Architecture-Level Fault-Tolerance for Biomedical Implants

Robert M. Seepers*†, Christos Strydis*†, Georgi N. Gaydadjiev‡
*Dept. of Neuroscience, Erasmus MC
Dr. Molewaterplein 50, 3015 GE Rotterdam, The Netherlands
{r.seepers,c.strydis}@erasmusmc.nl
†Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
‡Department of Computer Science and Engineering
Chalmers University of Technology, Rannvagen 6, Goteburg, Sweden
georgig@chalmers.se

***Abstract*—In this paper, we describe the design and implementation of a new fault-tolerant RISC-processor architecture suitable for a design framework targeting biomedical implants. The design targets both soft and hard faults and is original in efficiently combining as well as enhancing classic fault-tolerance techniques. The proposed architecture allows run-time trade-offs between performance and fault tolerance by means of instruction-level configurability. The system design is synthesized for UMC 90nm CMOS standard-process and is evaluated in terms of fault coverage, area, average power consumption, total energy consumption and performance for various duplication policies and test-sequence schedules. It is shown that area and power overheads of approximately 25% and 32%, respectively, are required to implement our techniques on the baseline processor. The major overheads of the proposed architecture are performance (up to 107%) and energy consumption (up to 157%). It is observed that the average power consumption is often reduced when a higher degree of fault tolerance is targeted. It is shown that test sequences can effectively be scheduled during the available program stalls and that nearly all soft faults are tolerated by using instruction duplication. The main advantages of the proposed architecture are the high portability of the introduced architecture-level fault-tolerance techniques, the flexibility in trading processor overheads for required fault-tolerance degree as well as affordable area and power consumption overheads.**

I. Introduction

The first pacemaker, an external motor with electric wires going to the patient's heart to keep it beating at a steady pace, was revolutionary in the 1950's. Thanks to technological advances, pacemakers or, generally, Implantable Medical Devices (IMDs) these days are not only fully implanted into the body, but they also gather data for analysis, have a long lifetime and can even be configured to the patient's specific needs during a visit to a cardiologist.

Traditionally, these devices have been designed from scratch in order to satisfy the tight constraints of power and energy consumption. We believe technology has now advanced enough to move away from this approach and implement a design framework targeting medical implants. Such a framework consists of a number of basic components such as sensors, actuators, batteries and processors, which can be tailored very easily to a particular medical application. Advantages of using such a framework are numerous, including a lower development-risk factor, reduced design time and as a result lower development costs. Creating such a framework is the main goal of the SiMS project [1] and, in this paper, we focus on one of the main components: the SiMS processor.

Medical implants are tightly constrained in terms of area, power consumption and energy consumption, yet, being safety-critical systems, require a high degree of fault tolerance. Common examples of fault tolerance include Triple-Modular Redundancy (TMR) [2], self-checking circuits [3], built-in self-test [4] and software techniques [5][6][7], among many others. However, many of these techniques are not suitable for a design framework targeting biomedical implants. For example, the average power consumption is tripled in triple-modular redundancy, which is not viable for biomedical implants as high power dissipation is related to various medical conditions, such as blood vessel dilation. Other techniques, such as built-in self-test, are not able to detect all soft faults, which are the main type of faults occurring in digital systems nowadays [2]. Finally, some of the techniques, such as self-checking circuits [3], are too specific to be part of a design framework. Generic techniques are much more suitable for a design framework, as they allow easy design tailoring to a particular application. It is concluded that a number of commonly used fault-tolerance techniques are not in tune with the demands of a design framework targeting biomedical implants. As such, it is our desire to propose an architecture using techniques which are well-suited for such a framework.

In order to provide generic yet highly effective fault-tolerance techniques, we are interested in using architecture-level techniques. While fault tolerance is widely researched, not much research has been conducted into fault-tolerance techniques which optimize specifically for the constraints set in the biomedical environment. Therefore, the focus of this work is on the design of an architecture using fault-tolerance techniques at the architectural level. This design

is then implemented and evaluated, in order to estimate the viability of the architecture for the intended design framework.

The main contributions of this paper are:

- A new architecture suitable for a range of biomedical implant-applications, with high soft- and hard-fault coverage yet affordable power and area overheads;
- An efficient implementation and synthesis-based evaluation of existing architectural level fault-tolerance techniques. All employed techniques are portable across different architectures and are suitable for architectures targeting (ultra-)low-power consumption;
- Combining and improving known fault-tolerance techniques which, resultingly, allows for a straight-forward, runtime trade-off between power consumption, performance and enhanced fault tolerance.

This paper is structured as follows. First, we present background information in Section II and related work in Section III. In Section IV we present the implementation of our fault-tolerant architecture. Section V contains the experimental results and reports on fault coverage, performance, area, power and energy consumption along with various compound metrics. Finally, overall conclusions are drawn in Section VI.

II. Background

In the following Sections, we will briefly present the architecture on which our fault-tolerance techniques are implemented, as well as the targeted fault models.

A. The Baseline architecture: SISC Baseline

The majority of our work consists of adding fault tolerance to the Baseline Smart-Implant Security Core (SISC) [13]. The SISC Baseline is designed for low-power consumption and, thus, has no advanced architectural features such as a dedicated multiplier, divider or forwarding unit. The main characteristics of the Baseline architecture are:

- 5-stage RISC processor, with an in-order IF / DE / EX / MEM / WB pipeline;
- 16-bit ISA with 24 instructions;
- 16 32-bit entry register file;
- Stall-detection unit which inserts stalls in the program execution at runtime, i.e. no nops are required in the application binary; and
- Separate 16-kB (16-bit wide) Instruction Memory (IM) and a 16-kB (32-bit wide) Data Memory (DM).

