A Dependable Coarse-grain Reconfigurable Multicore Array

Georgios Smaragdos*, Danish Anis Khan†, Ioannis Sourdis†, Christos Strydis*, Alirad Malek† and Stavros Tzilis†
*Neuroscience dept., Erasmus University Medical Center, The Netherlands
{g.smaragdos, c.strydis}@erasmusmc.nl
†Computer Science and Engineering dept., Chalmers University of Technology, Sweden
{adanish, sourdis, aliradm, tzilis}@chalmers.se

***Abstract*—Recent trends in semiconductor technology have dictated the constant reduction of device size. One negative effect stemming from the reduction in size and increased complexity is the reduced device reliability. This paper is centered around the matter of hard-error tolerance and graceful system degradation in the presence of permanent faults. We take advantage of the natural redundancy of homogeneous multicores following a sparing strategy to reuse functional pipeline stages of faulty cores. This is done by incorporating reconfigurable interconnects next to which the cores of the system are placed, providing the flexibility to redirect the data flow from the faulty pipeline stages of damaged cores to spare (still) functional ones. Several micro-architectural changes are introduced to decouple the processor stages and allow them to be interchangeable. The proposed approach is a clear departure from previous ones by offering full flexibility as well as highly graceful performance degradation at reasonable costs. More specifically, our coarse-grain fault-tolerant multicore array provides up to $\times$4 better availability compared to a conventional multicore and up to $\times$2 higher probability to deliver at least one functioning core in high fault densities. For our benchmarks, our design (synthesized for STM $65nm$ SP technology) incurs a total execution-time overhead for the complete system ranging from $\times$1.37 to $\times$3.3 compared to a (baseline) non-fault-tolerant system, depending on the permanent-fault density. The area overhead is 19.5% and the energy consumption, without incorporating any power/energy-saving technique, is estimated on average to be 20.9% higher compared to the baseline, unprotected design.**

I. Introduction

Recent engineering trends have resulted in the emergence of new challenges in multicore-computer design. With transistor device sizes now in the realm of nanometers and their density increasing, one of the important factors that suffers is reliability. Transistors of smaller size are more sensitive to wear-out phenomena [1] and dielectric breakdown [2] resulting in permanent faults that can render a device unusable. Moreover, it is argued that for the same reasons, the ability of the designer to create a fault-free system will become increasingly more difficult in the future while device testing will become increasingly considerably lengthier (and costlier) [3]. The wide use of embedded systems in various application domains and the fact that such systems tend to become more complex and advanced as time passes, makes reliability and availability an even more pressing issue. Furthermore, there is a number of mission-critical applications whereby a fault is unacceptable; e.g. space, automotive, and medical applications. Such applications require efficient techniques for fault tolerance to deal with the increasing number of faults in emerging technologies.

In this paper we concentrate on system recovery from permanent (or hard) errors – simply denoted in this paper as *faults* – aiming at increased system availability and fault tolerance at high fault densities. One general approach to fault tolerance relies on dividing a design into basic blocks identical to each other, called *Substitutable Units* whereupon so-called *sparing* strategies are employed: A faulty block of a system is substituted by a spare (functioning) one. Clearly, this strategy requires redundancy of components used to replace damaged parts so that the system can remain functional. The granularity of the redundant components can be large (e.g. memories [4]) or small (e.g. adders [5]).

In our work we seek to combine sparing with the natural redundancy that multicores exhibit creating a fault-tolerant (FT), coarse-grain, reconfigurable multicore array (MA). Multicore systems are often composed of multiple identical components, thus encompassing significant amounts of inherent regularity and redundancy. Then, if the architecture allows it, the spare (undamaged) components could be used to repair a faulty core improving the system's availability and fault tolerance.

We use the *pipeline stage* of a processor as a Substitutable Unit to implement our sparing strategy. When faulty cores are detected, we isolate the dissfunctional pipeline stages and replace them with existing spare functional ones to create new working cores. This requires rewiring inter-stage connections to divert the data flow around the faulty stages.

For diverting to spare units, state-of-the-art works such as the StageNet [6], the Viper [7] and the CCA [8] have used crossbars or all-to-one multiplexers introducing a (high) uniform delay to the critical path of a processor. In contrast, *we use reconfigurable wires to interconnect the decoupled stages*. Reconfigurable wires have a delay proportional to the distance between the spare and the replaced faulty unit. Consequently, algorithms can be used to generate configurations that replace faulty parts with closely located spare resources recovering lost performance [9]. We, also, propose to pipeline these reconfigurable wires, thus introducing extra, empty (bubble) pipeline stages in the cores in order to further reduce the wire delay at the cost of deeper pipelines. Therefore, for maintaining a reasonable complexity/flexibility ratio, we are considering reconfiguration *only* for the wires between stages.

Besides, we wish our proposed design to be attractive for realistic use. We have, thus, placed the strict design

This work has been supported by the EU-funded FP7 project DeSyRe: on-Demand System Reliability (Grant agreement no: 287611).

requirement that *the proposed FT MA (in all its configurations) will be binary-compatible with the baseline, unprotected MA*. This means that the multicore reconfigurability functionality will have to be completely transparent to the application level.

However, to enable MA reconfigurations while guaranteeing backwards (ISA) compatibility regardless of the different MA configurations, several micro-architectural modifications are required. These modifications, both at the core- and at the system-level are tackled in this paper.

Concisely, the main contributions of this work are:

- The design and implementation of a novel core based on a 5-stage RISC pipeline with decoupled stages to simplify reconfiguration. Through (i) distributed control logic, and (ii) dynamic insertion of a variable number of empty stages, the core offers binary compatibility among the baseline MA and the various FT MA configurations.
- The design, implementation and evaluation of a coarse-grain, reconfigurable MA that uses the above core and reconfigurable interconnects to tolerate permanent faults.
- The fault-tolerance analysis of our FT system compared to a reference FT system using core-level redundancy.

