Low latency reconfiguration mechanism for fine-grained processor internal functional units

Segabinazzi Ferreira, Raphael; Nolte, Jörg

DOI
10.1109/LATW.2019.8704560

Publication Date
2019

Document Version
Accepted author manuscript

Published in:
2019 IEEE Latin American Test Symposium (LATS)

(Article begins on next page)

Low latency reconfiguration mechanism for fine-grained processor internal functional units

Raphael Segabinazzi Ferreira
Distributed and Operating Systems
Brandenburg University of Technology Cottbus-Senftenberg
Cottbus, Germany
R.SegabinazziFerreira@b-tu.de

Jörg Nolte
Distributed and Operating Systems
Brandenburg University of Technology Cottbus-Senftenberg
Cottbus, Germany
Joerg.Nolte@b-tu.de

***Abstract*—The strive for performance, low power consumption, and less chip area have been diminishing the reliability and the time to fault occurrences due to wear out of electronic devices. Recent research has shown that functional units within processors usually execute a different amount of operations when running programs. Therefore, these units present different individual wear out during their lifetime. Most existent schemes for reconfiguration of processors due to fault detection and other processor parameters are done at the level of cores which is a costly way to achieve redundancy. This paper presents a low latency (approximately 1 clock cycle) software controlled mechanism to reconfigure units within processor cores according to predefined parameters. Such reconfiguration capability delivers features like wear out balance of processor functional units, configuration of units according to the criticality of tasks running on an operating system and configurations to gain in performance (e.g. parallel execution) when possible. The focus of this paper is to show the implemented low latency reconfiguration mechanism and highlight its possible main features.**

***Keywords*—functional units, fine-grained, reconfiguration, low latency, software.**

I. Introduction

Electronic devices have been decreasing their reliability and lifetime due to technology aims like high performance, low power consumption, and less chip area.

However, recent works have been done in multiple areas of research to counter these drawbacks. Some of these techniques are applied in the lower levels of electronic systems [1] [2] [3], and others are software frameworks running on top of Operating Systems (OSs) [4] [5]. Yet other measures are also applied to operating systems resulting in partially fault tolerant OSs like MINIX3 [7], dOSEK [8], SAFERTOS [9] and ERIKA Enterprise [10]. At the same time, standards like ISO26262 and operating systems specifications (e.g. AUTOSAR [11] and OSEK/VDX [12]) were established to cope with critical systems which demand high levels of reliability, and a fault event must not result in harmful effects.

Looking into processors of electronic systems, works have proposed management techniques and redundancy in the level of cores [14] [16]. However, recent research has shown that internal units of processors (e.g. ALU, multipliers, shifters, dividers...) do not execute the same number of operations while running any kind of program [13]. Thus, each of these units presents different individual levels of wear out during their lifetime. This means that core level redundancy is costly.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722325.

This paper presents a low latency mechanism capable to configure and reconfigure processor internal Functional Units (FU). The aim of this mechanism is to enable an efficient configuration of hardware (HW) resources from software (SW) level routines. And, as a consequence, allow SW routines in OSs to perform optimized configurations for critical and non-critical applications.

This article exposes, in the following sections, some of the works related to this topic (II), the proposed reconfiguration mechanism (III) and its implementation details (IV), the preliminary results (V), the use cases and the integration possibilities for the implemented mechanism (VI) and, finally, a conclusion (VII).

II. Related Work

Recent work presented in [18] proposes an extension to a dynamically scheduled processor architecture in order to increase fault-tolerance against transient and permanent faults. The work relies on operation mode configuration: high-performance, fail-safe and fault-tolerant. The high-performance mode is already provided by multiple functional units available in the superscalar processor architecture. In addition, it uses these available HW resources to implement the other two operation modes. The fail safe mode is based on concurrent checking which executes the same instruction in two different units; the fault tolerant mode extends the fail-safe one performing majority voting on demand. Additionally, an extension to this work was proposed by the same authors in [19]. It proposed the use of error corrections codes, rollback and recovery approaches to protect processor components, which are not covered by the replication scheme, and critical signals against temporary and permanent faults.