B. Fault models

Our fault-tolerant architecture consists of one technique targeting soft (temporary) faults and another targeting hard (permanent) faults (Section IV) and, as such, two fault models are assumed: a transient-fault model for soft faults, and a behavioral-fault model for hard faults. In both fault models we assume the occurrence of only a single fault in the system at any given time. The reason for this assumption is that, statistically, most faults are single in nature and fault-tolerance mechanisms which target single faults cover roughly 99.6% of all cases with multiple faults [14].

Soft faults are modeled by a temporary upset of a signal, which appears and disappears within time ΔT. For the behavior of these faults, we make the following assumptions:

1) Only one signal is faulty at any given time (single-fault assumption). Note that we do allow multiple faults on a single signal if the signal consists of multiple bits;
2) The faults are well-behaved, i.e. during execution time the fault is permanent for one cycle and the signal is free of faults outside this interval (ΔT is one cycle); and
3) The faults are signal independent, i.e. the fault is not depending on the signal's current value.

We assume that the result of an operation will not corrupt between passing our error-detection unit (EDU) and the next use of this result. For example, the result of the beqz instruction (taken or not taken) will not corrupt between the EDU and the control unit. While it is possible this may occur, the fan-in, fan-out and the size of the circuits which follow the EDU are small compared to the rest of the architecture and, as such, we currently neglect their contribution. For hard faults, we model the faults with a behavioral fault model. In this model, the component-level design of the processor is used where components or signals are modeled to either fail or not. While single stuck-at-fault modeling is more common and accurate for fault modeling and test-pattern generation, it requires gate-level details. As we strive to use architecture-level techniques which are independent of the implementation, we use a behavioral fault model.

III. Related Work

For our target architecture, there is a sufficient amount of time slack available [13] to introduce fault-tolerance mechanisms using time redundancy. The key advantage of exploiting time redundancy is that while the execution time is increased, the increase in average power consumption is lower compared to e.g., hardware or information redundancy.

Error detection using duplicated instructions (EDDI) [6] is a purely software-based technique for adding fault tolerance using time redundancy. Variables are kept in duplicate registers and a comparison between these results indicates if an error has occurred. The technique detects soft faults in arithmetic and memory operations, but lacks support for control-flow operations, error correction, hard faults and memory protection. In this work, we strive to add support for control-flow operations, error correction and hard faults. In addition, we strive to have the design configurable in order to make it more suitable for a design framework.

1) Soft Faults: Control-flow fault tolerance can be added by signature checking and preemptive checking. Using signatures to prevent control-flow errors has been used widely [3][7][15]. Static signatures, which summarize the information of e.g., the opcodes in a basic block, are added to the program by the compiler and are compared to a runtime-calculated signature. Signature-based techniques may [3] or may not [7] be supported by additional hardware and can also be used to prevent data-flow errors [15]. A preemptive technique is proposed in [10]. Control-flow instructions are duplicated and,

before a jump is taken, the target addresses of both operations are compared by custom hardware. If the target addresses are not the same, an error flag is raised. Both methods for error detection have minimal area and performance overheads.

2) Hard Faults: Hard-fault detection can be done by using structural or functional test sequences[1], among others. The result of a test sequence is compared to a value known at compile time in order to determine if the runtime value is different from the static value, i.e. if an error has occurred. Functional tests have the advantage of being more portable, while structural tests result in shorter tests with higher fault coverage [5]. The disadvantage of functional testing is that not all components may be functionally testable and, in some cases, the number of functionally untestable components may be large [16]. Test sequences may replace nops in a program, lowering the performance penalty [4].

3) Configurability: Offering a configurable trade-off between the degree of fault tolerance and overheads, which is highly desirable in an architecture for a design framework, can be done using instruction-level configurability [9]. A flag may be set for each instruction, signaling the degree of fault tolerance required. Alternatively, a custom instruction, which sets the required degree of fault tolerance for a block of code, can be used. The former method has the advantage of specifity, while the second option promises to be more generic.

IV. Implementation

By combining some of the techniques discussed in the previous Section in an efficient way, a high degree of soft- and hardware fault tolerance can be achieved while minimizing overheads due to resource sharing. We have both improved and extended most of the employed techniques. From an organizational point of view, the design is split in two techniques: one to detect and correct soft faults and one to detect hard faults, both of which will be discussed in the following Sections.

A. Fault-Tolerant Design for Soft Faults

Our method for providing fault tolerance to soft faults builds on the techniques presented in [6][10][9]. We first describe the general technique followed by the implementation details.

Instructions for which fault tolerance is required are duplicated at runtime and the original and duplicated instruction are executed in sequence. The result of the original instruction is stored in a dedicated register and, if the result of the original and the duplicated instruction are the same, the result is committed. If the results differ, the pipeline is flushed and the error is corrected by refetching the original instruction. We allow a trade-off to be made between fault-tolerance and performance by storing the duplication mode, i.e. the set of instructions which has to be duplicated, in a user-programmable dedicated register. An example is given in Fig. 1, where all control-flow instructions are made fault-tolerant and, therefore, instruction A is duplicated. Additionally, we

[1]In structural testing, the exact specifications of the micro-architecture are known, whereas the micro-architecture is modeled as a black box in functional testing.

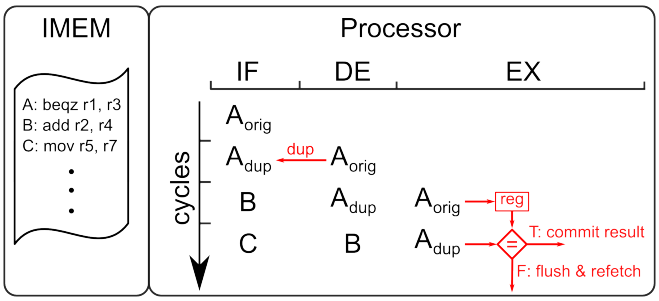

Fig. 1. Example of soft-fault tolerance through instruction duplication.

have added additional fault-tolerance mechanisms to prevent opcode and Program-Counter (PC) corruption. All in all, the additions we have made to make our architecture soft-fault tolerant are: 1) Instruction duplication, 2) Error detection, 3) Error correction, 4) Configurability, 5) Instruction-word parity bit and 6) a shadow PC. In the following text, we will discuss the details and contributions of each addition.

