Dynamic Trade-off among Fault Tolerance, Energy Consumption, and Performance on a Multiple-issue VLIW Processor

Anderson L. Sartor, Pedro H. E. Becker, Joost Hoozemans, Stephan Wong, and Antonio C. S. Beck

Abstract—In the design of modern-day processors, energy consumption and fault tolerance have gained significant importance next to performance. This is caused by battery constraints, thermal design limits, and higher susceptibility to errors as transistor feature sizes are decreasing. However, achieving the ideal balance among them is challenging due to their conflicting nature (e.g., fault-tolerance techniques usually influence execution time or increase energy consumption), and that is why current processor designs target at most two of these axes. Based on that, we propose a new VLIW-based processor design capable of adapting the execution of the application at run-time in a totally transparent fashion, considering performance, fault tolerance, and energy consumption altogether, in which the weight (priority) of each one can be defined *a priori*. This is achieved by a novel decision module that dynamically controls the application's ILP to increase the possibility of replicating instructions or applying power gating. For an energy-oriented configuration, it is possible, on average, to reduce energy consumption by 37.2% with an overhead of only 8.2% in performance, while maintaining low levels of failure rate, when compared to a fault-tolerant design.

Index Terms—Adaptive processor, Energy consumption, Fault tolerance, Soft errors, VLIW

1 Introduction

As technology continues to evolve, more attention is paid to energy consumption and fault tolerance when designing new processors. While most of the embedded devices are heavily dependent on battery power, General-Purpose Processors (GPPs) are being held back by the limits of Thermal Design Power (TDP), highlighting the importance of reducing energy consumption. In addition, the need for fault tolerance on both space and ground-level systems is increasingly present in current processor designs. As the feature size of transistors decreases, their reliability is getting more compromised as they become more susceptible to soft errors [1]. Therefore, energy consumption and fault tolerance, together with performance, should be considered and balanced to address the aforementioned issues according to given design constraints.

On the other hand, current processors are designed to focus on one or, at most, two of these axes. Achieving the ideal balance among them is challenging, due to their conflicting nature. For example, let us consider fault tolerance, in which replication techniques are widely used to detect or mask faults during execution: while hardware-based techniques increase the power dissipation, software-based ones increase execution time; and both increase the total energy consumption. In the same way, reducing energy consumption will likely reduce performance; and improving performance will affect the energy consumption and possibly reduce fault tolerance.

In this scenario, we propose a new processor design capable of automatically and transparently adapting the execution of the application at run-time, considering performance, fault tolerance, and energy consumption altogether, in which the weight (priority) of each one is defined *a priori* by the designer. This is achieved by a novel decision module that dynamically controls the application's ILP to increase the possibility of replicating instructions or applying power gating when necessary. In more details, the system comprises four different modules, which were developed and coupled to the processor:

- A modified dual modular redundancy approach with rollback that exploits idle issue-slots (i.e., functional units that execute No-Operations (NOPs)) and dynamically duplicates instructions to improve the reliability of the system.
- A power gating mechanism that can be applied to the datapath to reduce both static and dynamic power. As idle issue-slots are completely turned off, they are not able to execute duplicated instructions, but they also reduce the sensitive area of the processor, influencing fault tolerance.
- A dynamic Instruction-Level Parallelism (ILP) controller that can reduce the parallelism (and therefore performance) at run-time. In this case, issue-slots are artificially freed by automatically moving operations to the next cycle, offering opportunities to maximize the power gating phases (as it will be explained, power gating demands that an issue-slot must be turned off for a minimum period, so the gains in power overcome

- *A.L. Sartor, P.H.E. Becker, and A.C.S. Beck are with the Institute of Informatics, Universidade Federal do Rio Grande do Sul, BR. E-mail: {alsartor, phebecker, caco}@inf.ufrgs.br*
- *J. Hoozemans and S. Wong are with the Computer Engineering Laboratory, Faculty of EEMCS, Delft University of Technology, The Netherlands. E-mail: {j.j.hoozemans, j.s.s.m.wong}@tudelft.nl.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2760299, IEEE Transactions on Multi-Scale Computing Systems

its costs).
- Considering that decreasing performance will affect power, which indirectly influences fault tolerance, a novel decision module to balance them is necessary. This module dynamically detects phases as the application is executed to decide when and which hardware parts will be shut down; and when the ILP reduction is applied. It can be configured by the designer to prioritize one or more axes.

By using this set of techniques in line with the decision module, it is possible to exploit the trade-off among these axes at a fine granularity (instruction-level), consequently allowing a fine tune of the application's execution. The ρ-VEX Very Long Instruction Word (VLIW) processor [2] was used as target architecture for this study. VLIW processors are representative examples of current VLIW multiple-issue architectures (e.g., Intel Itanium [3], Trimedia CPU64 [4], and TMS320C6745 [5]). These processors are able to deliver high-performance at a low energy consumption cost, as they do not rely on runtime dependency-checking mechanisms for scheduling the instructions that will be executed in parallel [6]. Therefore, VLIW processors are suitable options for safety-critical systems in several fields, such as automotive, space, and avionics. Notably, a VLIW multi-core was recently used by NASA in its Mars rover for image processing [7].

The remainder of this work is organized as follows. Related work comprising the aforementioned axes are discussed in Section 2. Section 3 describes the proposed techniques for each of the axes and their implementation. Next, in Section 4, the results are discussed. For that, a fault injection campaign was performed in several benchmarks. Finally, Section 5 concludes this work and discusses future directions.

2 Related Work

2.1 Fault Tolerance

Several works have been proposed for the detection and correction of transient faults in multiple-issue processors (e.g., VLIW and superscalar). These works aim to improve the fault tolerance of the target system, typically based on redundancy, which may be implemented in software, hardware, or both. Next, some of these techniques will be discussed.

2.1.1 Non-adaptive Fault Tolerance Approaches

Dual Modular Redundancy (DMR) based on checkpoints with rollback was used by [8] and [9] to detect and correct errors. Whenever an error is detected, the state in which the execution was correct is recovered. Therefore, the latency to detect the error on these approaches will vary according to the periodicity of the checkpoints (i.e., when a new checkpoint must be made). These approaches were implemented in software.

A hardware-based replication with rollback mechanism for superscalar processors is proposed in [10]. In this approach, the application is replicated into two or more threads, and the computed results are checked before commit. If any inconsistency is found, the rollback is activated, the re-order buffer is flushed, and the execution restarts by fetching once more the next-PC from the last committed instruction. These threads have tightly coupled executions (both must execute at the same time) and, consequently, each type of functional unit used by the main thread must be duplicated so that the replicated thread may execute concurrently.

In [11] and [12], the authors propose a software-based redundancy based on Duplication With Comparison (DWC) for VLIW datapaths aiming to reduce the performance overhead by using the idle functional units. However, these techniques still present huge performance degradation and increase code size, as they are implemented in software. Authors in [13] propose an optimization to the DWC's generated code by reducing the impact of the basic block fragmentation caused by the check instructions, having lower, but still not negligible, performance degradation than the previous two techniques.

Another common approach is to triplicate hardware components and use a majority voter to mask the faults (Triple Modular Redundancy (TMR)), as implemented in [14] and [15]. In these cases, they only triplicate the functional units of a VLIW processor rather than the entire processor; therefore, it is possible to reduce area and power dissipation costs. In [14], the authors proposed the Reduced TMR, in which both hardware and software needed to be changed. If the two instructions (main and duplicated) compute different results, the instruction is executed a third time.

In [16], the combinational logic parts of a MIPS processor are triplicated. However, only two copies run in parallel, while the third one uses time redundancy in the case of an error to vote the correct result. A state machine controls which two combinational logic modules will be active, and a control module manages the comparator, pipeline register, demultiplexer, and multiplexer. In order to re-compute the operation in case of an error (third execution), shadow latches are used to hold the last fault-free state of the pipeline. Then, the comparator votes the three computed results.

Watchdog processors [17] are also used for fault tolerance purposes, as they execute concurrently to the main processor and compare the outputs from the main processor with their own (pre-computed or concurrently computed). Examples of watchdog processors are discussed next.