The work proposed in [20] shows a health monitoring and management system infrastructure for Systems on Chip (SoCs) based in the IEEE 1687 standard [21]. It mainly uses the interface provided by the standard, the Internal JTAG (IJTAG), to exchange messages between HW instruments and software routines. However, due to the communication latency of the IJTAG interface, the authors implemented an additional asynchronous signal to trigger operating system interruptions in case of uncorrected errors.

Further research has been done by companies to improve the dependability of their processor architectures. As a result, processors with lock-step capabilities were released. This capability enables redundant execution of operations and comparison of results by the multiple cores of these designs. Related to the results comparison, there are approaches which perform it at every clock cycle [15] [16] and others periodically [17].

Finally, in [14], a common of the shelf (COTS) processor was extended to make it more fault tolerant with the Triple Core Lock-Step (TCLS) scheme. However, the control logic was built in such a way that it is possible to start the recovery process on demand by a software issuing an interrupt to the CPU design.

III. Reconfiguration Mechanism

To enable integration of hardware units and software routines, a mechanism to configure processor internal functional units was built and is being presented in this paper. Then an optimized way to configure HW resources can be used in the struggle for dependable execution of critical tasks running under an OS.

Fig. 1 shows the main implemented units of the mechanism and its context in the processor pipeline. The "Pre-Decoder" and the "Units Controller" block was designed to allow the reconfiguration of the functional units. These blocks will be explained further in detail in sections IV-B and IV-C respectively. Although these units were introduced in the processor pipeline, specific signals were added allowing the Pre-Decoder block to send requests to Units Controller. Additionally, new instructions and new registers were defined, respectively, to allow SW routines command HW reconfiguration actions, and to store data about events on functional units and their status.

The whole mechanism works as follows (Fig. 2):

- The reconfiguration command is stated in SW as a raw operation code (OpCode) and the compiler keeps it as it is after compilation.
- The OpCode is fetched from memory, and the Pre-Decoder block is responsible to recognize the reconfiguration commands, writes in the reconfiguration registers (explained in section IV-D) and, when necessary, forward the required configuration to the Units Controller block.
- The Units Controller receives these signals and configures the adequately Switches to connect and disconnect the functional units from the pipeline. Also, it configures the multiplexer to only forward the signal from the active FU or enable the voter to perform majority voting when required.
- Once the reconfiguration is finished the Pre-Decoder clears one predefined architecture register. So that, the SW routine can use this register to get feedback from HW to recognize when the reconfiguration is done.

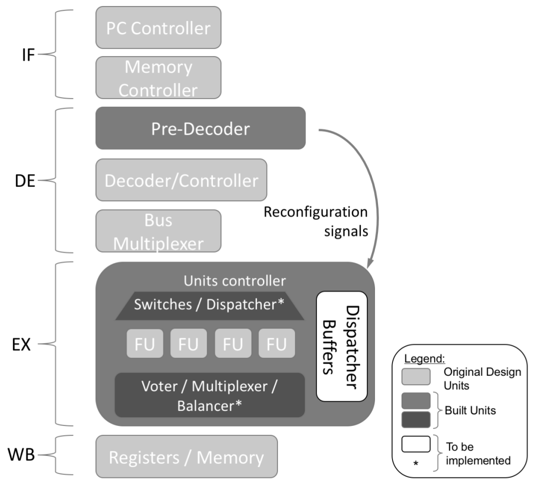

Fig. 1. General overview of the processor design and the reconfiguration mechanism blocks.

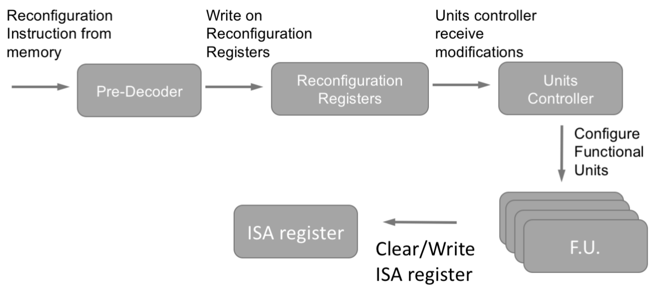
Reconfiguration Instruction from memory
Pre-Decoder
Write on Reconfiguration Registers
Reconfiguration Registers
Units controller receive modifications
Units Controller
Configure Functional Units
F.U.
Clear/Write ISA register
ISA register

Fig. 2. Reconfiguration actions performed by the mechanism.