1) Instruction Duplication: Instruction duplication is done by the processor at run-time rather than at compile-time, for a number of reasons: First and foremost, doing duplication at runtime allows the degree of fault tolerance to be set dynamically. This means that the set of instructions which has to be made fault tolerant may not only be a function of the application, but also a function of run-time parameters such as the remaining battery lifetime. Secondly, the timing overhead of refetching the same instruction in case it needs to be duplicated is expected to be small or non-existent as it only requires a comparison between the decoded opcode and the degree of fault tolerance (explained below). Finally, as previously stated in Section II-A, a stall-detection unit is already present in the SISC Baseline processor, which is crucial when doing duplication at runtime as it prevents data-dependencies to be violated.

Configuration of the runtime duplication is implemented by comparing the decoded opcode to the set duplication mode. This duplication mode is stored in a dedicated register and can be set by the application using a new instruction *ft_set #*. Runtime duplication has originally been proposed in [8]. However, our improvement is the flexibility to make the trade-off between fault tolerance and overheads at runtime.

2) Error Detection: In the target processor a total of six results are committed. For each of these results and the opcode of the fetched instruction, we add a register-comparator block. The result of an original instruction is stored in the register and is compared to the result of the duplicated instruction in the next cycle. If the results are identical, it is safe to commit this result; if not, an error signal is sent to the error-correction unit. Two adjacent pipeline stages, both of which are executing an instruction which requires fault tolerance, can never both execute a duplicated or original instruction at the same time. As such, these stages can share hardware resources.

The advantages of performing error detection using these register-comparator blocks over a software-only implementa-

tion are as follows: First, by adding extra hardware for the storing and comparing of each result, the performance drop of the additional compare instruction in software-only techniques is not required. Moreover, hardware additions are required to preemptively detect errors in control-flow instructions, as demonstrated in [10]. Secondly, as the register file is not used for holding the intermediate values, the register pressure is not increased. Thus, the scheduler puts the same effort scheduling the instructions and excessive register spilling is avoided. Thirdly, having registers dedicated to a particular stage allows the registers to have a custom size in order to hold the expected result of a given size, reducing the required area. For example, the branch result is only 1 bit and storing this in a 32-bit register would, therefore, be a waste of resources. Furthermore, using additional hardware for error detection allows us to execute the same instruction twice in a row, which will be shown to have a positive side-effect of reducing average power consumption in Section V. Using these register-comparator structures is similar to what has been done in [10] for the target-address of control-flow operations. However, we apply the same technique to all stages which commit results.

3) Error Correction: Error correction is done by refetching the instruction of which the result has corrupted and flushing the pipeline. Instruction refetching is facilitated by buffering the instruction's address throughout the pipeline. When multiple results have corrupted, i.e. multiple error signals are set, priority is given to refetch the instruction which first entered the pipeline. Error correction by refetching is an improvement over the works in [10] where only detection is employed.

4) Configurability: As the processor is designed to be part of a framework, customization is an important factor of the design, including fault tolerance. By allowing an application developer to make a trade-off between fault tolerance and performance, as presented in [9], the processor can easily be adapted to many applications. It should be noted that such applications also exist in the biomedical field. For example, more performance may be required for image or sound processing (artificial eye or ear) in which temporal error in the processed data might be acceptable, while other applications (artificial pancreas, data encryption) may require more fault tolerance due to the application criticality.

Configuring the degree of fault tolerance can be done using a new instruction, *ft_set #*, which saves the degree of fault tolerance in a special register used by the duplication unit. This is an implementation of one of the two ways of setting the degree of fault tolerance proposed in [9]. Using this instruction, we have implemented four duplication policies: "none", "CF instr.", "full CF instr." and "full instr.". In "CF instr.", the duplication mode is set to duplicate control-flow instructions, i.e. only the control-flow instructions are duplicated. In "full CF instr.", the duplication mode is set to duplicate control-flow instructions and switches to full duplication when branching conditions are evaluated. Both these modes are intended to reduce the performance overhead of "full instr." or supporting other (software) fault-tolerance techniques.

5) Parity Bit: One crucial limitation in [10] is the assumption that the original instruction will never corrupt. Indeed, for our design described so far, this limitation also exists. If the opcode of an instruction which is intended to be duplicated corrupts, runtime-duplication would not take place. To tackle this limitation we add a parity bit in each instruction word that summarizes the opcode of that instruction. This parity bit is stored in the instruction memory and is compared to the parity which is computed at runtime.

6) Shadow PC: While the aforementioned additions guard against corruption of an instruction or its results (including control-flow operations), the PC is not covered by this method. As a fault in the PC will, without a doubt, have a high chance of causing errors, we add a "shadow" PC to the processor. This shadow PC mirrors the main PC and both are constantly compared. If an error occurs, a flag is raised and program execution is halted. The idea of using a shadow PC is an implementation of what has been proposed in [10].

B. Fault-Tolerant Design for Hard Faults

While soft faults are by far the most common faults [2] and are expected to become worse in the coming years [17], covering hard faults is also required due to the safety-critical nature of biomedical implants. To guard against hard faults, an approach is taken using a combination of the methodologies presented in [5], [4] and [12]. We run a sequence of instructions native to the ISA in order to generate an expected value in a register. By comparing this result to a value known at compile-time, we can determine if part of the processor has succumbed to any hard faults.