DIVA [18] proposes to increase the reliability of a superscalar processor by augmenting the commit phase of the pipeline with a checker unit (watchdog processor). This checker is a simple in-order processor that does not have any mechanism to speed up computation (e.g., predictors, re-namers and dynamic schedulers). The checker will verify and commit the results if the computation is correct, and flush the computation and restart the processor in case of an error. The checker is considered to be more robust than the other parts of the circuit, and the verification cost of the checker is lower than verifying traditional processor design due to its simplicity. In addition, the authors discuss the buffering of results in order to deeply pipeline the checker unit, which will permit implementations with large time margins and large transistors (more resistant to transient faults and radiation interference).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2760299, IEEE Transactions on Multi-Scale Computing Systems

Other approaches use redundant threads as watchdog processors [19], [20]. SHREC [21] proposes an approach for asymmetric re-execution, similar to DIVA. In addition, it allows threads to be replicated and staggered. Hence, the difference in the execution progress between the leading and trailing (replicated) thread hides cache-miss latencies and allows the leading thread to provide branch prediction information to the trailing thread. The trailing thread instructions are only checked using input operands produced by the leading thread, avoiding bottlenecks in the issue queue and reorder buffer. Also, the functional units are shared, unlike DIVA.

All previous works discussed in this section propose static fault-tolerance techniques, which are used for the single purpose of improving the reliability of the system. On the other hand, the proposed system focus on using several techniques to dynamically trade-off the axes of fault tolerance, performance, and energy consumption.

2.1.2 Adaptive Fault-Tolerance Approaches

To provide adjustment of the fault tolerance according to system requirements and application's execution, several works have been proposed. The authors in [22], propose a framework that switches between different fault-tolerance techniques (that include TMR, DWC, high-performance - and no fault tolerance) depending on a priori knowledge of the environment, external events, or application-triggered events. However, this approach is static and limited to FPGAs only, as the hardware needs to be reconfigured. Therefore, it could not be used in our processor (that is dynamic and independent of technology).

An adjustable checkpointing interval was proposed in [23], which is adjusted during the execution based on the occurrence of faults and the available slack. An offline preprocessing based on linear programming is used to determine the parameters for the online checkpointing process, which presents some detection latency. On the other hand, all decisions in the proposed processor must be taken at run-time, with a zero-latency detection. The authors in [24] propose to replicate only critical instructions of the application. This is done by first identifying which are the critical variables, extracting the critical code sections and finally instrumenting the code with check instructions. A dynamic code duplication mechanism to detect faults on VLIW processors was proposed in [25], it depends on the compiler to instrument the code and to indicate which instructions will be duplicated during run-time. However, besides being able to only detect faults, not correct them; it also involves preprocessing as the application needs to be profiled or it depends on the compiler to instrument the code.

Aaron [26] tackles software and hardware errors by using diversified software components in the CPU spare cycles. Eight methods are used in this diversification, including SWIFT [27], a software-based fault-tolerance approach that duplicates instructions and registers. The system load is estimated, and the scheduler chooses the best variant to fill the spare cycles (e.g., executing a reliability-oriented variant, which takes longer to execute, but is able to provide fault tolerance). However, the software variants for the dynamic Design Space Exploration (DSE) still need to be created before execution (i.e., statically), with a huge overhead on memory occupation.

All these previous works require preprocessing, which make them not suitable to be included in an adaptive and transparent processor. Moreover, the hardware-based mechanisms are usually technology-dependent, while the software-based ones present a huge overhead in performance or are applied in thread-level, not allowing the exploitation at finer granularities (instruction-level).

2.2 Energy-aware Fault Tolerance

Few works propose to apply energy optimization mechanisms to reduce the cost of fault-tolerance approaches. The authors in [28] propose to increase fault tolerance and minimize the energy consumption in a no-fault scenario, by scheduling and applying Dynamic Voltage Scaling (DVS) on a heterogeneous distributed time-triggered system, in which the processes are statically scheduled. The transient faults are tolerated by re-executing the whole process, which is very expensive in terms of performance. The proposed processor works at the instruction level amortizing performance costs, and it is completely transparent to the software layer (i.e., no software or operating system modifications are needed).

In [29], the authors combine procedure call duplication with statement duplication to reduce the energy consumption overhead when compared to the duplication of every program instruction. This approach is only able to detect errors, not correct them; and the overhead in energy consumption is still very significant since it works at the method level. Moreover, software modifications are also necessary.

2.3 Fault Tolerance, Energy Consumption, and Performance Trade-off

Only a reduced number of works considers all three axes (fault tolerance, energy consumption, and performance). Tricriteria Scheduling Heuristics (TSH) [30] proposes an offline scheduling heuristic that produces a static multiprocessor schedule, based on a given software application graph and a given multiprocessor architecture (homogeneous and fully connected). In order to increase reliability, the instructions are replicated; and to reduce the energy consumption, Dynamic Voltage and Frequency Scaling (DVFS) is applied. A greedy scheduling algorithm takes as input the algorithm and architecture graphs, power and reliability constraints, and the execution time of the operation considering the maximum frequency to meet the reliability and energy requirements and minimize the scheduling length. All the variables of the design space exploration are defined at design time, which means that the whole process is neither automatic nor transparent to the developer. On the other hand, the proposed processor is able to dynamically balance the axes, without the need of any prior knowledge of the application nor extra tools to generate the options for design space exploration, making it fully automatic and transparent to the designer or user. The designer also has the freedom of easily choosing which is the most important parameter (e.g., make the processor more fault tolerant or more energy oriented).

3 Proposed Adaptive Processor

The proposed adaptive processor (depicted in Fig. 1) comprises four modules: fault tolerance (instruction duplication), energy optimization (Power Gating (PG) module), ILP control, and decision module. They all work at run-time. The P_0-P_7 represent the *pipelanes* (i.e., the issue-slots) of an 8-issue VLIW processor. The pipeline view comprising the Fetch, Decode, Execution stages, and Write-back depicts how each pair of pipelanes is compared by the checker when instructions are being duplicated. For instance, pipelane P_0 is compared to P_4 in order to detect the errors, and the rollback mechanism is responsible for correcting it (which will be explained in detail later). Note that the modified version of the processor does not have dedicated pipelanes for executing duplicated instructions. Instead, existing pipelanes can be used to execute program instructions and duplicated instructions when required, avoiding the huge area and power overheads.

The fault-tolerance mechanism dynamically duplicates instructions, whenever there is a free issue-slot and such issue-slot is turned on, to exploit idle hardware and improve the reliability of the circuit. The energy optimization module (PG module from Fig. 1) shuts down functional units or entire issue-slots to save energy. The ILP control mechanism dynamically reduces the ILP of the application by splitting bundles (VLIW instruction words) into two in order to maximize a given power gating phase (a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency [31]), consequently saving more energy. All these mechanisms are controlled by the decision module, which is responsible for two things: decide when and which parts of the hardware should be shut down based on a given constraint defined by the designer; and when the ILP should be reduced to extend a given idle phase. As will be shown, from these two operations it is possible to balance performance, energy consumption, and fault tolerance.

As already mentioned, the current design does not require any application modification, as all the proposed techniques were implemented in hardware. Modifying and recompiling the binary code may not be a trivial task, leading to incompatibility with future processors and losing backward compatibility. Hence, any compiler that supports the VEX instruction set architecture may be used to compile the applications. Next, the processor configuration used in this work will be discussed, followed by each of the modules that comprise the adaptive processor.

3.1 Processor Organization

The ρ-VEX softcore VLIW processor [2], implemented in VHDL, is used in this work. It has a Load/Store Harvard architecture with a five-stage pipeline: *Fetch*, *Decode*, *Execute 1*, *Execute 2*, and *Write-back*. The issue-width (e.g., 2, 4, or 8), type and organization of functional units, and register file size may be configured during design time. Each pipelane may contain different Functional Units (FUs) from the following set: Arithmetic Logic Unit (ALU) (always present), multiplier, memory, and branch units.

In this work, it is used the 8-issue version, and it has eight ALUs, four multipliers, two memory units, and two branch units (one branch and one memory unit only execute duplicated instructions, as it will be explained later). The register file contains 64 general-purpose registers of 32 bits.

Several compilers may be used to compile VLIW code for the ρ-VEX processor: HP VEX compiler, GCC VEX, LLVM, and Open64, and the compiler is responsible for scheduling independent instructions to be executed concurrently. However, in several cases, it is not able to fill all slots of the bundle with independent instructions [32]. The solution is filling the unused slots with NOPs. These NOPs require memory bandwidth to be fetched, potentially increasing cache misses, which would result in performance degradation and extra energy consumption.

In order to amortize such costs, several techniques have been proposed to remove these NOPs: compressed encoding for VLIW instruction [33]–[36]; instruction template bits [3], [37]; and stop-bits [38]–[41]. Even so, the functional units of the issue slot responsible for executing the NOP (whether it was removed from code or not) will still be idle, which potentially allow this idle hardware to be exploited in a more efficient way: executing duplicated instructions for fault tolerance, or shutting down these functional units to save energy (the proposed processor is able to do both).