A. Mechanism Operation Modes

Since critical and non-critical applications may run under the application scenario, as well as requirement for performance. Three different operation modes were foreseen. So, the mechanism can be used to configure FU in the following:

1) Generic Operation Mode: in this mode, the processor units are working as usual with no redundancy and no parallelism.

2) Dependable Operation Mode: this is the mode for dependable execution. Units are switched on to work on double or triple redundancy schemes. Thus, the Units Controller is responsible to connect the required number of units to the pipeline and perform the majority voting.

3) Performance Operation Mode: for future expansion, not implemented in the design by now. This mode is to target performance, it can mainly be used for applications which present mixed tasks with different criticality requirements. For example, a non-critical task can use multiple functional units to increase its performance with parallel execution.

IV. Mechanism Implementation Details

The following sections explain in detail each reconfiguration element and how it works.

A. Reconfiguration Commands

New commands were defined to be able to reconfigure and control HW fine-grained units from software. These commands are OpCodes explicitly stated in the software and expected by the Pre-Decoder block. As a requirement, these commands should not match any other OpCode from the already running Instruction Set Architecture (ISA).

Table I shows these commands and their operations. The first column represents the first two bytes of the command; it defines the OpCode Identifier which is used to identify it as a reconfiguration command by the Pre-Decoder. The second column shows the third byte as the operation command. And, finally, the third column shows the fourth byte which defines an index of a register when necessary.

The actions performed by each command are the following:

1) 0x55AF 0x00 - READ_FAULTS: read statistics about the functional unit correspondent to the register specified in the last byte of this command;

2) 0x55AF 0x01 - READ_UNITS_STATUS: read the status register (described in section IV-D);

3) 0x55A0 0x00 - OFF_UNIT: switch off the correspondent unit specified in the last byte of this command. When receiving this command, right after switching off the required unit the Units Controller will automatically switch on another spare unit to overcome the missing one;

4) 0x55A0 0x01 - ON_UNIT: switch on the specified unit;

5) 0x55A1 0x00 - OPMODE_GENERIC: switch the operation mode to the generic mode.

6) 0x55A1 0x01 - OPMODE_DEPENDABLA: switch to dependable operation mode.

7) 0x55A1 0x02 - OPMODE_PERFORMANCE: switch to performance operation mode.

As can be noticed these commands are not fully explored, so there is space left for easy extensions in the future.

TABLE I
OPCODE COMMANDS CREATED FOR SW CONTROL OF PROCESSOR FUNCTIONAL UNITS.

OpCode Identifier	Operation	Units Index Register
0x55AF	0x00 - READ_FAULTS	0x00 - 0xff
	0x01 - READ_UNITS_STATUS	Don't Care (D.C.)
0x55A0	0x00 - OFF_UNIT	0x00 - 0xff
	0x01 - ON_UNITS	0x00 - 0xff
0x55A1	0x00 - OPMODE_GENERIC	D.C.
	0x01 - OPMODE_DEPENDABLE	D.C.
	0x02 - OPMODE_PERFORMANCE	D.C.

B. Pre-Decoder

In order to modify as little as possible the original processor design, a combinational Pre-Decoder unit was built to decode the reconfiguration commands. It was introduced right before the original decoder of the design. This way to build this element has the advantage that only the Pre-Decoder unit needs to be prepared to receive and decode the reconfiguration commands OpCode.

Additionally, the evaluation of this unit becomes easier and its extension for future improvements and features is also facilitated due to this design choice.

Once this unit receives an instruction it will perform different actions in three different situations:

1) Normal ISA OpCode: in this situation it simply forwards the OpCode as it is to the next units in the pipeline. And because it was built using combinational logic it does not generate additional clock cycles for that.