In our implementation, these test sequences replace nops in the original program, as proposed in [4]. This is very effective for programs compiled for our architecture, as many nops exist in the program static binary due to the inherent data-dependencies in these programs (e.g. 5,219 out of 14,606 instructions in one of our benchmarks) and the dependencies occurring due to our simplistic, low-power architecture. Note that while this is a high percentage (35.7%) in the static code, most nops appear outside the computationally intensive parts of the program at run-time; for example, loading variables or retrieving arguments from the stack after a function call. As such, the processor is *not* truly stalling for 35.7% of the execution time. As hard faults occur much less frequent than soft faults [2], it is not required to test for hard faults continuously. Therefore, it is acceptable if a sequence is only executed at e.g., the start of a function.

We run a sequence of instructions, native to the architecture, and have the result of this sequence be compared to a known value. Consider, for example, the following code:

```
mov r1, r0
bneqz r1, #trap
```

In this code, the program will be redirected to the trap address in case r1 is not 0. This can be due to a number of reasons, e.g. a stuck-at 1 fault on r1, on r0, etc.

Since we are interested in architecture-level techniques for fault tolerance, we model our processor as a gray box using

component-level detail, and apply functional testing. More structured tests could be employed by using gate-level details, resulting in smaller test sizes with a higher coverage [12]. The method we have devised to design the test sequences is the same as in [5] and is as follows:

1) The activated datapath is determined for each instruction;
2) For each of the instructions, the *controllability* and *observability* of the signals in the processor is determined;
3) Out of all combinations, a test sequence is generated containing the fewest possible instructions to observe an expected stuck-at fault, and determine its fault coverage;
4) Step (3) is repeated until all testable paths in the RTL-schematics of the processor are covered.

In general, the instruction memory is larger than required by the target application and, therefore, scheduling test sequences can be done without increasing the size of the instruction memory, i.e. no hardware overhead is required. If the instruction memory is not large enough, i.e. the fault coverage of the scheduled test sequences is not sufficient, increasing the memory size is always possible for additional cost.

A major disadvantage of using functional testing is, among others, that, while most (simple) units can be tested quite well, it is simply not feasible to test more complex units, e.g., ALU and shifter units, as every input pattern has to be applied in order to guarantee functional correctness. We have created functional test sequences which, if lower-level details become available, can easily be converted to structural test sequences. By inspecting our processor we have found that all paths, with the exception of the branch target and branch-result paths, are functionally testable. In order to test these functionally untestable paths, we have added custom hardware and one custom instruction. This custom instruction (br_tst #) allows us to directly operate on the untestable paths.

V. Experimental Results

In this Section, we first present our experimental setup for evaluating the architecture, after which we present our experimental results and evaluate our architecture accordingly.

A. Experimental Setup

The experimental setup consists of measuring the fault coverage, performance, processor area, average power consumption and energy consumption. First, we will present our processor designs, after which we present our method to determine these metrics.

1) Processor Designs: We evaluate three designs with increasing levels of fault coverage as shown in Table I. The following acronyms are used: *I)* BL: BaseLine, *II)* DUP: instruction DUPlication, *III)* PCBR: shadow PC and BR_tst instruction and *IV)* TS: Test Sequences. In the "BL/DUP" design, *most soft faults* are tolerated by the instruction duplication technique. *Complete soft-fault coverage* is achieved in the "BL/DUP/PCBR" design, wherein the shadow PC and the br_tst instruction are added. Note that in this design the br_tst instruction has no purpose, as the instruction is never scheduled. Finally, hard-fault detection is added by test-sequence scheduling in the "BL/DUP/PCBR/TS" design. Since we cannot accurately measure the fault coverage of our test sequences (Section V-A3), we choose to double the instruction memory size for this case. This allows us to replace all nops in the program and provide insight in the worst-case performance, area, power and energy consumption. We choose to only replace nop instructions with test sequences to demonstrate that hard-fault coverage can be added at a 0% performance overhead.

TABLE I
Processor designs

Name	Coverage			Resources		
	Soft (no PC)	Soft (PC)	Hard	Instr. Dup	Shadow PC, br_tst	Mem. Size +100%
Baseline	-	-	-	-	-	-
BL/DUP	x	-	-	x	-	-
BL/DUP/PCBR	x	x	-	x	x	-
BL/DUP/PCBR/TS	x	x	x	x	x	x

2) Soft Faults: To measure the soft-fault tolerance we have taken the "BL/DUP" design and added a Linear-Feedback-Shift-Register (LFSR) to generate (pseudo-)random numbers in the processor (simulation only). For the soft-fault tolerance it does not matter which soft-fault-tolerant processor design we evaluate, as each design uses the same fault-tolerance mechanism. The LFSR is fed by random initial value at test program beginning. The LFSR is connected to a number of test sets: *I)* Instruction Corruption (IC), *II)* Corruption of data after the Register File (RF), *III)* Control Signals (CS), *IV)* Our fault-tolerance additions and *V)* Other Faults, which includes, among others, the ALU and branch signals. We run 4,000 simulations for each duplication mode and test set in order to get an accurate estimation of the fault tolerance. Soft-fault coverage of the shadow PC is evaluated using similar methodology. However, as the fault-coverage added by the shadow PC is currently independent of the duplication mode, we only profile for one duplication mode.

At arbitrary time, a random signal in the fault set gets replaced by bits from the LFSR. Note that this does not guarantee an error manifests, i.e. the fault may be masked. We account for these masked faults by normalizing the determined fault coverage over the fault coverage (masking) of the Baseline design. The soft-fault coverage is reported in control-flow and total errors. Control-flow errors are determined by comparing the execution time of the program to the known correct execution time, while non-control-flow errors are determined by comparing the result of the program to the known, correct result. We do not check every resource in the architecture for correctness, as this would considerably increase the testing time. The total number of errors is the accumulation of control-flow and other errors.