3.2 Fault-Tolerance Mode

When the fault-tolerant mode is active, the idle pipelanes are used to execute duplicated instructions whenever possible. After fetching an instruction word, each pipelane receives one instruction for decoding and further execution. We have modified this process so that the pipelane can either receive a regular program instruction (no duplication); or a duplicated instruction from another pipelane in case the current pipelane is idle (i.e., it would execute a NOP). For that, at each cycle it is verified whether there are idle pipelanes or not. As instructions are duplicated after the data has been fetched from memory, no additional memory accesses for the duplicated instructions are required.

To keep the overhead low (area and delay), the duplication pairs are placed in the following order: *pipelane 0* with *pipelane 4*, *pipelane 1* with *pipelane 5*, and so on. Therefore, the issue-slots are combined in a way that the first four pipelanes are compared with the four last ones. This approach efficiently exploits the scheduling mechanism of the HP VEX compiler, which always schedules the instructions starting from the lower issue-slots (from *0* to *7*). An additional memory and branch unit must be added to the core to allow duplication of all instructions. These two specific extra functional units are used only to execute duplicated instructions (i.e.: they cannot be used for regular instructions), since the ρ-VEX does not support more than one memory or branch operations per cycle.

A checker compares the results (i.e., all output signals) of the pipelanes that are executing duplicated instructions (e.g., arithmetic operations, jump address of a branch, or the values of a memory operation are checked) to detect errors. The destination register, the register file's and memory's write enable signals are also compared. When an error is detected, a rollback mechanism is triggered to correct it by executing the last uncommitted instruction again (which is also depicted in Fig. 1). The Program Counter (PC) for the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2760299, IEEE Transactions on Multi-Scale Computing Systems

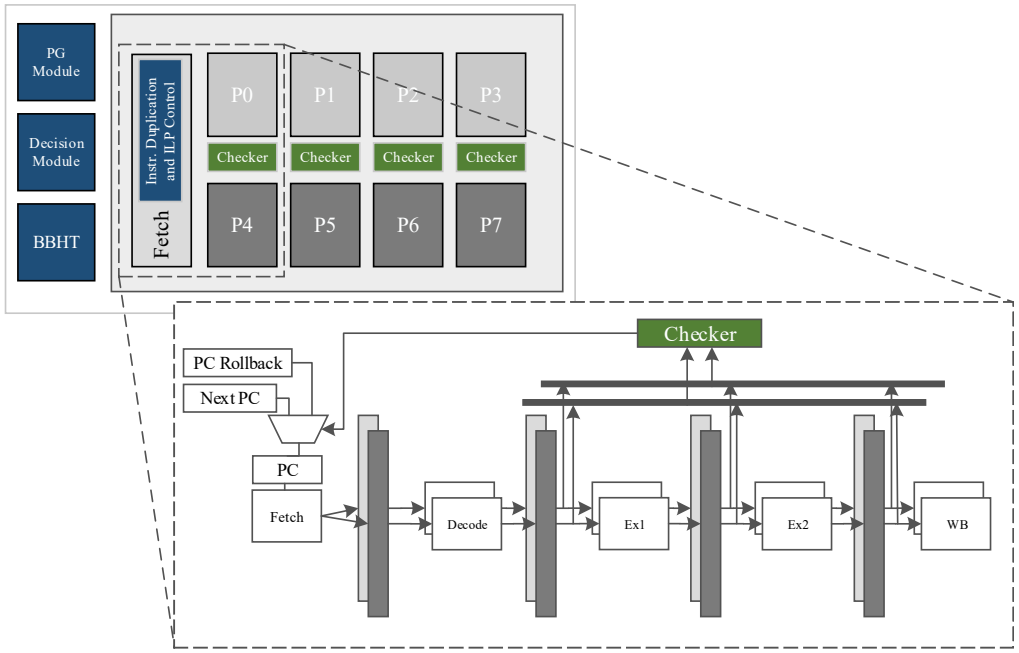

Fig. 1. Adaptive Processor

rollback is stored in a register and, in the case of an error, this stored PC overwrites the current PC.

As the memory and register file were not modified in the meantime (between the rollback PC and the current PC), the pipeline is simply flushed, and the writing to the memory and register file are blocked, avoiding memory corruption. Once the rollback PC is loaded, the instruction corresponding to that PC is fetched again, and the execution resumes from that point. Therefore, the process has a fixed cost of five cycles to refill the pipeline, which is negligible considering the total number of cycles of an application, and that this cost is only paid in case of an error. Both the checkers and the rollback mechanism do not affect the critical path of the processor, as they operate in parallel to the rest of the processor. The memory and the register file are considered to be protected with Error Correction Code (ECC).

3.3 Energy Optimization Mode

The PG is applied to the functional units of the processor. When all functional units from a given pipelane are turned off, the whole pipelane is also turned off to reduce energy consumption even more. By shutting down idle blocks of the circuit, PG reduces both static (leakage represents 20% to 30% of the total power dissipation of modern processors [42]) and dynamic power. A header transistor is used to turn off the supply voltage to the circuit block, which creates a virtual *Vdd*. A sleep signal controls this transistor so the *Vdd* can pass through when the circuit is active (virtual *Vdd* = *Vdd*) and gate the *Vdd* when idle (virtual *Vdd* = 0V). As this extra transistor is used, this comes at the cost of performance penalty and power overhead as the power lines are not able to be charged and discharged in one clock cycle.

To cope with this additional cost, the module that is being power gated must remain shut down for at least the break-even time. The break-even is the time necessary to compensate the additional energy consumption of the gating transistor. Considering the technology parameters used in this work, the wake-up time of this transistor is three cycles [43]. After this point, the longer the circuit remains turned off, the more energy will be saved [43]. Therefore, to minimize these overheads, the power gating mechanism is applied to the granularity of Basic Blocks (BBs) (i.e., a code sequence with no branches in except to the entry; and no branches out except at the exit). Whenever a new BB starts to execute, its instructions are analyzed so it can be evaluated which functional units are required for the execution of that BB. Based on this evaluation, a power gating configuration is saved for that basic block for future reuse (this is controlled by the decision module and will be discussed in detail in Section 3.5).

When a given functional unit is turned off during the execution of a BB and the next BB needs this functional unit at its very beginning, it would take three cycles so it is completely active. In this case, the processor would have to wait for the FU to be active (stalling the execution). On the other hand, if the FU is only needed after the wake-up time, it would be ready by the time the given instruction that needs it starts executing, and there would not be a performance overhead.

Fig. 2a depicts an example of code execution of a basic block on the regular unprotected 8-issue processor (same processor organization as shown in Fig. 1). In this figure, the C_1 to C_n depict the cycle *1* to *n* of the execution of such BB, and in each cycle, a bundle (i.e., a VLIW instruction word) is executed. Note that there are NOPs in some bundles due

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2760299, IEEE Transactions on Multi-Scale Computing Systems

to of lack of parallelism, which will waste energy to be executed and will not provide any meaningful computation. Therefore, the proposed techniques exploit such NOPs to improve reliability and reduce energy consumption.

Fig. 2b presents a scenario that only the Fault Tolerant and Energy Optimization modules are active, without the ILP control mechanism (which will be explained in the next subsection). In this approach, it is possible to use those functional units that were idle to execute duplicated instructions in the pipelanes P_4 and P_5 (increasing fault tolerance) and turn off the idle pipelanes P_6 and P_7 (saving energy). In the case of pipelanes P_4 and P_5, at least one functional unit is used in this BB (at C_2), so the whole pipelane could not be shut down, due to the power gating costs. On the other hand, pipelanes P_6 and P_7 may be completely turned off as only NOPs are executed during the whole BB.

3.4 ILP Control Module

A power gating phase may be interrupted by a few instructions: Fig. 2b shows a BB in which a given pipelane is idle, except for one instruction (i.e., I_4 or I_5 at C_2). In this case, according to the previous discussion on the break-even, power gating could not be applied to this pipelane, therefore losing significant opportunities for energy savings. The ILP Control module (which is controlled by the decision module, explained in the next subsection) is responsible for splitting bundles in a dynamic fashion to increase the length of a potential power gating phase, so more energy savings can be reached. It also guarantees correct program execution by detecting data dependent operations: instructions that write to a certain register in the first half of the instruction word and read from the same register on the second half would introduce a read after write hazard when split, resulting in wrong computation. These hazards are detected in hardware, and these instructions are not split, preserving program correction.

