponed to software level. In [10] and [11], beside error detection, also recovery from transient faults is provided by hardware. If a fault was detected the processor is rolled-back, like after a mis-predicted branch operation, and execution is continued by fetching the failed operation again. However, duplicates are not distributed to different functional units. Thus, permanent faults may not be detected at all and additionally the recovery could end up in an endless loop, detecting always one and the same permanent error.

Additionally, in all of these papers the fault detection capabilities are only investigated analytically and no evaluation was done. In our experiments it turned out that there are several signals even in protected components, which are critical and should be protected by other techniques.

3. Processor architecture

This section introduces the architecture of our dynamically scheduled processor, which was used for demonstrating the proposed fault handling approach. The architecture is based on the Tomasulo-Architecture with a reorder-buffer [12] and supports out-of-order and speculative execution (see Fig. 1). Our fault tolerance extensions (in color) are described in the next sections.

The processor is composed of a 5-stage pipeline. In the *fetch-stage*, operations are fetched from the program memory into the IQ (instruction queue) component. The instructions are loaded from the address determined by the PC-register and are stored in a ring buffer. This process is stalled if the buffer is full.

In the *issue-stage* the next instruction from the IQ is issued to a reservation station (RS). Each functional unit (FU) has a dedicated RS, which buffers several instructions until their source operands become available. These operands can be either taken from the register file (REG), or they are bypassed from the reorder buffer (ROB), when they become available there. The scheduler distributes one instruction per cycle from the IQ to a RS in a Round-Robin manner. Therefore a free RS entry and additionally a free reorder buffer entry must be available. The ROB entry is reserved to store the result of that instruction.

In the *execute-stage*, instructions are processed by the functional units (FU). All RS-FU-units operate independently, thus instructions get out of order here. If all operands of an instruction are available and the FU is free, then this instruction can be executed. The instruction is then forwarded from the RS and the corresponding RS entry is freed. Multiple cycles may be used for executing an instruction.

When the execution of an instruction is finished, then the result is stored into the reserved entry of the ROB during the *write-back-stage*. As mentioned above, results can be forwarded to waiting instructions in the reservation stations. Results from all FUs can be written to the reorder buffer simultaneously.

Finally, in the *commit-stage*, the reorder buffer commits instructions in the original program order. The buffer is organized as a ring buffer. In the issue-stage the entries have been reserved in the order of the original order of the instructions, thus the next entry which should be committed is always known. When this entry receives a result this instruction could be completed. Committing a data operation means to write the computed result into the register file. If necessary also a memory access (read or write) is performed. Committing a branch operation means to write the result into the PC. We have implemented a static branch prediction that is always *branch is not taken*. Thus, if a branch is taken all *administration tables* (instruction queue, reservation stations and reorder buffer) are cleared. Otherwise the speculative processing of the instructions stored in these tables was correct, and they can be processed further.