2) Reconfiguration command to read data: when it receives one of the commands to read any data from the Reconfiguration Registers the Pre-Decoder generates and forwards an original instruction from the ISA to write the required data in the monitored original architecture register. Then it is possible to get this data from software by reading this register.

3) Reconfiguration command to perform action: when receiving one of the OpCodes to perform any reconfiguration action, the Pre-Decoder forward the signals to the Units Controller to tell it which action to perform. After that, the Pre-Decoder itself generates and forwards an instruction to clear the monitored register which is used by SW as operation feedback.

C. Units Controller

This controller is responsible to configure the functional units under his control according to the reconfiguration signals from Pre-Decoder. It means that according to these signals it will configure the Switches to connect the inputs and the multiplexer to connect the outputs of the units to pipeline signals, or disconnect them assigning High Impedance status - ”Z”.

In the Generic Operation Mode only one unit is connected to the pipeline, then the controller only needs to configure the input switches and the output multiplexer accordingly. On the other hand, for the dependable mode, three units need to be connected. Then, despite the switches configuration, this controller also needs to configure the output voter to perform majority voting of the results obtained from the connected units.

For the performance mode, this controller should be extended with appropriate buffers, dispatcher, and registers to enable parallel execution and re-ordering of instructions.

D. Registers

Two types of Reconfiguration Registers were defined:

1) Units Status Register: it is a register to keep track of units status. Each register bit represents a unit, and its status represents if it is in ”On” or ”Off” state. This register can be read and as well configured by software using the commands mentioned in section IV-A.

2) Individual units Stats Register: each functional unit has its own Stats Register which is used to keep track of events (e.g. faults). These registers can be read by software, but only configured by specific hardware mechanisms. For example, fault detection mechanisms can use these registers to record faults detected in monitored units and allow SW routines to keep track of this data.

E. Software Feedback

A register from the original processor architecture was reserved by the software routines to keep track of the results and data about the hardware reconfiguration actions. Thus, once the software sends one of the reconfiguration commands it waits until it gets the results on this reserved register.

V. Results

The described mechanism was implemented over the Plasma processor, which is a synthesizable 32-bits RISC microprocessor design. It is able to execute all MIPS I(TM) user mode instructions except unaligned load and store operations [22]. This design and the reconfiguration mechanism were synthesized, implemented and simulated using Xilinx development tools.

Also, as the first target of this implementation, the functional unit taken into consideration was the Arithmetical Logic Units (ALU), so it was replicated and placed under control of the Units Controller.

A computer simulation was done using the built design to evaluate the latency of the reconfiguration. As a result, the mechanism described above was able to perform a low latency reconfiguration of the functional unit of the processor design. Fig. 3 shows the right moment of the simulation when one ALU is switched off and another one is switched on. The wave forms described by a_in, b_in, $alu\_function$ and c_alu are the input signals coming from the pipeline arriving at the Units Controller. The a_in1, b_in1, $alu\_function1$ and c_alu1 correspond to ALU1 signals and a_in2, b_in2, $alu\_function2$ and c_alu2 to ALU2. The signals clk and $opcode\_out$ represent, respectively, the clock signal and the current OpCode just fetched from memory. Finally, the signals $opcode\_in\_p$ and $opcode\_out\_p$ are, respectively, the incoming and the outgoing OpCode from the Pre-Decoder block.

As can be noticed in Fig. 3, just after fetching a reconfiguration command from memory (0x55a00000 - OFF_UNIT) which can be found in the $opcode\_out$ and $opcode\_in\_p$ signal, the ALU1 input and output signals were switched to high impedance state "Z", and the ALU2 ones switched from "Z" to working state within the period of **only one clock cycle**. It means that the actions showed in Fig. 2 were performed within this measured latency.

To evaluate the hardware overhead generated by the mechanism the design was synthesized and mapped for a Xilinx FPGA device XC7Z020-CLG484-1 available in the Zynq^{TM}-7000 SoC. Table II shows the usage of FPGA primitives for the following: an original central processing unit (CPU) of the Plasma design (one core), an original Plasma ALU, and the implemented Pre-Decoder and Units Controller. Table III shows the overhead percentage of the ALU, the Pre-Decoder and the Units Controller when compared to the original Plasma CPU.