The remainder of this paper is organized as follows: We present related work in Section II. In Section III, we describe the design challenges and the micro-architectural implementation of the pipeline, interconnect and the complete FT system. In Section IV, we evaluate our approach on specific FT MA instances (4-core, 6-core), highlighting its benefits in terms of fault tolerance and availability while quantifying its overheads in terms of performance, area, power and energy consumption. Finally, in Section V overall conclusions are drawn.

II. Related work

In the past, several related works have been described on homogeneous multicores. These solutions vary depending on the chosen granularity that affects the performance/fault-tolerance trade-offs and nature of the architectures.

One approach for the tolerance of permanent faults is that of core redundancy [10], [11], defining the complete core as a substitutable resource. This solution, in general, has the advantage of simple implementation and lower performance costs, compared to other granularity choices, but tends to be less fault-tolerant than other solutions for high fault densities where device degradation becomes less graceful.

Other works choose a finer granularity defining as substitutable resources the processor components. The Viper [7] uses a distributed execution engine composed of loosely coupled functional units to form out-of-order pipelines, which execute bundles of instructions (small blocks of instructions ending in a control-flow instruction). At the same granularity, a damaged core can share components (such as ALUs or floating-point units) as described in [12]. Here, when a component is damaged, the core can continue to operate correctly by replacing the faulty parts with equivalent ones from a neighboring core, suffering some performance overhead.

There are also a few works that have used the pipeline stage as a Substitutable Unit. Such works are the *StageNet* [6] and the *Core Cannibalization Architecture* (CCA) [8]. The CCA approach does not allow a core to borrow more than one stage, in order to keep the complexity of the control logic and, thus, the performance overhead low. This constraint impacts, though, the flexibility of the system and limits its fault tolerance. The StageNet architecture decouples the various stages of a 5-stage processor and interconnects them using crossbars [6]. When an erroneous stage is detected the crossbars reroute the data from the faulty processor stage into another spare one of the cluster. This enables the design to be very flexible in replacing faulty stages, alleviating the constraints found in the CCA. The use of crossbars in this design, however, incurs significant performance costs compared to the original 5-stage pipeline. In the crossbar, the delay is constant regardless of the data destination. In every case, the StageNet exhibits the maximum delay overhead regardless of the number of operational cores and of the distance of the neighbors sharing stages.

In our proposed reconfigurable interconnect, the performance overhead is not constant and depends on the number and position of faults. Additionally, the flexibility of the reconfiguration is the same as in the StageNet, without the limitations posed in the CCA architecture.

III. The coarse-grain reconfigurable multicore architecture

In this section we describe the proposed coarse-grain reconfigurable multicore architecture for providing tolerance to permanent faults as well as the micro-architectural changes needed to allow the pipeline stages to be decoupled and substitutable. Figure 1 illustrates a generic view of our approach. Four cores have reconfigurable wires in between their stages to bypass faulty stages (shadowed boxes) and form three functional pipelines.

This MA is composed of a number of RISC cores, each having 5 pipeline stages: Instruction Fetch (IF), Decode (DEC), Execute (EX), Memory (MEM) and Write Back (WB). The ISA comprises 32-bit instructions of 3 basic instruction types: *integer-arithmetic*, *memory* and *control-flow*. The register file (RF) has 16 32-bit registers. Finally, there are separate 4Kbyte instruction and 32Kbyte data memories of 32-bit words. In this work, the baseline RISC architecture is modified to support the necessary flexibility for tolerating permanent faults.

The design of an adaptive FT MA as the one we propose is faced with two main challenges: (1) The architectural challenge of building a FT multicore system, which is the subject of this paper, and (2) the selection of a new, optimal core configuration whenever a new fault occurs. The optimality question is relevant here since *the performance overhead varies when connecting stages of different distances*. This problem has already been shown to be NP-complete and has been tackled using a *heuristic-search method* in a work complementary to the current paper: Vasilikos et al. [9] have shown that a fast-greedy heuristic algorithm can generate array configurations which achieve performances of 70% or better with respect to an (exhaustively sought) optimal configuration. It is worth noting that both the StageNet and the CCA *cannot*

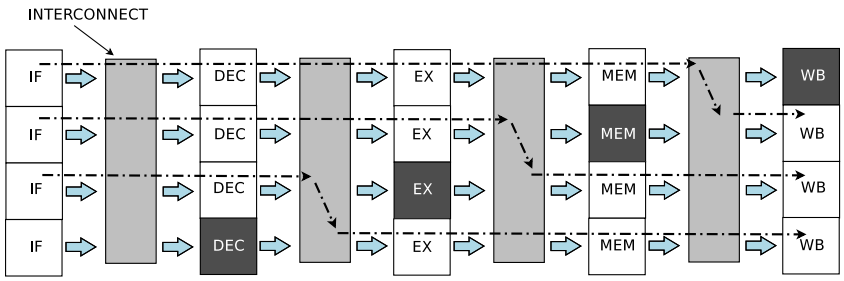

Fig. 1. A generic view of the proposed coarse-grain reconfigurable multicore architecture. Dark boxes signify faulty stages. In this example, by rerouting the data-flow we can create 3 functioning cores.

exploit such optimizations due to the fixed delay introduced by the crossbars between the stages. In addition, the CCA places strict constraints on the reconfiguration options to guarantee that the rearrangement of stages will not drop performance to unacceptable levels; therefore, choosing a configuration is – in that case – substantially simpler.

A. General design functionality

As previously discussed, on one hand the StageNet provides very high flexibility but with a fixed, high overhead in performance due to the constant wire delay. On the other hand, the CCA achieves better performance but with strict constraints which limit core flexibility and fault tolerance. Our approach offers a combination of *high flexibility* and *graceful performance degradation* by exploiting decoupled pipeline stages and reconfigurable interconnects. A permanent fault can be detected using techniques such as in [13]. In the presence of a fault, the MA is turned off and a new configuration is pursued, as described in [9]: The faulty stage is isolated and, if possible, replaced by an available (identical) stage scavenged from other damaged cores in the MA. Interconnect reconfiguration enables the creation of new cores by tweaking the interconnect switches in a way that the data-flow is redirected through the spare stages (Figure 1). A simple control using the result of the heuristic search can handle the switch re-configuration. In the scope of this work we consider the interconnect as unprotected. As far as the control logic of each core is concerned, the core sparing policy makes it obvious that the end design cannot be governed by global control signals. Instead, the control path needs to be *distributed* across the various, decoupled stages.