Let us consider the previous example presented in Fig. 2b again: the pipelanes P_4 and P_5 would remain turned on due to the execution of instructions I_4 and I_5 at C_2 (and therefore they would be used for replication). The presence of the ILP Control Module adds another alternative: by applying the ILP reduction (Fig. 2c), these same pipelanes can now be turned off (since they now reach the break-even constraints), maximizing the power gating phase. Therefore, while the energy consumption and the sensitive area will reduce, it also restricts the duplication of some instructions. It is evident that there is a trade-off in the number of instructions that can be replicated; the pipelanes that can be power gated; and the parallelism that can be lost (and therefore, performance) so even more pipelanes can be turned off. For that, a decision module, which will be described next, was developed.

3.5 Decision Module

The current design operation flow is depicted in Fig. 3 and it is explained in detail next:

1) Whenever there is an opportunity (i.e.: a NOP is being executed on any pipelane of the upper half of the VLIW word), the correspondent instruction will be duplicated.

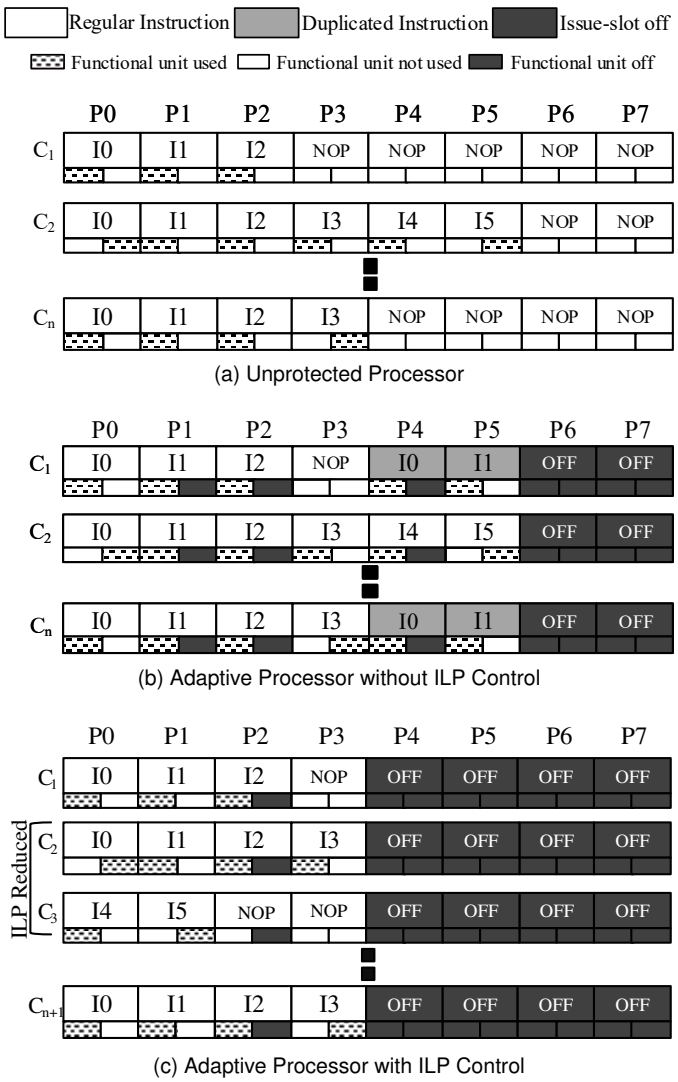

Fig. 2. Adaptive Execution Example

2) Whenever a program phase that respects the break-even constraints is detected in a given pipelane or functional unit, power gating will be applied. This will influence the previous item (duplication), since fewer pipelanes will be available for duplication.
3) The ILP control can split instruction words, as already explained. This open more opportunities for power gating, which will positively impact (2) and, by consequence, negatively influence (1).

The fault-tolerant mode (1) is automatic and always on. For (2), the decision module detects, at run-time, which parts of the hardware can be shut down; and, for (3), when and how the ILP reduction shall be applied.

For (2), the decision module detects program phases as the application is executed. This is done by using a similar approach as [44], in which the information regarding the hardware utilization of each basic block is saved for future reuse; in our case, in a direct mapped memory. This memory is called Basic Block History Table (BBHT), which is indexed by the PC of the first instruction of the BB. Each entry of this memory contains one bit per functional unit that indicates if it will be turned on or off (i.e., power gated) next time

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2760299, IEEE Transactions on Multi-Scale Computing Systems

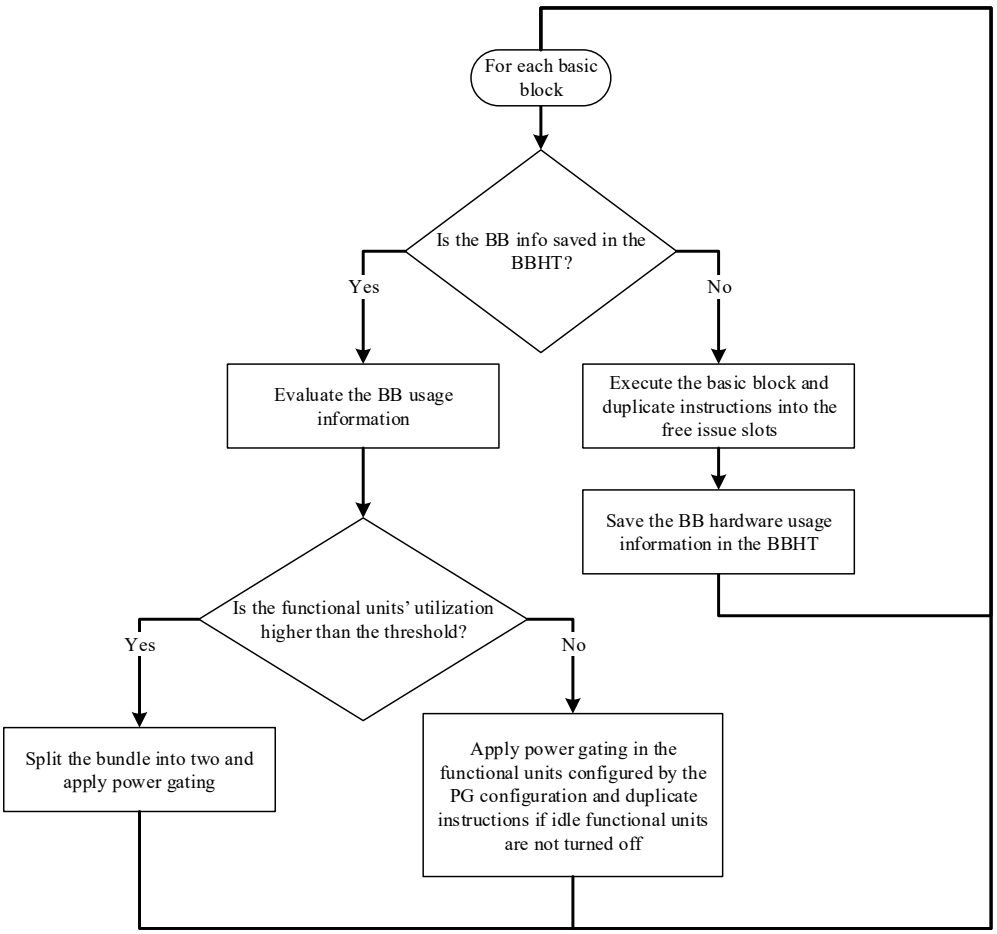

Fig. 3. Processor's Execution Flow

the same BB is executed. If the functional unit was used at least one time during the execution of the BB, it means it must be turned on next time the same BB is found, so the correspondent bit is set to *0*; otherwise, it is set to *1*. As there are 16 functional units (two per pipelane), 16 bits are needed for this purpose. Adding to them, there are 32 bits of the PC; and two control bits, in a total of 50 bits per entry (called PG configuration).

Every time a BB ends its execution (reaches a branch instruction), the PC of the next BB to be executed is searched in the BBHT. In case the configuration for the next BB is not found (because that BB was never executed before or because it was replaced), the decision module will build its PG configuration. If the entry is found, the decision module will turn off the functional units according to the correspondent bits in the PG configuration.

The ILP control (3) is also managed by the decision module, based on a threshold. This threshold represents the utilization ratio of a given functional unit during the execution of the BB, defining whether instructions will be split into two or not. For instance, if the threshold is configured to 50%, a given functional unit on the high part of the issue-width must be used more than 50% so it will remain active during that BB, otherwise (if it is used less than 50%), the instructions will be split into two cycles and the correspondent functional units will remain turned off to maximize the power gating phase.

Therefore, the higher the threshold value is, the more VLIW instructions will be split. By consequence, there will be more program phases so power gating can be applied, and fewer instructions will be replicated. By adjusting the threshold to its minimum value, fault tolerance and performance will be prioritized over energy consumption. The very same opposite reasoning can be used for high threshold values: energy optimization will be given more weight (because more hardware would be artificially freed so it could remain turned off). Alternatively, one could also totally disable the power gating to focus only on the compromise between fault tolerance and performance, by controlling the ILP module to vary the number of duplicated instructions (this setup will also be explored in the results).