As can be noticed the biggest introduced element added an overhead of no more than 13%. When comparing to a full core level redundancy, a work based on an ARM processor showed a control logic overhead of 18% compared to a Cortex-R5 CPU, despite the overhead incurred by duplicating/triplicating the whole core itself [14].

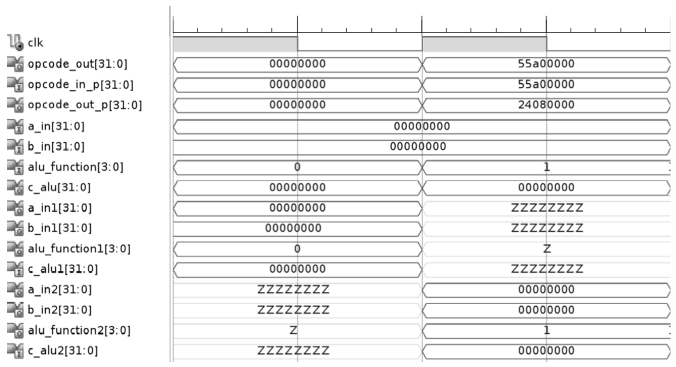

Fig. 3. Signals overview of the reconfiguration performed by the implemented mechanism in the simulated environment.

TABLE II
FPGA primitives Usage by Plasma CPU and Individual Units

FPGA Primitive	CPU Plasma (Original Design)	Plasma ALU	Pre-decoder	Units Controller
Slice LUTs	1815	276	72	270
Slice Registers	273	0	51	0
Muxes	47	0	0	0
Total	**2135**	**276**	**123**	**270**

TABLE III
Primitive usage percentage related to overall original Plasma design

FPGA Primitive	Plasma ALU	Pre-decoder	Units Controller
Slice LUTs	15,2%	4,0%	14,9%
Slice Registers	0%	18,7%	0%
Muxes	0%	0%	0%
Overhaed Total	**12,9%**	**5,8%**	**12,6%**

VI. Use cases and Integration Possibilities

According to the results shown in section V, the mechanism was able to perform a low-latency functional units reconfiguration. Therefore, this mechanism becomes suitable for real-time systems, and real-time OSs may use this mechanism to add HW controllability to their routines. Thereby, features like the ones mentioned below emerge.

A. Criticality Aware execution Path Configuration

It is possible to extend OS functions assigning criticality levels to threads. Then, the OS scheduler can use this information to program the processor design to use more reliable functional units when necessary. Moreover, it can switch ON fault tolerant mechanisms like TMR on individual units. The scheduler only needs to use the reconfiguration commands described in this work to perform this operation, which becomes possible because of the low latency of the mechanism.

The OS can use the reliability calculations and measurements used in [5] to get units' reliability estimations and, according to this data, perform units' reconfiguration. Additionally, numbers about detected faults are available using the READ_FAULTS command.

B. Units usage balancing

As soon it is possible to keep track the usage of functional units, a routine could be created and added to the OS to perform usage balancing of HW resources in fine-grained mode.

C. Parallelism for performance

A mode to target performance using parallelism becomes possible due to the spare units expected in this mechanism. So it can be another feature of the OS scheduler to find out that the upcoming task does not have criticality requirements but aims for performance. So the OS can use the reconfiguration commands to switch between the operation modes.

D. Application specific configurations

The low latency of the mechanism enabled the capability of units configuration according to application targets. Since different algorithms can be created on the higher levels of electronic systems, possibilities like wear out aware algorithms, execution of operations by specific units and other ones arise.

VII. Conclusion

This paper presented a mechanism to reconfigure fine-grained processor internal functional units. As exposed in the results section (V), the mechanism was able to effectively reconfigure the execution path with the desired unit within a latency of one clock cycle. Due to the low latency, this mechanism becomes suitable for real-time systems and can be useful for run-time configuration of HW resources (in fine-grained mode) targeting a reliable execution path.

Additionally, the hardware overhead incurred by the implemented units (Pre-Decoder and Units Controller) was less than 13% of the Plasma CPU design each. This overhead is noticeable lower when compared to core level redundancy schemes.