The algorithm for generating new configurations has the twofold objective of providing as many functioning cores as possible and of maximizing their performance. An obvious concern for performance is the interconnect wire delay: The further away the cores sharing stages are, the longer it takes for the data to travel through the interconnect and the larger the impact on the critical-path delay (and performance) of the design. To tackle this, we pipeline the interconnect. We specify that each core in the array has its own plane wherein the wire delays are in acceptable levels, meaning that signals between stages of the *same* core do not need to be pipelined further. On the other hand, we assume (perhaps pessimistically) that the wire delays for data to travel from one plane to its immediate neighbor need *one extra register level* to avoid violating the baseline critical-path delay. The further away the data needs to travel from its original core, the more extra empty (or *bubble*) stages need to be inserted into the respective pipeline to maintain a constant critical-path delay, of course at the expense of additional clock cycles.

The immediate effect of this method is that we eventually get a design with a variable number of pipeline stages. Our core architecture should support such functionality and *operate correctly* while remaining oblivious to the number of (extra) stages in its pipeline; that is, reconfigurability needs to be transparent to the application level. This design requirement is imperative for guaranteeing *binary compatibility* between the FT MA and a reference, unprotected MA, regardless of the configuration state of the former. Binary compatibility means that no recompilation of the binaries running in the cores is required, which is a crucial feature for adoption by the industry of our proposed FT MA.

The binary-compatibility requirement further explains our design choice to limit reconfigurability to only the wires between stages (cf. functional-unit reconfiguration). This approach not only makes the reconfiguration process faster (by keeping complexity tractable) but also contributes to making the reconfiguration effects transparent to the core ISA.

B. The fault-tolerant pipeline

The decoupling of pipeline stages and the design requirements mentioned in the previous section call for several microarchitectural changes in the data flow and control flow of the baseline core. Allowing a variable number of stages per processor and eliminating global control directly influences the resolution of hazards in a typical 5-stage RISC architecture. The issues that need to be addressed to provide a fault-tolerant pipeline according to the previous requirements are three:

1) Data-hazard resolution;
2) Control-hazard resolution and pipeline flushing; and
3) Global-stall support.

Next, we present these three issues in more detail and describe the distributed strategies devised for tackling them.

1) Data-hazard resolution: Since we allow a variable pipeline depth, the moment that a value (produced by an instruction) is committed is not fixed. Whenever extra bubble stages are inserted after the DEC stage, the direct effect is that the number of potentially uncommitted instructions traveling through the pipeline increases. Thus, the instruction window within which a data conflict may occur increases, too.

Having in mind that the more stages we have in the pipeline the more places need to be checked for data conflicts, one

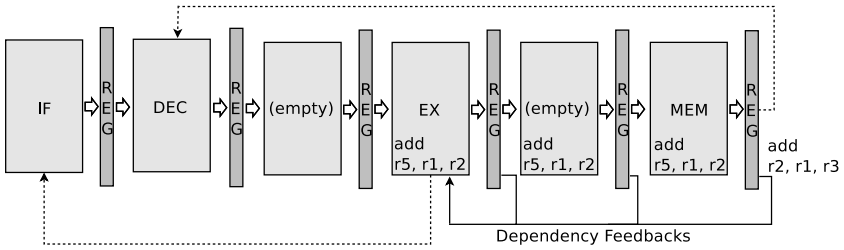

Fig. 2. Data conflicts on a reconfigured core. The reconfiguration calls for 2 extra bubble stages (a case that can arise if EX is replaced). Extra dependency checks are required.

immediate *optimization* can be done in the architecture: the incorporation of the write-back stage in the interconnect. Even though the WB logically does separate actions, physically it shares the bulk of its logic with the DEC stage (practically with the RF). The only extra hardware in the WB stage is, then, a few multiplexers. If these are placed physically at the DEC stage, the only logic that is left in the WB stage is the write-back (feedback) wires which can be easily *incorporated in the interconnect*. The functionality of the processor remains the same but, without loss of generality, we can now approach the architecture as a 4-stage pipeline. The number of possible bubble stages that need to be inserted decreases as there are fewer possible connections between stages. Also, fewer possible uncommitted instructions are in flight at any given time, simplifying data-hazard handling substantially.

Generally speaking, the proposed MA increases data-hazard resolution complexity. Every single bubble stage, inserted due to a reconfiguration, has to be checked for dependencies in order to detect all data conflicts. Moreover, if the pipeline incorporates value bypassing to service the conflicts, a feedback line is needed from each new stage to the stages that might require a forwarding value, mainly the EX and MEM stages. A processor with two bubble stages and the feedback lines needed to resolve its data hazards are illustrated in Figure 2. Additionally, there exist cases where simple bypassing is not sufficient for resolving data conflicts. These are conflicts between a memory operation and a subsequent arithmetic operation. In such cases, *pipeline stalling* is required (to be discussed in detail in Section III-B3) combined with some more advanced data-hazard resolution mechanism. The problem in our FT-MA case is that the number of stall cycles needed varies depending on the number of stages in the pipeline. In either case, simple bypass signals between stages cannot work and a resolution mechanism is needed which is agnostic to the number of stages in the pipeline and can handle the variable pipeline depth. This mechanism is called *pipeline state saving* and is described below.

Pipeline state saving keeps track and bypasses uncommitted results in the pipeline locally, eliminating the need for dependency checks and forwarding via feedbacks. It is implemented by saving copies of the produced (still uncommitted) results locally at the stage where they are produced and would possibly need to be forwarded to for data-hazard resolution. There are two such stages in our pipeline, namely the EX and MEM stages. For this reason, all uncommitted results of instructions in flight are stored locally in the EX and MEM stages in two FIFOs acting as *bypass buffers*. Each bypass buffer provides read access to its entries in order to forward any of the stored values.