By configuring the threshold, which is done at design time, the designer can tune the processor to be more performance, energy optimization, or fault-tolerance-oriented. This tuning highly depends on the target application's and system's requirements. For instance, one may require hav-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2760299, IEEE Transactions on Multi-Scale Computing Systems

ing an application that does not spend more than X Joules, executes in less than Y cycles, or has a certain degree of reliability. These constraints can be met by adjusting the threshold value and testing the application, or enabling/disabling a certain mechanism. In this work, the threshold for the ILP control mechanism was evaluated with its maximum and minimum values, so the trade-off between all axes can be evaluated.

4 Results

4.1 Methodology

By having access to the hardware description of the ρ-VEX processor, we are able to measure performance at cycle-level; measure energy consumption and area after synthesis; and inject faults at the gate-level. The synthesis tools used were: Cadence Encounter RTL compiler to obtain power dissipation and Application-Specific Integrated Circuit (ASIC) area, using a 65nm CMOS cell library from STMicroeletronics (the operating frequency was set to 200MHz). CACTI-P [45] was used to estimate the area and energy consumption of the BBHT, which was set to have 256 lines of 64 bits each (the closest power of two of 50 bits), one write and one read port, with ECC enabled. We focus on protecting the issue-slots of the processor (which occupy about 45% of the total core area) since the register file (which occupies the rest) can be protected with parity [46] or ECC [47]. For the area and energy consumption evaluation, all additional modules that were implemented are also taken into account, which includes the duplication and ILP control modules, checkers, PG and decision modules, and BBHT.

4.1.1 Benchmarks

The benchmark set is composed of 12 applications from the WCET [48] and Powerstone [49] benchmark suites. The HP VEX compiler was chosen because it is more stable and robust than the GCC VEX compiler. A soft-float library [50] was used for floating-point operations, and the original input data from the benchmark suite was used. Each of the chosen benchmarks is discussed next.

- *Adaptive Differential Pulse-Code Modulation (ADPCM)*: completely structured code (i.e., no conditional branches, no exit from loop bodies) and contains loops;
- *Discrete Fourier Transform (DFT)*: uses arrays and nested loops;
- *Engine*: engine control application. It has sequences of condition statements and loops;
- *Expint*: performs series expansion for computing an exponential integral function. Contains nested loops and arrays;
- *JPEG*: JPEG 24-bit image decompression standard. Uses arrays and nested loops;
- *LU Decomposition (LUDCMP)*: performs calculations based on floating point arrays with the size of 50 elements;
- *Matrix multiplication*: multiply two 20x20 matrices. It has multiple calls to the same function, nested function calls, and triple-nested loops;
- *Minver*: inversion of floating point matrix. Floating value calculations in 3x3 matrix with nested loops;
- *POCSAG*: POCSAG paging communication protocols. Contains loops and several condition statements;
- *Qurt*: root computation of quadratic equations. The real and imaginary parts of the solution are stored in arrays;
- *Sums*: recursively executes multiple additions on an array;
- *x264*: contains arrays, matrices, and loops to calculate a sum of absolute differences of the H.264 video encoding standard.

4.1.2 Fault Injector and Fault Model

Subsequently, the motivation to use a detailed simulation to inject faults is discussed, and the fault model used in this work is presented. In simulation-based fault injection, the hardware (target system) is simulated, and the faults are injected into the simulation. A simulation-based approach has several advantages, such as: low cost; can be applied at design time, before deployment (so the designer can change the circuit in case its reliability does not meet a certain criteria); high controllability of the fault injection; no need for special equipment or hardware components; and flexibility to choose between different fault models. Both high-level and low-level simulations are possible with this method.

Fault injectors can be used with high-level simulators (which are fast) by usually changing the value of registers during the execution of an application (e.g., the architectural registers of a processor). However, it leads to highly inaccurate results: for example, given a complex superscalar processor, the architectural registers (those visible to the programmer) represent a small fraction of all registers present in the system (i.e., the pipeline registers, instruction queue, branch history buffer and so one will not be affected). On the other hand, when one considers low-level simulators (e.g., Register-Transfer Level (RTL) simulation), faults can be injected in the hardware module's low-level signals, which has a high controllability and accuracy. However, detailed RTL simulations present extremely high simulation times.

In order to maintain the accuracy of detailed RTL simulations and accelerate the simulation process, we developed a hybrid fault injection framework. It is able to speed up the fault injection campaign by using two different simulation levels: a detailed one (low-level) to inject the faults accurately, and a behavioral one (high-level) to speed up the process; automatically switching between both at simulation time.

The fault model that was used in this work is described next:

- *Fault type*: The injected faults are transient and comprise a Single Event Transient (SET) that will affect a signal from the design. Note that a SET may cause a Single Bit Upset (SBU) or Multiple Bit Upset (MBU).
- *Injection place*: The faults are injected into any atomic signal of the target module. All internal and low-level signals are considered (39,178 signals).
- *Injection instant*: Follows a uniform probability function in the range between zero and t equal to the expected execution time of the application without faults.
- *Fault duration*: To increase the likelihood of the SET to be captured by a flip-flop, the signal is forced for the duration of one clock cycle.

One fault is injected per application's execution, due to the extremely low probability of more than one fault affect the same execution of a given application. The total number of injected faults was 4 million (so there was the same number of application executions). Compared to the number of signals from the target module, the number of injected faults provides high confidence for the reliability evaluation. Both data and control flow failures are verified: the former, comparing the memory dump to the golden copy and the latter, with the number of executed cycles.

4.2 Failure Rate, Performance, and Energy

In this section, the results are compared in terms of failure rate, performance, energy consumption, area, and Energy-Delay-Failure Product (which will be explained in detailed later). Table 1 presents the six designs that were evaluated and which techniques each design uses, as discussed next.

1) ***Unprotected*** : Baseline processor, without fault tolerance and power gating mechanisms (prioritizes performance);
2) ***Unprotected with PG*** : Baseline processor, without fault tolerance and with power gating (prioritizes energy consumption);
3) ***Duplication with ILP control*** : fault tolerant only, without power gating. It duplicates instructions whenever possible and splits bundles to increase fault tolerance (prioritizes fault tolerance);
4) ***Adaptive without ILP control*** : fault tolerant and with power gating (weighs fault tolerance and power gating equally);
5) ***Adaptive with ILP control*** : fault tolerant, with PG and with ILP control (weighs more energy consumption than fault tolerance);
6) ***Triple Modular Redundancy (TMR)*** : included in the experiments for comparison purposes only. The processor is triplicated and the results are voted in order to mask the faults, using the traditional TMR technique (it is only fault tolerant).

The absolute results in terms of performance, energy consumption, and failure rate for these designs are presented in Table 2, and the relative numbers are explained next.

4.2.1 Performance

Fig. 4a presents the relative performance overhead of the five versions when compared to the ***Unprotected***. As aforementioned, when power gating is being used, and a given BB starts its execution, a given functional unit may not be completely turned on and ready to execute instructions. When this occurs, the processor must be stalled, incurring in performance overhead. This is the case for the ***Unprotected with PG*** and ***Adaptive without ILP control***. Both present an average performance overhead of 9.5%, varying from 0.6% to 33.9%, as these designs do not have the ILP control mechanism activated. When the ILP control is activated in the ***Adaptive with ILP control*** version, more energy is saved as the power gating phases will be extended, and this design incurs in an average performance overhead of 13%. The ***Duplication with ILP control***, which does not use PG, has an average overhead of 4.4%, varying from zero to 11.4%. Finally, the ***TMR*** maintains the performance of the unprotected processor, with a huge overhead in energy, which will be discussed next.

TABLE 1
Evaluated Designs and Their Techniques

Design	FT technique	PG	ILP Control
Unprotected	None		
Unprotected with PG	None	✓	
Dupl. with ILP ctrl	Dupl. with rollback		✓
Adaptive w/o ILP ctrl	Dupl. with rollback	✓	
Adaptive with ILP ctrl	Dupl. with rollback	✓	✓
TMR	Triplication		

4.2.2 Energy and Area

Fig. 4b shows the relative energy consumption considering the ***Unprotected*** version as baseline once more. When applying power gating to the baseline version (2), which does not have fault-tolerance mechanisms, the average energy consumption is reduced by 26.3%, varying from 4.3% to 35.5%. As aforementioned, applying fault-tolerance mechanisms increase the energy consumption of the design, resulting in an average overhead of 200% in the ***TMR*** (since the processor is triplicated), and 86.9% (from 83.4% to 94.5%) in the ***Duplication with ILP control*** version.