3) Hard Faults: Due to timing limitations, we have not yet been able to create a fault injector for the faults in the design. As such, testing the hard-fault coverage is done by having a case study for a variety of faults by manually injecting a hard fault in the processor design and, after program execution, determining if the program has reached a #trap address.

4) Overheads: The performance overhead is determined by calculating the number of cycles required to execute a benchmark. For profiling, we have used the MISTY1 encryption benchmark which is a symmetrical encryption method suitable for biomedical implants [18].

Due to availability reasons, the design is synthesized for a UMC 90nm CMOS technology in Synopsys Design Compiler (DC) using Faraday SP libraries, giving the critical-path delay and area cost. The design is required to run under 20 MHz and, therefore, we need to guarantee that critical-path delay does not violate this constraint. The power and, accordingly, the energy consumption figures are not accurate in DC as switching activity of 50% is assumed. In order to obtain more accurate results, we extract the switching activity of the processor from running the benchmarks in Modelsim and this is, along with the synthesized VHDL netlist, passed to Synopsys PrimeTime (PT). As the libraries are not optimized for low-power applications, we expect that the power and energy consumption can further be reduced. To evaluate the effect of increasing the instruction-word size to add the parity bit, we have generated and synthesized SRAM memory components for the same technology as the processor designs. We only discuss the area overhead and a first-order approximation of the power consumption, as we currently do not have an accurate estimation of the memory-power consumption. Future work includes a case study on various memory sizes, along with the system power and energy consumption.

Finally, we introduce a number of compound metrics: *I)* Power-Area product, *II)* Energy-Area product, *III)* Fault-Tolerance/Power-Area and *IV)* Fault-Tolerance/Energy-Area. Metric *I)* summarizes the hard constraints of area and power. Area is important due to the finite size an implant is allowed to take, while the power consumption (dissipation) is related to medical conditions such as tissue burns and vessel dilation. Metric *II)* summarizes the total overhead of our system, taking energy and thus battery lifetime into account. Metric *III)* and *IV)* normalize the added soft-fault tolerance to the compound metrics *I)* and *II)* and, thus, show the gain/cost ratio of the design. In the future we will extend these metrics by also including hard-fault tolerance and increased memory sizes.

B. Experimental Results

In this Section, we present the results of our experiments and, due to space limitations, draw general conclusions. Each result which is expressed in percentages is taken relative to the Baseline design unless stated otherwise.

1) Soft-Fault Tolerance: Fig. 2 shows the soft-error manifestation for the various duplication policies. Each column corresponds to the number of soft faults manifesting into errors, out of which the control-flow errors are depicted as the solid fill. The various columns correspond to the different fault sets as introduced in Section V-A2. We have measured that our fault-tolerance additions are inherently fault tolerant regardless of the duplication policy and, therefore, they are not depicted in Fig. 2. First, we will discuss the total amount

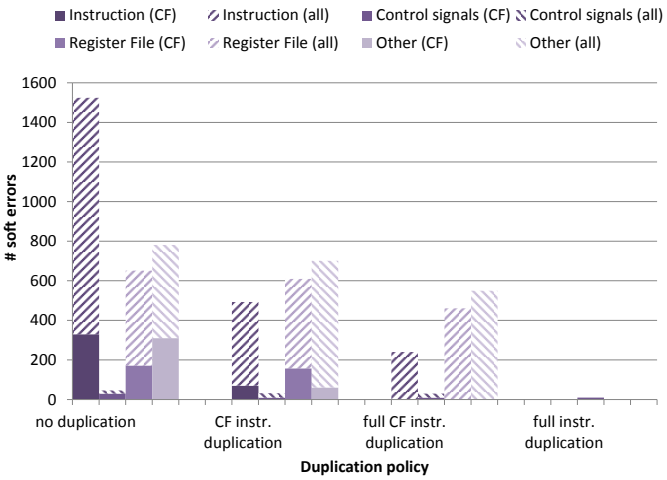

Fig. 2. Manifested soft errors.

of soft errors, after which we will take a closer look at the control-flow errors.

When no duplication is employed, it is observed that only 1,524 out of 4,000 faults result in an error when the fetched instruction is corrupted. However, as the instruction is read and used 100% of the execution time, it would be expected that all faults will manifest into errors. The reason for this observed behavior is two-fold:

i. The corrupted instruction depends on data currently used in the pipeline, injecting a stall instead of the instruction and, hence, no error is manifested;
ii. The result of corrupted instruction is not used, e.g. it corrupts a register which is never used. Since only the result of the function is considered, it is possible that faulty register contents may have slipped past our error measurement method.

Note that in the last case an error *has* occurred, but this is not detected by our testing method. It is expected that more errors will have occurred, but since we compare our results to the Baseline design, which is tested by the same method, we consider this a good approximation of the actual fault tolerance.

It is obvious from Fig. 2 that the number of manifested errors decreases with a more fault-tolerant duplication policy. Here, we describe the most crucial conclusions:

i. The parity bit covers 50% of all instruction corruption, as intended;
ii. As expected, no control-flow errors occur in the "full CF. instr." and "full instr." duplication policies;
iii. The only errors occurring in "full instr." are a result of control-signal faults. While not implemented, control-signal errors can be prevented by e.g. increasing the Hamming distance of the control signals; and
iv. From an experiment with "BL/DUP/PCBR", we have found that adding the shadow PC adds a fault coverage of roughly 7.5%.