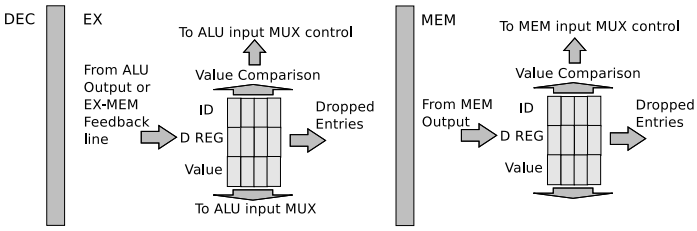

Fig. 3. The bypass buffers implementing the bypass mechanism.

In order to have sufficient information about the uncommitted instructions in the pipeline, 3 types of data need to be stored in the bypass buffers:

- The instruction type encoded in a 2-bit ID, which determines whether an instruction (i) produces a value in the ALU or (ii) is a Load or (iii) does not produce a value to be written to the RF;
- the destination register (DREG) of the produced value (if any); and
- the actual produced value (if any).

The size of the bypass buffers is determined by the worst-case pipeline depth, since it must be deep enough to store all uncommitted results. This, in turn, depends on the number of processors (N) in the array and the maximum number of bubble stages inserted to connect two stages in a valid array configuration. A bypass buffer of $2N-1$ entries in each of the EX and the MEM stages suffices to cover the worst-case pipeline depth. As depicted in Figure 3, the buffer in EX provides the values that arithmetic instructions need from instructions still traveling in the subsequent stages. The logic in MEM services the dependencies concerning memory (load/store) operations. When an instruction enters one of the two stages, the buffer entries (ID and DREG fields) are checked for a data conflict. A match in any of the entries indicates a hazard and the value of the uncommitted instruction is read and forwarded. If a value produced in the MEM stage is not in the buffer yet, the stage stalls (including the buffers) until the value is provided. A bypass signal is used to provide values from the MEM to the EX buffer. This signal is also pipelined if bubble stages have been injected between the EX and MEM stages. Since the maximum distance possible between two uncommitted instructions that produce an as-of-yet undetected conflict is $2N-1$ (due to the maximum span

of N cores, forward and return signals), the $2N-1$ entries in the buffers are sufficient to keep track of all instructions in flight without evicting useful entries prematurely.

2) Control-hazard resolution and pipeline flushing: The second major issue to be addressed is control-hazard resolution and the mechanism of pipeline flushing. When a branch misprediction occurs, a number of invalid instructions, fetched before branch resolution occurs, need to be flushed. In order to avoid long wires in the MA, a scheme is needed for flushing the pipeline without using global signals. We have incorporated a scheme similar to that in the StageNet, yet somewhat simpler because of the difference in the architectures of the two solutions. The StageNet implements a common EX and MEM stage and a separate Issue and DEC stage, including a scoreboard; this creates the need for more localized checks, compared to our case, for correct control-hazard resolution.

As shown in Figure 5, the IF and EX stages include an *Instruction-Flow bit (InF)*. Its value is used to identify whether an instruction is valid or is part of an invalid instruction stream due to a misprediction. More precisely, each instruction travels in the pipeline carrying one additional bit, a *Stream Identification Bit (SIB)*. Its value is given in the IF stage, when the instruction is fetched, by the InF register of that stage. When an instruction enters the EX stage, its SIB is compared to the InF bit of the stage. If the values match, the instruction is allowed to continue, otherwise it is flushed and not allowed to make any changes in the bypass buffers of the EX stage.

In case a branch misprediction is detected in the EX stage, the value of the InF bit of the stage is inverted. When the correct branch target is loaded to the PC, the IF-stage InF-register value is also toggled. As a result, the invalid instructions that were fetched in-between, are marked with a different SIB than the current value in the EX InF register and are therefore flushed. The subsequent instructions will have, on the other hand, the same SIB value as the InF register in the EX stage and execute normally. As such, the flush mechanism for a branch misprediction is implemented locally in the EX stage, eliminating the need for global flush signals, while requiring very little extra circuitry.

3) Global-stall support: Decentralizing the core control logic calls, lastly, for a new mechanism to facilitate pipeline stalling. In order to avoid global signals there are two methods that could be used, each with is own merits:

- *Per-cycle propagation* of the stall signal across stages.
- Implementation of the stall functionality by *flush and reload* instead of an actual pipeline stall.

In the first alternative, stalls propagate their signals along the entire pipeline. The number of additional cycles needed for stall propagation is expected to be lower than that of flushing. However, stalling requires the use of *double buffers* at the output of every pipeline stage. This would ensure that no instructions are lost due to the propagation delay of the stall signals from stage to stage. Besides the area overhead, this would increase the critical-path delay of the pipeline and, in addition, would virtually introduce extra "stages" in the core pipeline requiring larger bypass buffers and even longer delay. Therefore, we chose to implement a flush/reload scheme.

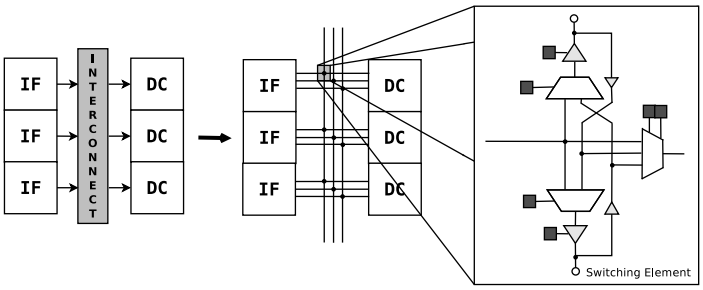

Fig. 4. Reconfigurable interconnects using bidirectional switches.

The flush/reload scheme is simpler to implement: When the need for a stall arises, the core drops the currently fetched instruction as well as all subsequently fetched ones the moment it produces a stall. Then, it reloads the dropped instruction on the IF stage. By the time the instruction reaches again the stage in which the need for pausing the pipeline arose, the situation is expected to have been resolved and execution resumes; if not, it is stalled (flushed) again. The bypass buffers are also stalled when an invalid instruction enters the stage so as to retain their state throughout the stalling.