When compared to the ***Duplication with ILP control***, the ***Adaptive without ILP control*** is able to reduce the energy consumption by 34%, on average; and by 37.2%, in the ***Adaptive with ILP control*** (from 17% to 44%). Therefore, the ***Adaptive*** approaches are able to significantly reduce the energy consumption when compared to a protected version of the processor. Even when compared to the ***Unprotected*** design (that does not have any fault-tolerance mechanism), the ***Adaptive*** versions consume, on average, only 23.3% and 17.4% more energy, for the version *without* and *with* ILP control, respectively (while the non-fully adaptive ***Duplication with ILP control*** consumes 86.9% more).

The additional modules of the proposed adaptive design incur an area overhead of only 15.05% compared to the unprotected version, which is low considering that we have mechanisms for fault tolerance, energy optimization, and performance management all implemented in a single processor.

4.2.3 Failure Rate

Fig. 4c depicts the failure rate of the chosen benchmarks for each design after the fault injection campaign. The ***Unprotected with PG*** maintained the failure rate of the ***Unprotected*** design, as no fault-tolerance mechanism is applied. Therefore, the probability of a failure occurring remains the same. However, an application being executed with PG will take more time to be executed, which will increase the time it is exposed to energetic particles. As expected, all the other versions present lower levels of failure rate than the ***Unprotected***. On average, the ***Unprotected*** has 2.63% of failure rate (which is similar to other commercial processors, for instance, the IBM Power6 microprocessor [51]); the ***Duplication with ILP control***, 0.20%; the ***Adaptive***

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2760299, IEEE Transactions on Multi-Scale Computing Systems

TABLE 2
Performance, Energy Consumption, and Fault Tolerance Comparison

Benchmark	Metric	1) Unprotected (baseline)	2) Unprotected with PG	3) Duplic. with ILP control	4) Adaptive w/o ILP control	5) Adaptive w/ ILP control	6) TMR
ADPCM	Perf. (cycles)	568	596	633	596	630	568
	Energy cons. (J)	4.83E-08	4.62E-08	9.39E-08	6.88E-08	6.32E-08	1.45E-07
	Failure rate (%)	3.66	3.66	0.59	0.89	0.90	0.00
DFT	Perf. (cycles)	32,575	41,630	32,979	41,630	42,024	32,575
	Energy cons. (J)	2.23E-06	1.45E-06	4.09E-06	2.44E-06	2.34E-06	6.68E-06
	Failure rate (%)	2.68	2.68	0.15	0.33	0.36	0.00
Engine	Perf. (cycles)	691,437	724,482	695,852	724,482	723,553	691,437
	Energy cons. (J)	4.66E-05	3.08E-05	8.54E-05	5.45E-05	5.07E-05	1.40E-04
	Failure rate (%)	1.80	1.80	0.24	0.27	0.34	0.00
Expint	Perf. (cycles)	9,097	9,305	9,257	9,305	9,509	9,097
	Energy cons. (J)	6.36E-07	4.10E-07	1.17E-06	6.53E-07	6.45E-07	1.91E-06
	Failure rate (%)	2.37	2.37	0.07	0.09	0.10	0.00
JPEG	Perf. (cycles)	1,448,615	1,620,785	1,572,092	1,620,785	1,745,014	1,448,615
	Energy cons. (J)	1.13E-04	8.02E-05	2.10E-04	1.29E-04	1.18E-04	3.38E-04
	Failure rate (%)	2.15	2.15	0.12	0.22	0.22	0.00
LUDCMP	Perf. (cycles)	44,558	47,222	47,545	47,222	50,076	44,558
	Energy cons. (J)	3.13E-06	2.25E-06	5.93E-06	3.86E-06	3.72E-06	9.39E-06
	Failure rate (%)	2.49	2.49	0.19	0.35	0.40	0.00
Mat.Mul.	Perf. (cycles)	111,025	117,563	113,929	117,563	118,057	111,025
	Energy cons. (J)	7.50E-06	5.79E-06	1.39E-05	9.70E-06	8.54E-06	2.25E-05
	Failure rate (%)	5.68	5.68	0.17	0.25	0.25	0.00
Minver	Perf. (cycles)	12,224	12,779	12,767	12,779	13,063	12,224
	Energy cons. (J)	8.34E-07	6.40E-07	1.56E-06	1.13E-06	1.10E-06	2.50E-06
	Failure rate (%)	2.77	2.77	0.34	0.49	0.63	0.00
POCSAG	Perf. (cycles)	18,926	21,247	21,027	21,247	23,302	18,926
	Energy cons. (J)	1.64E-06	1.19E-06	3.13E-06	1.94E-06	1.80E-06	4.91E-06
	Failure rate (%)	1.97	1.97	0.15	0.36	0.36	0.00
Qurt	Perf. (cycles)	17,972	18,691	19,016	18,691	19,554	17,972
	Energy cons. (J)	1.25E-06	9.72E-07	2.37E-06	1.69E-06	1.64E-06	3.76E-06
	Failure rate (%)	1.80	1.80	0.12	0.22	0.23	0.00
Sums	Perf. (cycles)	319	321	319	321	321	319
	Energy cons. (J)	2.00E-08	1.66E-08	3.68E-08	3.04E-08	3.04E-08	6.01E-08
	Failure rate (%)	2.96	2.96	0.27	0.30	0.30	0.00
x264	Perf. (cycles)	15,089	20,203	15,090	20,203	20,203	15,089
	Energy cons. (J)	1.08E-06	7.31E-07	2.00E-06	1.18E-06	1.18E-06	3.23E-06
	Failure rate (%)	2.94	2.94	0.33	0.45	0.55	0.00

without ILP control, 0.31%, and the ***Adaptive with ILP control*** 0.34%.

The ***TMR*** executes every instruction three times and votes the correct result, so it is able to mask all single faults that are injected (i.e., 0% of failure rate). Obviously, it is the best alternative when it comes to fault tolerance only. However, it has a huge overhead in energy and area. On the other hand, the ***Adaptive*** versions trade-off energy consumption, performance, and fault tolerance. Therefore, these versions have a slightly higher failure rate when compared to the techniques that focus only on fault tolerance, because they will not duplicate every single instruction of the program in order to allow energy optimization through power gating.

4.2.4 EDFP - Energy-Delay-Failure Product

To better analyze the trade-off among all axes (energy consumption, performance, and fault tolerance) for each design, we extended the equation for Energy-Delay Product (EDP) to Energy-Delay-Failure Product (EDFP). The EDFP is the product between energy consumption, performance, and the failure rate of a given application running on a certain processor configuration. Fig. 4d presents the relative EDFP when compared to the baseline configuration (i.e., the ***Unprotected*** processor). Note that the lower the EDFP, the better, as the goal is to reduce the energy consumption, delay (i.e., performance overhead) and failure rate. In addition, note that the EDFP for the ***TMR*** design will always be zero because such configuration masks all single faults in the processor. Therefore, when the processor is completely protected against transient faults (including protected checkers and voters), it is not possible to use EDFP to evaluate the trade-off as the failure rate would be zero. This boundary-value imprecision of the EDFP metric occurs also in other well-established fault-tolerance metrics such as the Mean Instructions to Failure (MITF) [52] and Mean Work to Failure (MWTF) [53], which are metrics that evaluate the performance-reliability trade-off. However, applying ***TMR*** comes with a huge overhead in area and energy consump-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TMSCS.2017.2760299, IEEE Transactions on Multi-Scale Computing Systems