The total soft-fault tolerance is also reported in Table II, where column 2 and 3 from the left represent the fault coverage of the "BL/DUP/PCBR" design, i.e. with a shadow

TABLE II
AMOUNT OF SOFT-FAULT TOLERANCE

Dup. mode	All (PCBR)	CF (PCBR)	All (DUP)	CF (DUP)
CF instr.	22.23 %	56.63 %	14.70 %	50.82 %
full CF instr.	41.02 %	99.84 %	33.49 %	94.03 %
full instr.	99.97 %	99.87 %	92.44 %	94.06 %

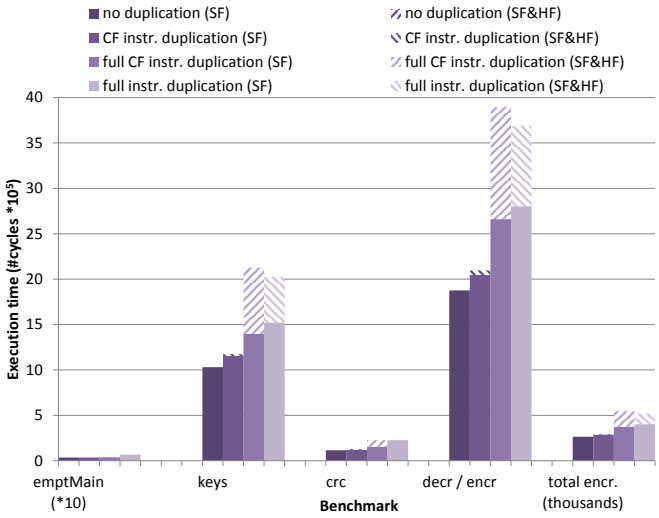

Fig. 3. The number of cycles required to execute the benchmarks for Soft Faults (SF) and Hard Faults (HF).

PC, and column 4 and 5 represent the fault coverage of the "BL/DUP" design. As an afterthought, we are able to both detect *and* correct a soft fault in the PC with a negligible overhead by handling it as if a fault has occurred in the decode stage, i.e. the pipeline is flushed and the PC is reset to the buffered PC-value as discussed in Section IV-A3. This will be included in the next version of our design.

2) Hard-Fault Detection: As discussed in Section V-A3, we have run a number of experiments where a hard fault is manually injected in the processor. Our experiments have shown that the sequences cover the paths they are designed for. A first-order estimation has shown that less than 400 instructions are required to test the processor for hard faults in case gate-level details are available. Currently, we cannot determine the fault coverage as we are unable to test the design exhaustively: We cannot show that *I)* the test sequences will be executed in case a control-flow error occurs and *II)* all hard faults in more complex arithmetic units, e.g. the ALU, are covered using functional testing. Future work includes building a fault injector to accurately measure the fault coverage.

3) Performance: Synthesis results have shown that the critical path is, roughly, the same in all the processor designs, i.e. the performance overhead is solely determined by the number of cycles. Fig. 3 depicts the execution time for the various functions in our benchmark. The bottom part of each bar depicts the execution time when *no* test sequences are scheduled, while the overall bar represents the execution time *with* scheduled test sequences. From this plot, we draw the following general conclusions:

TABLE III
PROCESSOR AND SYSTEM AREA

Processor	Processor Area (#l. cells)	System Area (#l. cells)
Baseline (BL)	50292	413966
BL/DUP	61950 (+23.20%)	436196 (+5.37%)
BL/DUP/PCBR	62161 (+23.60%)	439063 (+6.06%)
BL/DUP/PCBR/TS	64817 (+28.88%)	723678 (+74.82%)

i. When no test sequences are scheduled the performance drop is, at worst, 42%, in "full instr.". This is unexpected, as by executing every instruction twice, a performance overhead of 100% is expected. However, as the duplicated instructions can be executed when the processor would normally stall, the performance drop is considerably lower than the expected 100%;
ii. Without test sequences, the execution time is increased when a more soft-fault-tolerant duplication policy is followed, i.e. more instructions are duplicated;
iii. With test sequences scheduled, the execution time is not increased when no instructions are duplicated. This is expected, as we only schedule our test sequences by replacing nop instructions. This shows that we can achieve hard-fault coverage at a zero-performance overhead;
iv. When test sequences are scheduled, the performance overhead of "full instr." is close to 100%. As the nops are replaced with test instructions, we can not execute the duplicated instruction in the timeslot of a nop, causing the earlier expected performance drop of 100%; and
v. The performance drop of "full CF instr." is 107% when test sequences are scheduled. This shows that, while the policy was intended to have better performance than "full instr." at the cost of fault tolerance, the constant switching between FT modes is not always the most optimal solution with respect to performance.

The critical-path delay of the parity-instruction memory is the same as the Baseline-instruction memory and, as such, the increased memory size will not cause an additional performance overhead.

4) Area: Table III shows the processor-area overheads in number of logic cells for the various processor designs in column 2 and the processor/memory-area overheads in column 3. It is shown that implementing soft-fault tolerance using our technique adds a 23.20% area overhead. The highest area overhead is, as expected, in the most complex design (28.88% in "BL/DUP/PCBR/TS").

The second column of Table IV reports the area of the memories. As the logic for selecting the correct word remains the same when the word size is increased to 17-bits, the area overhead is less than 1/17th (+5.12%). Note that both the instruction and data memory are substantially larger than the processor area. As such, the increase of the processor areas in column 2 of Table III has a marginal impact on the system area in column 3 of Table III. This is most obvious in the "BL/DUP/PCBR/TS" design, where the instruction memory size is doubled ("IMEM 17 (x2)" in Table IV).