In our particular RISC pipeline, the need for stalls due to a dependency can only arise when an instruction requires a value in the EX stage. As such, we use the same flush mechanism described for control hazards. The only addition is some extra logic for activating the mechanism when a dependency requiring a stall is detected and for adding this case in the PC selection in the IF stage.

The same stalling mechanism is needed in case of *cache misses*. In our experiments, we consider one level of memory (4KB IMEM, 32 KB DMEM) which takes one cycle to access and, therefore, do not have cache misses. In a different memory hierarchy, cache misses may yield additional performance overheads, compared to the traditional stall scheme. However, if we synchronize the resuming of execution, after flushing, with the arrival of the data that caused the cache miss, then the performance penalty of flushing would be the same as for stalling. In case the execution is resumed only after the data arrives, then the additional overhead compared to stalling shall be equal to the number of cycles needed for the instruction to reach again the MEM stage. Such overhead is probably small compared to the number of cycles needed to fetch data from the main memory or the disk.

C. Reconfigurable interconnects

An important element of the complete core is the reconfigurable wires between the cores stages. The more flexibility one includes in the switching nodes, the better configurations can be achieved in terms of performance and efficiency, paying the cost of added functionality in area and power. As illustrated in Figure 4, the reconfigurable wires interconnect two subsequent stages of the same cores, as well as stages of different cores. We have opted for *bidirectional wires* rather than double, unidirectional wires. This choice reduced the total number of wires needed but possibly increased the delay of the switches which now need to use tri-state buffers. Each switch is controlled by a 3-bit signal. The bidirectional switches are able to send data from the level above to the level below (and

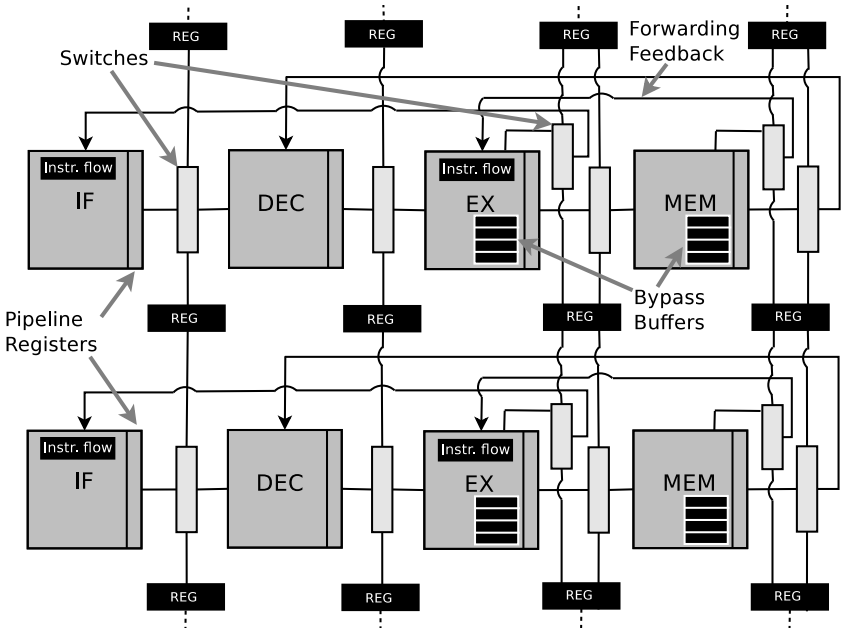

Fig. 5. The proposed multicore system, illustrating the decoupled pipeline stages, the interconnect switches, bi-directional registers, bypass buffers and instruction-flow registers.

vice versa) while the normal input/output path is also in use. This offers to the system high reconfiguration versatility and results in better configurations in case of a fault.

D. Overview of the proposed system

An overview of the proposed system is shown in Figure 5. Each core consists of 4 decoupled stages interconnected using switches to reroute the data according to the configuration of the MA. Data traveling from one core to another pass from one switch to the one directly above or below through a pipelined wire. Except for the switches that forward the data coming out of the pipeline registers from one stage to the next, there are also additional switches to route the bypass signals from MEM and EX stages. These signals are needed for memory-value bypassing and for branch resolution. To illustrate the costs, an example 4-core array requires 16 switches for passing the data though the pipeline and 8 more for the bypass signals, plus 24 bi-directional registers.

IV. Evaluation

In this section, we first describe our experimental setup and benchmarks used. Subsequently, we measure the performance, area, power and energy overheads of our proposed approach and, finally, analyze the benefits of the reconfigurable MA in terms of fault tolerance and availability.

A. Experimental setup

For fast processor implementation, we have used a high-level description language (Lisa 2.0) through the Synopsys Processor and Compiler Designer tool. With this toolset we automatically generated an RTL description of our processor as well as a simulator and a compiler. However, this advantage comes with the cost of having less control on the produced RTL, which could be significantly more optimized if done manually. Cadence Encounter RTL compiler was used to synthesize our design at STM $65nm$ SP technology and acquired measurements of area, power, energy and timing.

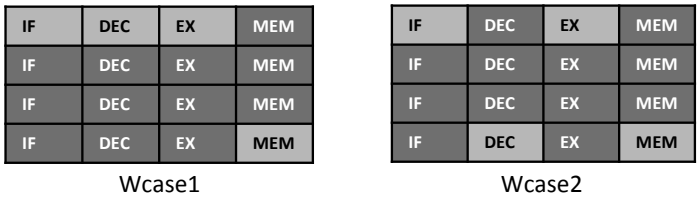

Fig. 6. The two worst-case scenarios used in our experiments. In light gray the still functional stages of the configuration.