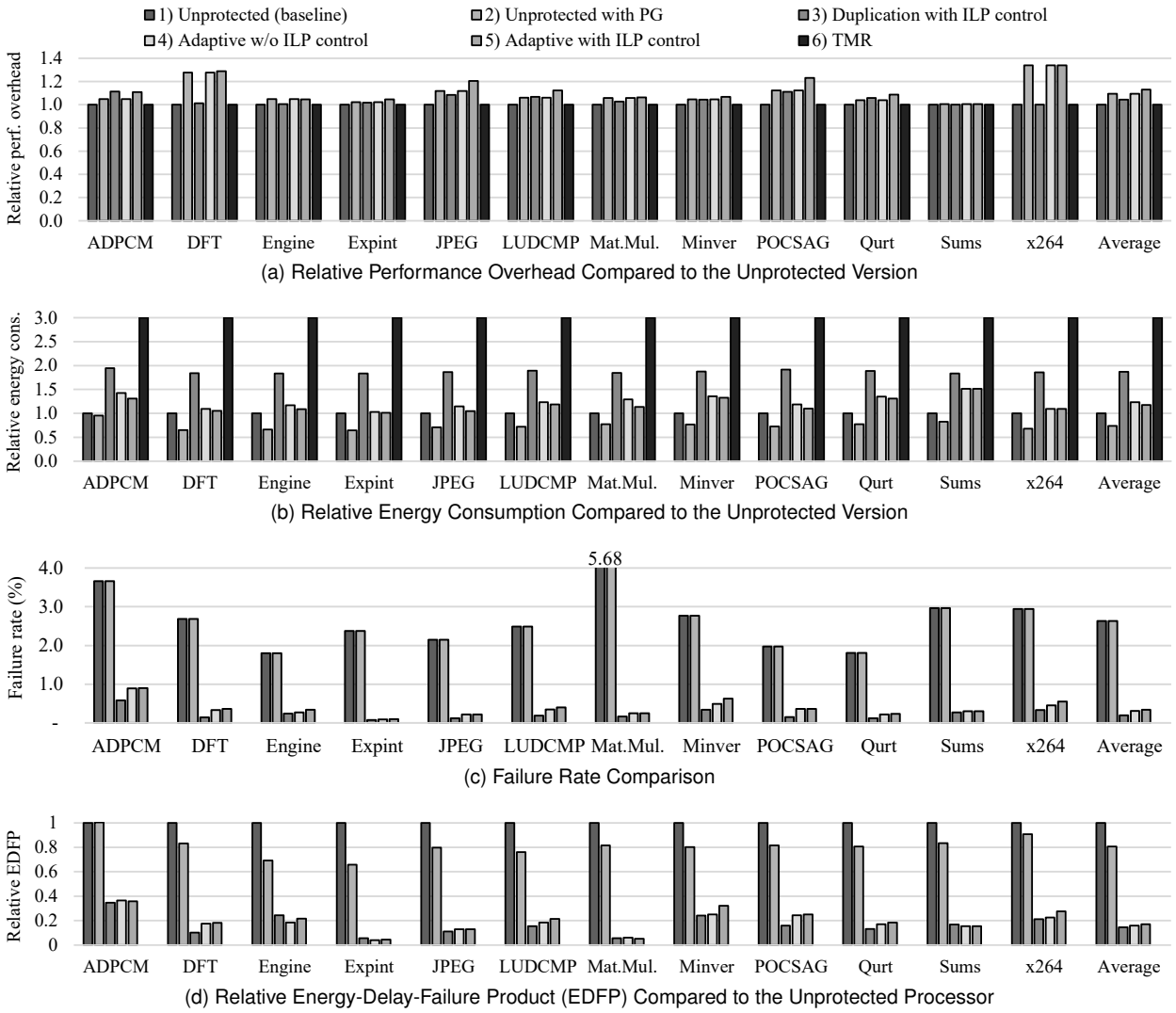

Fig. 4. Performance, Energy Overhead, Failure Rate, and EDFP Comparison

tion (200%). The other designs are compared in terms of EDFP next.

The average relative EDFP is of 0.81 for the ***Unprotected with PG***, which means that the energy optimization is proportionally higher than the performance overhead for such design. For the ***Duplication with ILP control*** version, even though the energy consumption is increased when compared to the baseline processor, the failure rate reduction compensates such overhead, resulting in an average EDFP of 0.14. The ***Adaptive*** versions (4)-(5) have an average EDFP of 0.16 and 0.17, respectively. Therefore, these designs efficiently trade-off the target metrics in order to balance the axes of fault tolerance, energy consumption, and performance.

4.2.5 The influence of NOPs

Finally, we also performed experiments to correlate the number of NOPs to the target axes in order to assess whether such metric can be used to aid the prediction of the application's behavior. The correlation coefficients were of only 0.14, -0.02, and -0.22 to the axes of fault tolerance, energy consumption, and performance, respectively, which shows that there is no correlation between these metrics in our case study. The low correlation indicates that other factors weigh more than the number of NOPs: for example, for fault tolerance, some instructions may be more critical (and sensitive) than others, which may lead to different failure rates, without depending solely on the NOP count. For energy consumption, the type of functional units that are used by the application also will influence in the correlation (which also depends on the type of instructions that are being executed). Finally, for performance, the issue-width occupation is dictated by the available parallelism in the application (i.e., instructions that do not have any data dependency can be executed in the same time-slot) and the available functional units in the processor. Thus, simply counting NOPs cannot be used to accurately predict the application's behavior in such scenario.

5 Conclusions and Future Work

In this work, an adaptive processor is proposed, and the axes of fault tolerance, energy consumption, and performance can be balanced, given a certain threshold defined *a priori* by the designer. Considering all aforementioned processor configurations, the ***Unprotected*** one is the best

alternative when considering only the axis of performance, so as the ***TMR*** and ***Duplication with ILP control*** are the best for the axis of fault tolerance. However, the ***Adaptive versions (4)-(5)*** are the only ones that can be tuned to prioritize and balance the axes of energy consumption, fault tolerance, and performance.

Finally, we will evaluate the applicability of this approach in other configurations and processors: in the current processor configuration (one branch, one memory unit, and four multipliers), we need to add one more branch and memory units to allow the execution of all duplicated instructions. In the simplest configuration of the current processor (one branch, one memory, and one multiplier), the overhead of adding one more of those units (for duplication) would be of only 2% in area. Therefore, it is very likely that this approach may be applied to any VLIW configuration/processor, with more or less area overhead depending on the available functional units.

This organization is similar to other commercial VLIW processors. For instance, the Intel Itanium [3] has eight ALUs (four integer and four MultiMedia eXtension (MMX)), two floating point multiply-add units, two memory units and three branch units; and the TMS320C6745 [5], from Texas Instruments, has six ALUs, two multipliers, and two memory units. Therefore, as long as the functional units are symmetric, the duplication can be applied to any type of functional unit, in any VLIW processor.

As future work, we will also consider other energy optimization techniques, such as DVFS. In addition, we will further assess the applicability of the proposed techniques to superscalar processors, considering that the execution behavior of a given application will very likely vary more executing on these processors than on the VLIW ones (mainly due to out-of-order and speculative execution). In a first analysis, the instruction duplication could be performed as described in [10] and the PG mechanism could be applied only to the functional units of such processor (instead of the whole *pipelane* as in a VLIW configuration). The dynamic profiling to detect application phases for PG would require modifications to cope with the dynamic behavior of the superscalar execution, for instance, the phase detection and prediction would probably require a more complex approach such as the one in [44].

REFERENCES

[1] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," *Dependable Systems and Networks (DSN), International Conference on*, pp. 389–398, 2002.

[2] S. Wong, T. Van As, and G. Brown, "ρ-VEX: A reconfigurable and extensible softcore VLIW processor," in *ICECE Technology, International Conference on*. IEEE, 2008, pp. 369–372.

[3] H. Sharangpani and K. Arora, "Itanium processor microarchitecture," *Micro, IEEE*, vol. 20, no. 5, pp. 24–43, 2000.

[4] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E.-J. D. Pol, M. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken, "TriMedia CPU64 architecture," in *Computer Design (ICCD), International Conference on*. IEEE, 1999, pp. 586–592.

[5] T. Instruments, "TMS320C6745/C6747 DSP technical reference manual," *SPRUH91A, Texas Instruments Inc*, 2011.

[6] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the IA-64 architecture," *IEEE Micro*, vol. 20, no. 5, pp. 12–23, 2000.

[7] B. Bornstein, T. Estlin, B. Clement, and P. Springer, "Using a multicore processor for rover autonomous science," in *Aerospace Conference*, 2011, pp. 1–9.

[8] R. Xiaoguang, X. Xinhai, W. Qian, C. Juan, W. Miao, and Y. Xuejun, "GS-DMR: Low-overhead soft error detection scheme for stencil-based computation," *Parallel Computing*, vol. 41, pp. 50–65, 2015.

[9] J.-M. Yang and S. W. Kwak, "A checkpoint scheme with task duplication considering transient and permanent faults," in *Industrial Engineering and Engineering Management (IEEM), IEEE International Conference on*. IEEE, 2010, pp. 606–610.

[10] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Microarchitecture, 34th annual ACM/IEEE international symposium on*. IEEE Computer Society, 2001, pp. 214–224.

[11] C. Bolchini, "A software methodology for detecting hardware faults in VLIW data paths," *Reliability, IEEE Transactions on*, vol. 52, no. 4, pp. 458–468, 2003.

[12] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-assisted soft error detection under performance and energy constraints in embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 4, p. 27, 2009.

[13] K. Mitropoulou, V. Porpodas, and M. Cintra, "DRIFT: Decoupled CompileR-Based Instruction-Level Fault-Tolerance," in *Languages and Compilers for Parallel Computing*. Springer, 2014, pp. 217–233.