TABLE IV
MEMORY AREA AND POWER CONSUMPTION

Memory name	Area (#l. cells)	Dynamic power (μW)	Leakage power (μW)
IMEM 16bit	194353	176.91	346.60
IMEM 17bit	204312 (+5.12%)	185.96 (+5.12%)	371.94 (+7.31%)
IMEM 17bit (x2)	501629 (+131.2%)	429.95 (+131.2%)	873.10 (+139.63%)
DMEM 32bit	157232	106.80	236.02

5) Power and Energy Consumption: The *average power* and *energy consumption* are reported in the leftmost columns 3 and 4 of Table V, respectively, for each processor design and duplication mode. In general, the following observations apply:

i. The soft-fault tolerance additions increase power consumption by 18.82% and energy consumption by 14.58% even when the degree of fault tolerance is zero, i.e. no instructions are duplicated. This can be explained by the fact that we have not implemented clock or power gating;
ii. Adding the shadow PC and br_tst instruction increases the power consumption (by around 5%) when no instructions are duplicated. We expect the shadow PC to be the main contributor due to the high switching activity of the PC;
iii. Without instruction duplication, the power and energy consumption of the "BL/DUP/PCBR/TS" design is increased by roughly 15%;
iv. An interesting observation, which is most apparent in the BL/DUP/PCBR and BL/DUP/PCBR/TS designs, is that the power consumption is *reduced* when instructions are duplicated. By keeping every instruction in the pipeline for an additional cycle dynamic switching activity is reduced leading to reduced power consumption. This is in line with the findings in [11], where a 10% power reduction is achieved by replacing nops with the previous instruction, yet not allowing it to commit any result. We have shown that we can achieve a similar effect and combine a reduction in power consumption with an increase in fault tolerance; and
v. While the power consumption varies between duplication policies, the maximum variation, which occurs between the "none" and "full instr." duplication policies, is 10%. Therefore, the energy consumption increases almost linearly with the execution time up to a maximum of 156.55% in the "BL/DUP/PCBR/TS" design.

Columns 3 and 4 of Table IV report the dynamic and leakage power consumption of the various memories. The dynamic power consumption of the memories is larger than that of the processor. We did not expect this since – while the memories are considerably larger than the processor designs – they should be much less active than the processor. This can be explained by the fact that DC assumes a 50% switching ratio and, as such, the approximation is far from accurate. The leakage power is considerably higher (almost twice as high) than the dynamic power and, as such, is the main contributor of the memory-power consumption. As the reported power consumption is a first-order approximation we currently do not draw any major conclusions on the system-power consumption, as the power-consumption figures can change dramatically in Synopsys PT.

6) Power-Area and Energy-Area: Columns 5 and 6 of Table V depict the PA and EA compound metrics, respectively, of our design. It is obvious that as the area for a particular processor is fixed, the metrics are a linear function of power and energy consumption. It is shown that the minimum and maximum PA overhead are 39.16% and 74.33%, respectively. The EA overhead varies between 35.69% for the simplest design without any fault-tolerance additions and 217.10% for the "BL/DUP/PCBR/TS" design in full duplication mode.

7) FT/PA and FT/EA: Columns 7 and 8 show the fault tolerance, as originally shown in Table II, normalized to the PA and EA metrics. For comparison, we estimate TMR adds 200% area and 200% power consumption overhead, i.e. the relative increase in PA and EA are 800% and, as TMR adds 100% fault tolerance to our fault model, the FT/PA and FT/EA metrics are 1/8 = 0.13. Self-checking circuits add roughly 100% area [3] and we assume the power increases similarly, i.e. the FT/PA and FT/EA metrics are 1/3 = 0.33. We draw the following conclusions for these metrics:

i. As the fault tolerance is low for the "CF instr." and "full CF. instr." policies, the FT/PA and FT/EA metrics are low for these duplication modes. The maximum is 0.14, which is comparable to TMR. While the overheads are much lower, the added fault coverage is as well, resulting in a poor gain/cost ratio;
ii. Adding a shadow PC, as is done in the "BL/DUP/PCBR" design, does not significantly differ from the FT/PA and FT/EA metrics with respect to the "BL/DUP" design and, as such, we conclude that adding a shadow PC is an addition of similar cost as our other low-power additions;
iii. The maximum FT/PA, 0.66, is measured for the simplest fault-tolerance design ("BL/DUP"). This means that in terms of area and power overheads, we perform twice as well than self-checking circuits; and
iv. The best FT/EA ratio is found in the same processor design / duplication mode and is 0.48, which is better than both TMR and self-checking circuits.

As such, we conclude that we perform better than both TMR and self-checking circuits in terms of soft-fault tolerance. Even when all nops are replaced by test sequences in the "BL/DUP/PCBR/TS" design, we outperform both TMR and are on-par with self-checking circuits in terms of FT/PA and FT/EA. However, self-checking circuits and TMR can also detect and tolerate hard faults, respectively. Our analysis will be more conclusive when measurements are done on the hard-fault coverage, memory overheads and, furthermore, a hard-fault-recovery mechanism is added to our architecture.

VI. CONCLUSIONS

In this work we have presented a fault-tolerant architecture for a design framework targeting biomedical applications [1]. Soft faults are tolerated by an instruction duplication technique, in which all instructions or a subset of instructions are duplicated. The set of instructions which needs to be