As benchmarks for our evaluation, we use small custom, C-code fragments running in a loop, created to test the efficiency of our proposed MA design under hazards. The benchmarks are extreme cases of instruction sequences that exacerbate processor performance and are important for assessing worst-case costs as well as for debugging and validating our design. Our benchmarks are the following:

- **Function-Argument Heavy**: Code that includes a function with a large number of arguments, prone to creating data hazards between arithmetic and memory operations, useful to assess the impact of the Flush/Reload mechanism when stalling the pipeline.
- **Heavy Read-After-Write Conflict**: Code that produces a large amount of RAW hazards, activating the bypass mechanism, which has an increased delay or flush penalty as the number of pipeline stages increases.
- **Heavy Branch-Mispredict**: A loop with non-taken branches used to assess the flush/reload mechanism, here mostly employed for branch misses. A significant

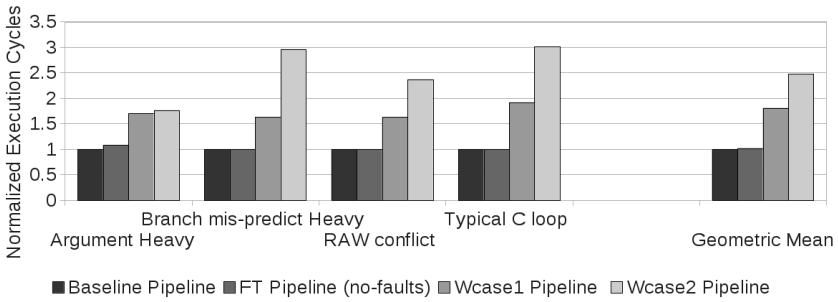

Fig. 7. Execution cycles for each benchmark for each test design, normalized by the baseline cycle count.

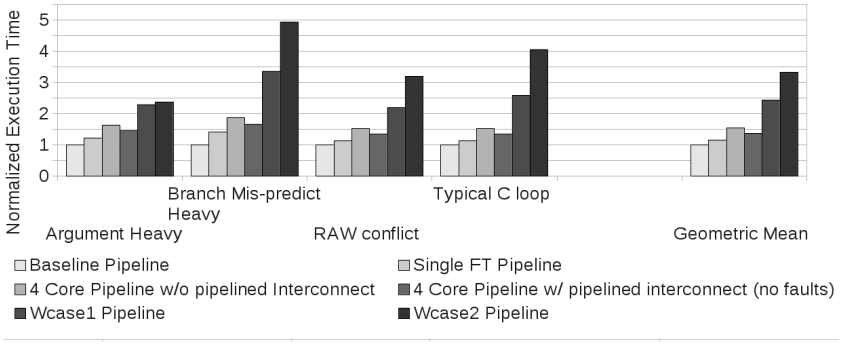

Fig. 8. Execution time for each benchmark normalized to baseline values.

performance impact is expected in configurations with long pipelines.

- **Normal-C for-loop**: A typical C for-loop. A large overhead is expected in configurations with long pipelines.

The experiments were performed in:

1) a (single-core) Baseline pipeline;
2) our new (single-core) FT pipeline, as would operate in a 4-core system without faults (see Section III); and
3) in two different, 4-core complete configurations representing two of the worst instances (Wcase1 and Wcase2) in terms of pipeline depth (Figure 6).

Wcase1 and Wcase2 represent instances of 75% of the system being damaged while still being able to provide a functioning core. In Wcase1, the top core uses the MEM stage of the bottom core. The system adds the maximum possible number of stages between EX and MEM stages suffering from the longest possible delays in both the WB and the forwarding signals. Furthermore, this increases the instruction window that data hazards might occur in while at the same time the pipeline requirement to wait for the transfer of values from MEM to EX for bypassing is maximized. In Wcase2, which is the worst-case possible in our system, a pipeline is formed connecting the IF and EX stages of the top-most core with the DEC and MEM stages of the bottom-most core. Here, in addition to the overhead of the previous configuration, we have an increased cost when flushing the pipeline in the case of branch mispredictions and data-hazard stalls due to the extra stages before and after the DEC stage.

B. Experimental results

Figure 7 depicts our benchmark results in terms of execution cycles. Between the baseline and our FT pipeline there is only a small difference in execution cycles in the case of the Argument-Heavy benchmark (about 7.5%), which is attributed to the flush/reload mechanism. On average, our FT processor – in the absence of faults – has a small overhead of 1.8% in execution cycles. For the 2 worst-case configurations, shown in Figure 6, the overheads are considerable ($\times$1.8 and $\times$2.5, respectively) but are incurred only in the worst-case configurations, after a large number of faults has occurred, which would normally render a baseline design entirely non-functional.

Table I reports the timing of the baseline pipeline, compared to our stand-alone FT pipeline and compared to a 4-core system with and without pipelining of the interconnects. The baseline core has a cycle time of 1.45ns. Our micro-architectural changes have yielded a clock cycle of 1.65ns (13.7% overhead). This is mostly a consequence of the bypassing mechanism which, together with the other additions described in the previous section, are all in the critical path reducing the operating frequency. The full 4-core system with non-pipelined interconnect has a clock period of 2.2ns, incurring a substantial overhead of 33.3% compared to the stand-alone FT core and 51% compared to the baseline core. When pipelining the interconnects, these overheads are reduced to 18.1% and 34.4%, respectively.

Considering the number of cycles needed per benchmark and the above operating frequencies, we calculate the overall execution time shown in Figure 8. All values are normalized

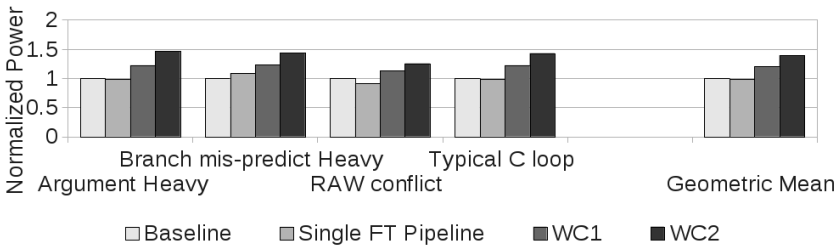

Fig. 9. Power consumption for our benchmarks normalized to baseline.