[14] M. Schölzel, "Reduced Triple Modular redundancy for built-in self-repair in VLIW-processors," in *Signal Processing Algorithms, Architectures, Arrangements and Applications*. IEEE, 2007, pp. 21–26.

[15] Y.-Y. Chen and K.-L. Leu, "Reliable data path design of VLIW processor cores with comprehensive error-coverage assessment," *Microprocessors and Microsystems*, vol. 34, no. 1, pp. 49–61, 2010.

[16] I. Wali, A. Virazel, A. Bosio, L. Dilillo, and P. Girard, "An effective hybrid fault-tolerant architecture for pipelined cores," in *Test Symposium (ETS), 20th IEEE European*. IEEE, 2015, pp. 1–6.

[17] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, 1988.

[18] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture. 32nd Annual International Symposium on*. IEEE, 1999, pp. 196–207.

[19] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Fault-Tolerant Computing, Twenty-Ninth Annual International Symposium on*. IEEE, 1999, pp. 84–91.

[20] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Computer Architecture (ISCA), 27th Annual International Symposium on*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 25–36.

[21] J. C. Smolens, J. Kim, J. C. Hoe, and B. Falsafi, "Efficient resource sharing in concurrent error detecting superscalar microarchitectures," in *Microarchitecture (MICRO), 37th annual IEEE/ACM International Symposium on*. IEEE Computer Society, 2004, pp. 257–268.

[22] A. Jacobs, G. Cieslewski, A. D. George, A. Gordon-Ross, and H. Lam, "Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 4, p. 21, 2012.

[23] Y. Zhang and K. Chakrabarty, "Dynamic adaptation for fault tolerance and power management in embedded real-time systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 2, pp. 336–360, 2004.

[24] N. Nakka, K. Pattabiraman, and R. Iyer, "Processor-level selective replication," in *Dependable Systems and Networks, 37th Annual IEEE/IFIP International Conference on*. IEEE, 2007, pp. 544–553.

[25] J. Lee, Y. Ko, K. Lee, J. M. Youn, and Y. Paek, "Dynamic Code Duplication with Vulnerability Awareness for Soft Error Detection on VLIW Architectures," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 48:1–48:24, 2013.

[26] M. Brünink, A. Schmitt, T. Knauth, M. Süßkraut, U. Schiffel, S. Creutz, and C. Fetzer, "Aaron: An adaptable execution environment," in *Dependable Systems & Networks (DSN), IEEE/IFIP 41st International Conference on*. IEEE, 2011, pp. 411–421.

[27] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Code generation and optimization, international symposium on*. IEEE Computer Society, 2005, pp. 243–254.

[28] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles, "Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems," in *Hardware/software codesign and system synthesis, 5th IEEE/ACM international conference on*. ACM, 2007, pp. 233–238.

[29] N. Oh and E. J. McCluskey, "Error detection by selective procedure call duplication for low energy consumption," *Reliability, IEEE Transactions on*, vol. 51, no. 4, pp. 392–402, 2002.

[30] I. Assayad, A. Girault, and H. Kalla, "Tradeoff exploration between reliability, power consumption, and execution time," in *Computer Safety, Reliability, and Security*. Springer, 2011, pp. 437–451.

[31] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *Micro, IEEE*, vol. 23, no. 6, pp. 84–93, 2003.

[32] S. Aditya, S. A. Mahlke, and B. R. Rau, "Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 5, no. 4, pp. 752–773, 2000.

[33] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse, "The MAJC architecture: A synthesis of parallelism and scalability," *IEEE Micro*, no. 6, pp. 12–25, 2000.

[34] R. P. Colwell, J. O'donnell, D. B. Papworth, and P. K. Rodman, "Instruction storage method with a compressed format using a mask word," 1991.

[35] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," in *Microarchitecture. MICRO-29. IEEE/ACM International Symposium on*. IEEE, 1996, pp. 201–211.

[36] S. Jee and K. Palaniappan, "Performance evaluation for a compressed-VLIW processor," in *Applied computing, ACM symposium on*. ACM, 2002, pp. 913–917.

[37] J.-W. de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra, and others, "The TM3270 media-processor," in *Microarchitecture, 38th annual IEEE/ACM International Symposium on*. IEEE Computer Society, 2005, pp. 331–342.

[38] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[39] P. A. Raje and S. C. Siu, "Method and apparatus for sequencing and decoding variable length instructions with an instruction boundary marker within each instruction," 1999.

[40] A. Suga and K. Matsunami, "Introducing the FR500 embedded microprocessor," *Micro, IEEE*, vol. 20, no. 4, pp. 21–27, 2000.

[41] B. Hubener, G. Sievers, T. Jungeblut, M. Porrmann, and U. Ruckert, "CoreVA: A Configurable Resource-Efficient VLIW Processor Architecture," in *Embedded and Ubiquitous Computing (EUC), 12th IEEE International Conference on*. IEEE, 2014, pp. 9–16.

[42] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, B. Cherkauer, J. Stinson, J. Benoit, R. Varada, J. Leung, D. Limaye, and S. Vora, "A 65-nm dual-core multithreaded Xeon{®} processor with 16-MB L3 cache," *Solid-State Circuits, IEEE Journal of*, vol. 42, no. 1, pp. 17–25, 2007.

[43] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural techniques for power gating of execution units," in *Low power electronics and design, international symposium on*. ACM, 2004, pp. 32–37.

[44] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA '03. New York, NY, USA: ACM, 2003, pp. 336–349. [Online]. Available: http://doi.acm.org/10.1145/859618.859657

[45] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *ICCAD: International Conference on Computer-Aided Design*, 2011, pp. 694–701.

[46] C. McNairy and R. Bhatia, "Montecito: A dual-core, dual-thread Itanium processor," *IEEE micro*, no. 2, pp. 10–20, 2005.

[47] T. J. Slegel, R. M. Averill III, M. Check, B. C. Giamei, B. W. Krumm, C. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, and others, "IBM's S/390 G5 microprocessor design," *Micro, IEEE*, vol. 19, no. 2, pp. 12–23, 1999.

[48] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The M{ä}lardalen WCET Benchmarks: Past, Present And Future." *WCET*, vol. 15, pp. 136–146, 2010.

[49] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the Low-Power M CORE TM Architecture," in *Power driven microarchitecture workshop*, 1998, pp. 145–150.

[50] J. Hauser, "Berkeley SoftFloat," 2002. [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html

[51] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, "Statistical Fault Injection," in *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 122–127.

[52] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 264–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006723

[53] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, "Design and evaluation of hybrid fault-detection systems," in *32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 148–159.

Anderson L. Sartor received his B.Sc. in Computer Engineering from the Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 2013. Presently, he is a PhD student at UFRGS, in the Embedded Systems Laboratory. His primary research interests include embedded systems design, adaptive processors, reconfigurable systems, fault tolerance, and power dissipation management. For more information, please visit: http://inf.ufrgs.br/~alsartor.

Pedro H. E. Becker is a computer engineering student at the Universidade Federal do Rio Grande do Sul (UFRGS), Brazil. Since 2015, he is an undergraduate researcher at the Embedded Systems Laboratory of UFRGS. His primary research interests include Computer Architectures and Embedded Systems Design.

Joost Hoozemans received his B.Sc. in Computer Science from Utrecht University in 2011 and his M.Sc. in Computer Engineering from Delft University of Technology in 2014. His interests include Operating System support and configuration scheduling for dynamic VLIW processors and programmable FPGA overlays for medical image processing.

Stephan Wong is currently an associate professor at the Delft University of Technology, The Netherlands. He obtained his Ph.D. from the same university in December 2002. His PhD thesis entitled "Microcoded Reconfigurable Embedded Processor" describes the MOLEN polymorphic processor, organization, and (micro-)architecture. His research interests include: Reconfigurable Computing, Distributed Collaborative Computing, High-Performance Computing, Embedded Systems, and Hardware/Software Co-Design. He is also an IEEE Senior Member.

Antonio C. S. Beck received his B.S. in Computer Science from the Universidade Federal de Santa Maria (UFSM), in 2002, and the MSc. and Dr. degrees from the Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 2004 and 2008, respectively. Presently, he is an associate professor at the Applied Informatics Department at the Informatics Institute of UFRGS. His primary research interests include reconfigurable systems and embedded systems design, focusing on power consumption. He has published several technical papers on those topics, and he is the co-author of the book "Dynamic Reconfigurable Architectures and Transparent Optimization Techniques (2010-Springer)" and "Adaptable Embedded Systems (2012-Springer)". For more information, please visit: http://www.inf.ufrgs.br/~caco.