TABLE V
POWER CONSUMPTION, ENERGY CONSUMPTION AND COMPOUND METRICS

Processor	Dup. mode	Power	Energy	PA	EA	FT/PA	FT/EA
Baseline	none	212.6 (μW)	132.0 (μJ)	10.7 (W*#l.cells)	6.6 (J*#l.cells)	-	-
BL/DUP	none	+18.82%	+14.58%	+40.71%	+35.69%	-	-
BL/DUP	CF. instr.	+17.50%	+35.85%	+39.16%	+60.88%	0.03	0.08
BL/DUP	full CF. instr.	+18.80%	+49.01%	+40.69%	+76.47%	0.11	0.14
BL/DUP	full instr.	+17.56%	+77.03%	+39.23%	+109.65%	0.66	0.48
BL/DUP/PCBR	none	+23.68%	+33.76%	+59.40%	+81.82%	-	-
BL/DUP/PCBR	CF instr.	+22.41%	+53.23%	+57.76%	+97.60%	0.08	0.06
BL/DUP/PCBR	full CF instr.	+24.17%	+60.19%	+60.03%	+106.46%	0.15	0.10
BL/DUP/PCBR	full instr. dup.	+16.58%	+91.17%	+50.25%	+146.38%	0.66	0.41
BL/DUP/PCBR/TS	none	+34.86%	+39.72%	+74.33%	+66.75%	-	-
BL/DUP/PCBR/TS	CF instr.	+33.07%	+55.33%	+71.90%	+92.09%	0.02	0.06
BL/DUP/PCBR/TS	full CF instr.	+26.63%	+89.40%	+63.27%	+134.10%	0.10	0.10
BL/DUP/PCBR/TS	full instr. dup.	+26.66%	+156.55%	+63.44%	+217.10%	0.61	0.32

duplicated can be set at runtime, allowing a dynamic trade-off to be made between performance and fault tolerance. Hard faults are tolerated by running test sequences when the processor stalls and by hardware additions for functionally untestable paths. Our techniques have been applied to the SISC Baseline processor [13] and have been analyzed in terms of fault tolerance, area, performance, power consumption and energy consumption as well as a number of compound metrics.

We have shown that up to 99.97% of all soft faults can be tolerated with affordable processor area and average power consumption overheads (23.2% and 17.6%, respectively) under our fault model. The energy overhead is directly related to the performance and is therefore directly related to the required degree of fault tolerance. In a setting where both soft faults and hard faults are assumed to be well-covered (full instruction duplication with test sequences scheduled in the program), the energy overhead can be as high as 157%. On the other hand, the energy overhead can be as low as 40-60%. when no fault-tolerance is required. The overheads in power and energy consumption could be reduced if e.g., clock gating is employed. In short, the area and power consumption overheads are acceptable, in particular when compared to the amount of faults covered. As such, we conclude that our techniques are suitable for (ultra-)low-power architectures.

It has been shown that the performance and energy consumption are the major overheads in our architecture. However, a trade-off can be made between performance and fault tolerance, both at processor-design time, compile time and at runtime (through instruction level configurability). We believe that due to the high portability of our techniques, the low-power-consumption overhead and the easy and dynamic configuration make our architecture viable for a design framework targeting future biomedical applications.

REFERENCES

[1] Smart implantable Medical Systems, *http://sims.et.tudelft.nl/*, Consulted on 25/09/2011.

[2] Johnson, B.W. *The Electrical Engineering Handbook*. CRC Press LLC, 2000, ch. 93.1-93.4, ISBN: 978-0849385742.

[3] A.P. Kakarotmta, V. Spiliotopoulos, S. Nikolaidis, C.E. Goutis. COSAFE: efficient safety-critical portable system design approach. *IEEE International Workshop on Biomedical Circuits & Systems*, 2004, pp. 13 - 16.

[4] Saeed Shamshiri, Hadi Esmaeilzadeh and Zainalabdein Navabi. Instruction-Level Test Methodology for CPU Core Self-Testing. *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, pp. 673 - 689, 2005.

[5] Nektarios Kranitis, Antonis Paschalis, Dimitris Gizopoulos and George Xenoulis. Software-Based Self-Testing of Embedded Processors. *IEEE Transactions on Computers*, vol. 54, pp. 461 - 475, 2005.

[6] N. Oh, P.P. Shirvani and E.J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Trans. Reliability*, vol. 51, no. 1, pp. 63 - 74, Mar. 2002.

[7] Reis, G.A.; Chang, J.; Vachharajani, N.; Rangan, R.; August, D.I. SWIFT: software implemented fault tolerance. *Code Generation and Optimization*, pp. 243 - 254, 2005.

[8] M. Franklin. A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors. *IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 207 - 215, 1995

[9] Demid Borodin, B.H.H. (Ben) Juurlink and Stamatis Vassiliadis. Instruction-Level Fault Tolerance Configurability. *ICSAMOS*, pages 110 - 117, 2007.

[10] Roshan G. Ragel and Sri Parameswaran. Hardware Assisted Pre-emptive Control Flow Checking for Embedded Processors to improve Reliability. *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pp. 100 - 105, 2006.

[11] Pejman Lotfi-Kamran e.a. Dynamic Power Reduction of Stalls in Pipelined Architecture Processors. *International Journal of Design, Analysis and Tools for Circuits and Systems*, vol. 1, pp. 9 - 15, June 2011.

[12] Wei-Cheng Lai and Kwang-Ting (Tim) Cheng. Instruction-level DFT for testing processor and IP cores in system-on-a-chip. *Proceedings of Design Automation Conference*, pp. 59 - 64, 2001.

[13] D. Siskos. A Co-processor for a Secure Implantable Medical Device. MSc Thesis, March 2011.

[14] M.L. Bushnell and V.D. Agrawal. Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits, 2000.

[15] Albert Meixner, Michael E. Bauer and Daniel Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* , pp. 210 - 222, 2007.

[16] Wei-Cheng Lai, Angela Krstic and Kwang-Ting (Tim) Cheng. Test Program Synthesis for Path Delay Faults in Microprocessor Cores. *ITC International Test Conference*, pp. 1080 - 1089, 2000.

[17] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, pp. 10-16, Nov.-Dec. 2005.

[18] C. Strydis and D. Zhu and G.N. Gaydadjiev. Profiling of symmetric encryption algorithms for a novel biomedical-implant architecture. *ACM International Conference on Computing Frontiers (CF'08)*, pp. 231 - 240, 5-7 May 2008.