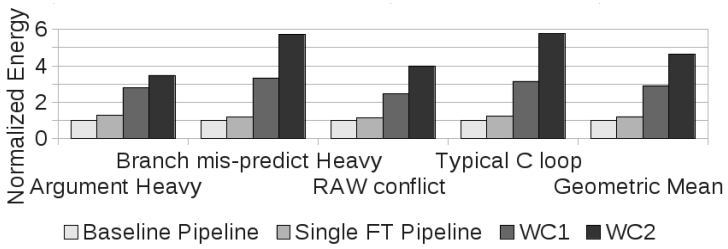

Fig. 10. Energy consumption for our benchmarks normalized to baseline.

TABLE I. TIMING MEASUREMENTS FOR OUR MA.

Design	**Clock Period (ns)**	**Overhead vs Baseline**	**Overhead vs FT-pipeline**
Baseline Pipeline	1.45	–	–
Fault-Tolerant Pipeline (w/o switches)	1.65	13.7%	–
4 Core w/o pipelined interconnect	2.2	51%	33.3%
4 Core w/ pipelined interconnect	1.95	34.4%	18.1%

to the baseline execution time. The increase in execution time ranges from $\times$1.37 (normal fault-free operation of the 4-core array) to $\times$3.3 (Wcase2 scenario) compared to the baseline, depending on the number of faults in the array.

Our architectural changes incur a 6% area overhead in the FT pipeline as shown in Table II. This stems from the bypass mechanisms in EX an MEM, the bypass buffers and the additional logic required by the flush/reload-mechanism for data- and control-hazard resolution. The 4-core system has an area overhead of 19.5% compared to the 4-core baseline array including the overhead of the reconfigurable interconnects.

Our single FT pipeline has *similar* power consumption with the baseline as the increase due to the additional logic is balanced out by the slower clock period. For the worst-case pipelines, we can see an average increase in power of 19.8% for Wcase1 and 39.1% for Wcase2 (Figure 9). In terms of energy consumption, shown in Figure 10, the single FT core exhibits a slight increase of 20.9% on average for our experiments. For our two worst-case scenarios, the energy overhead is much higher (Wcase1: $\times$2.9, Wcase2: $\times$4.6), which is to be expected because of the increase in execution times. It must be noted, though, that we did not use any advanced power-saving technique in our design and experiments.

C. Comparison with related works

Although it is hard to provide a direct comparison with related works, we discuss next the advantages and disadvantages of our solution compared to the 3 closest approaches – CCA [8], StageNet [6] and Viper [7] – when in fault-free mode. The CCA introduces minimal micro-architectural changes allowing stage borrowing only from *immediate* neighbors. This design choice severely limits flexibility and fault tolerance compared to the other approaches but at the same time restricts the execution-time overhead to only 4.5%, as shown in Table III. The StageNet and the Viper provide *similar* flexibility to our solution, yet the performance overheads reported there are, in fact, not the totals since they do *not* take into account the non-trivial delay introduced by the added crossbars. By ignoring this delay, we can state that the StageNet (w/o macro-ops) has a uniform execution-time overhead of $\times$2.1 compared to the baseline [6]. The optimized version with macro-ops exhibits a lower overhead of $\times$1.2. Even by co-factoring in the interconnect costs, our design exhibits a comparable overhead to that of the macro-op-based StageNet and only one third that of the bypass-cache-only StageNet ($\times$1.37). The reason is that our proposed FT MA allows graceful performance scaling with the distance between the interconnected stages. Finally, the Viper array has a variable overhead of 24% when compared to a single out-of-order core and it rises up to 60% when compared to the complete out-of-order baseline array due to increased structural hazards. Moreover, in order for the Viper to facilitate FT out-of-order execution, it requires compiler changes. On the contrary, our FT MA offers full backwards binary compatibility by keeping the hardware configurations transparent from the MA ISA.

TABLE II. AREA MEASUREMENTS FOR SINGLE CORES AND 4-CORE SYSTEM. INSTRUCTION AND DATA MEMORIES INCLUDED IN THE MEASUREMENT.

Design	**Area in mm^2**
Baseline	0.416
Fault-Tolerant Core (w/o switches)	0.441 (+6%)
4 Core System	1.990 (+19.5%)

TABLE III. PERFORMANCE OVERHEADS AND BINARY COMPATIBILITY OF VARIOUS FT MULTICORE ARRAYS IN *fault-free* OPERATION.

FT-array design	**Exec-time overhead vs. baseline core**	**Exec-time overhead vs. baseline array**	**Binary compatibility**	**Flexibility/ Fault-tolerance**
Core-level redundancy	—	—	yes	low
FT MA	37.0%	37.0%	yes	high
CCA	4.5%	4.5%	yes	medium
StageNet (w/o macro-ops)	>110.0%	>110.0%	yes	high
StageNet (w/ macro-ops)	>20.0%	>20.0%	no	high
Viper	>24.0%	>60.0%	no	high

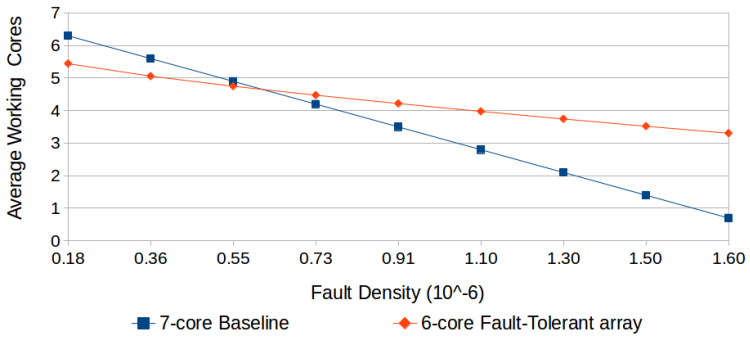

Fig. 11. Average availability: Average number of cores. Case1: Baseline (7 cores), Case2: Fault-Tolerant array (6 cores).