References

[1] S. Mitra, N. Seifert, M. Zhang, Q. Shi, K.S. Kim, "Robust system design with built-in soft-error resilience", Computer, vol. 38, issue: 2, pp. 43-52, Feb. 2005.

[2] S. Mitra, M. Zhang, N. Seifert, T. Mak, K. S. Kim, "Soft Error Resilient System Design through Error Correction", IFIP International Conference on Very Large Scale Integration, pp. 332-337, Oct. 2006.

[3] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. M. Harris, D. Blaauw, D. Sylvester, ""Bubble Razor: Eliminating Timing Margins in an ARM Cortex-M3 Processor in 45 nm CMOS Using Architecturally Independent Error Detection and Correction", IEEE Journal of Solid- State Circuits, vol. 48, Issue. 1, pp. 66-81, Jan. 2013.

[4] A. Baldassari, C. Bolchini, A. Miele, "A dynamic reliability management framework for heterogeneous multicore systems", IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Oct. 2017.

[5] P. Mercati, F. Paterna, A. Bartolini, L. Benini, T. Š. Rosing, "WARM: Workload-Aware Reliability Management in Linux/Android", IEEE Transactions on Compute-Aided Design of Integrated Circuits and Systems, vol. 36, issue 9, pp. 1557-1570, Sept. 2017.

[6] J. N. Herder, H. Bos, B. Gras, P. Homburg, A. S. Tanenbaum, "MINIX 3: a highly reliable, self-repairing operating system", ACM SIGOPS Operating Systems Review, vol. 40, issue 3, pp. 80-89, Jul. 2006.

[7] MINIX 3. http://www.minix3.org/, visited November 12^{th}, 2018.

[8] M. Hoffmann, F. Lukas, C. Dietrich, D. Lohmann, "dOSEK: the design and implementation of a dependability-oriented static embedded kernel", IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 256-270, April 2015.

[9] High Integrity Systems - SAFERTOS. https://www.highintegritysystems.com/safertos/, visited November 12^{th}, 2018.

[10] Erika Enterprise RTOS v3. http://www.erika-enterprise.com/, visited November 12^{th}, 2018.

[11] AUTOSAR Specification of operating system(version4.3.1). Technical report, Automotive Open System Architecture GbR, Dez. 2017.

[12] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, Feb. 2005.

[13] M. A. Skitsas, C. A. Nicopoulos, M. K. Michael, "DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective- Testing in Multi-/Many-Core Microprocessors", IEEE Transactions on Computers, vol. 65, issue: 5, pp. 1453-1466, May 2016.

[14] X. Iturbe, B. Venu, E. Ozer, S. Das, "A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications", 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop, pp. 246-149, Jun. 2016.

[15] ARM Ltd., "Cortex-R5 and Cortex-R5F. Technical Reference Manual", 2011.

[16] Infineon Technologies AG - 32-bit TriCoreTM Microcontroller. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/, visited November 30^{th}, 2018.

[17] S. Resch, A. Steininger and C. Scherrer, "Software Composability and Mixed Criticality for Triple Modular Redundant Architectures", Proc. Of the International Conference on Computer Safety, Reliability and Security, Jul. 2013.

[18] F. Mühlbauer, L. Schrör, M. Schölzel, "On Hardware-based Fault-Handling in Dynamically Scheduled Processors", 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), pp. 201-206, 2017.

[19] F. Mühlbauer, L. Schrör, M. Schölzel, "A Fault Tolerant Dynamically Scheduled Processor with Partial Permanent Fault Handling", IEEE 19th Latin-American Test Symposium (LATS), 2018.

[20] K. Shibin et al., "Health Management for Self-Aware SoCs Based on IEEE 1687 Infrastructure", IEEE Des. Test vol. 34, no. 6, pp. 27-35, 2017.

[21] IEEE Standard for Access and Control of Instrumentation Embedded Within a Semiconductor Device, 2014.

[22] Plasma - most MIPS I(TM) Overview. https://opencores.org/projects/plasma, visited on November 15^{th}, 2018.