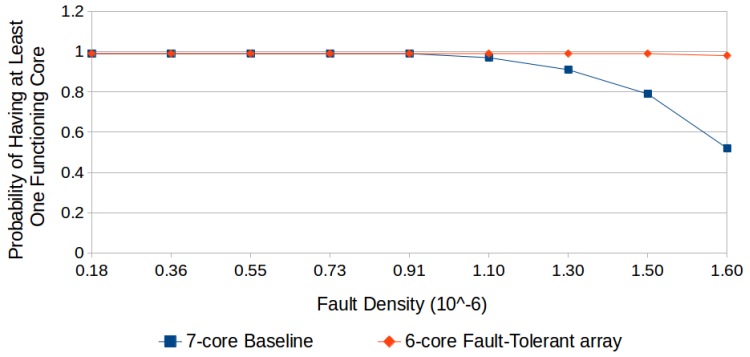

Fig. 12. Guaranteed availability. Case1: Baseline (7 cores), Case2: Fault-Tolerant array (6 cores).

D. Fault-tolerance analysis

Taking into account the area overheads of our multicore array, we can estimate overhead scalability and assess the fault-tolerance gains we achieve by paying the previously discussed costs. In our analysis, we assume a fixed die area A available to fit an array of 7 baseline cores (Case1) and an array of 6 FT coarse-grain cores (Case2) as proposed in this paper.

The fault tolerance of the above cases is evaluated using an analytical model in Matlab considering that the probability of a faulty substitutable unit (in each case) is $P_{sf}(N, P_f) = N * P_f$, where P_f is the probability of a single-transistor-device to be faulty and N is the number of devices per substitutable unit [14]. We perform our analysis considering hard-error densities of 1 fault per 0.6 to 5.5 million transistors (P_f being between $1.8*10^{-7}$ to $1.6*10^{-6}$). Assuming a uniform random distribution of faults, the above fault densities result in 10% to 90% probability to have 1 faulty transistor in area equal to that of a baseline core (approximately 550k transistors). As shown in Figures 11 and 12, we evaluate fault tolerance measuring the *average number of working cores* (denoted as average availability) and the *probability to have at least one functioning core* (denoted as guaranteed availability) under different fault rates for each of baseline and fault-tolerant systems.

For the average number of operational cores we can see that the degradation of the baseline case is quite steep while the FT MA case has a much smoother degradation because of the fault tolerance provided. The FT MA curve meets the baseline one at the fault density of about $P_f = 6*10^{-7}$ (1 fault per 1.6 million transistors). For higher fault densities, the FT system becomes more efficient than the baseline (even though the baseline system fits more cores in the area A) providing up to $\times 4$ more functioning cores. In terms of the probability to have at least one functioning core in the array, the FT case has a probability close to 1 across all fault densities. On the contrary, a significant drop for the baseline system can be observed for fault-densities higher than $P_f = 10^{-6}$ (1 fault per 1 million transistors) down to about 50% probability to deliver a functioning core.

V. Conclusions

In this paper we described the design of a fault-tolerant multicore array based on a reconfigurable pipelined interconnect. We described the architectural changes required to decouple the stages of a processor and allow them to be interchangeable. We further measured the performance, area, power and energy overheads of the added reconfigurability and evaluated the benefits of the improved fault tolerance. Our 4-core array can retain correct functionality with maximally 75% of the system being non-functional. Our architecture is, further, able to achieve up to $\times 4$ better availability and almost $\times 2$ higher probability of delivering at least one functioning core at fault densities above 1 fault per million transistors. The proposed micro-architecture requires about 1.8% more execution cycles for our benchmarks and has a critical-path delay overhead of 34.4%. A 4-core FT system implemented as described with pipelined interconnects is $\times 1.37$ slower compared to the baseline. In fault-free operation, our 4-core system has one third of the performance overhead compared to the StageNet due to our choice of reconfigurable interconnects, it exhibits 20.9% higher energy, on average, and 19.5% more area.

References

[1] A. Christou, *Electromigration and Electronic Device Degradation*, John Wiley and Sons, Inc., 1994.

[2] E. Wu, J. Suñé, W. Lai, E. Nowak, J. McKenna, A. Vayshenker, and D. Harmon, "Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate oxides," *Solid-State Electronics*, vol. 46, no. 11, pp. 1787–1798, 2002.

[3] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10 – 16, 2005.

[4] L. Spainhower and T. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective." *IBM Journal of Research and Development*, vol. 43, no. 6, pp. 863–873, 1999.

[5] D. P.Riemens, "Exploring suitable adder designs for biomedical implants," Master's thesis, CE, EEMCS dept., TU Delft, 2010.

[6] S. Gupta, S. Feng, A. Ansari, and S. Mahlke, "Stagenet: A reconfigurable fabric for constructing dependable cmps," *IEEE Trans. on Computers*, vol. 60, no. 1, pp. 5–19, 2011.

[7] A. Pellegrini, J. L. Greathouse, and V. Bertacco, "Viper: virtual pipelines for enhanced reliability," in *ISCA '12*, Jun. 2012, pp. 344–355.

[8] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults," in *PACT '08*, 2008, pp. 43–51.

[9] V. Vasilikos, G. Smaragdos, C. Strydis, and I. Sourdis, "Heuristic search for adaptive, defect-tolerant multiprocessor arrays," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, pp. 44:1–44:23, Mar. 2013.

[10] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *Int'l Symp. on Comp. Arch., ISCA '09*, 2009, pp. 93–104.

[11] D. Sylvester, D. Blaauw, and E. Karl, "Elastic: An adaptive self-healing architecture for unpredictable silicon," *Design Test of Computers, IEEE*, vol. 23, no. 6, pp. 484–490, 2006.

[12] A. Pan, O. Khan, and S. Kundu, "Improving yield and reliability of chip multiprocessors," in *DATE '09*, april 2009, pp. 490–495.

[13] R. Seepers, C. Strydis, and G. Gaydadjiev, "Architecture-Level Fault-Tolerance for Biomedical Implants ," in *Int'l. Conf SAMOS 2012* , jul 2012.

[14] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-based Computation.* , 2007, p. 834.