Adaptive and Polymorphic VLIW Processor to Optimize Fault Tolerance, Energy Consumption, and Performance

Anderson L. Sartor[1], Arthur F. Lorenzon[1], Sandip Kundu[2], Israel Koren[2], Antonio C. S. Beck[1]
[1]Institute of Informatics, Universidade Federal do Rio Grande do Sul (UFRGS), Brazil
{alsartor,aflorenzon,caco}@inf.ufrgs.br
[2]Dept. of Electrical and Computer Engineering, University of Massachusetts Amherst, United States
{kundu,koren}@umass.edu

ABSTRACT

Because most traditional homogeneous and heterogeneous processors have a fixed design that limits its runtime adaptability, they are not able to cope with the varying application behavior when one considers the axes of fault tolerance, performance, and energy consumption altogether. In this context, we propose a new dynamically adaptive processor design that is capable of delivering the best trade-off among these three axes according to the application at hand, or be tuned to optimize a specific metric. This is achieved by extending a polymorphic processor that can change its issue-width during runtime with specific mechanisms for fault tolerance, energy optimization, and performance enhancement. They are controlled by an optimization algorithm that evaluates and chooses which is the best configuration according to given requirements. Considering a metric that weighs all three axes, the proposed adaptive processor delivers a result that is 94.88% of the oracle processor on average, while a static configuration (defined at design time without runtime adaptation) only achieves 28.24% at most, which means that dynamic adaptation is required to cope with different application behaviors as there is not one specific configuration that fits all applications.

CCS CONCEPTS

• **Computer systems organization** → **Very long instruction word**; **Reliability**; *Redundancy*; • **Hardware** → *Power estimation and optimization*;

KEYWORDS

Adaptive processor, Energy consumption, Fault tolerance, VLIW

ACM Reference Format:
Anderson L. Sartor[1], Arthur F. Lorenzon[1], Sandip Kundu[2], Israel Koren[2], Antonio C. S. Beck[1]. 2018. Adaptive and Polymorphic VLIW Processor to Optimize Fault Tolerance, Energy Consumption, and Performance. In *CF '18: Computing Frontiers Conference, May 8–10, 2018, Ischia, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3203217.3203238

This work was conducted during a scholarship supported by CAPES: Anderson L. Sartor-PDSE-88881.135772/2016-01, Arthur. F. Lorenzon-PDSE-88881.135860/2016-01.

CF '18, May 8–10, 2018, Ischia, Italy

ACM ISBN 978-1-4503-5761-6/18/05…$15.00
https://doi.org/10.1145/3203217.3203238

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 [19].

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 instance, reducing energy consumption will likely reduce performance; increasing fault tolerance will increase energy consumption and possibly reduce performance; and improving performance will affect the energy consumption and possibly reduce fault tolerance.

Moreover, traditional core micro-architectures are not able to efficiently exploit the available resources to provide energy-efficient computation. Most commercial processors have large out-of-order cores (Intel Core i7) or small cores (ARM A15/A7). Large Out-of-Order (OoO) cores provide high performance for single threaded programs by exploiting Instruction-Level Parallelism (ILP), however, they are extremely power-inefficient for Thread-Level Paralellism (TLP) exploitation due to the complex OoO cores. On the other hand, small cores are able to exploit the TLP without wasting energy and area at the cost of reduced single thread performance. In order to cope with high ILP and TLP programs, heterogeneous chip multiprocessors have been proposed. These processors provide a few large cores for single thread performance and many small cores for multithreaded applications. However, the number of cores of each type must be chosen during design time, which restricts the ability of the core to adapt itself for different applications that do not fit the pre-determined number of cores, resulting in sub-optimal performance and energy consumption [2].

To this end, we propose a new processor design capable of dynamically and transparently adapt the hardware to the current application in order to deliver the best trade-off between fault tolerance, energy consumption, and performance. The proposed adaptive processor relies on a set of optimization techniques that work together to improve each of the aforementioned axes. This is achieved by extending a Very Long Instruction Word (VLIW) processor [16], providing it with a dynamic optimization algorithm that evaluates and chooses the best processor configuration for the application at hand during runtime, and including the ability of

switching between TLP and ILP modes (in a Morphcore fashion [9]), so the processor can also balance ILP and TLP with fault tolerance and energy consumption. In addition, we also enhanced the existent fault tolerance mechanism by providing both temporal and spatial instruction duplication, instead of only spatial, so we could enlarge even more our design space and opportunities for optimization.

As case study, the polymorphic version of the ρ-VEX VLIW processor [23], which is able to change the issue-width of the processor during runtime, was modified to support instruction duplication with rollback to improve fault tolerance, and power gating to reduce the energy consumption of the functional units. In addition, a decision module that controls all these techniques dynamically was implemented.

During the execution of each application, the optimization algorithm changes the configuration of the processor for each execution of a given kernel (i.e., a part of the program that is often executed a large number of times during its execution [21]), and it evaluates the outcome (a metric that considers the trade-off between fault tolerance, energy consumption, and performance). Based on this metric, the algorithm stops evaluating new configurations when the best one is found (the one that delivers the best trade-off). Therefore, a reduced number of steps can be performed by the learning algorithm. In addition, multiple applications can be executed in parallel, efficiently exploiting the available hardware.

Results show that the proposed processor is able to achieve, on average, 94.88% of the result delivered by an oracle processor when the trade-off between all axes is considered. The oracle processor executes all applications with the optimal processor configuration. On the other hand, the proposed adaptive processor evaluates and chooses the optimal configuration during runtime after a brief period of learning. A static processor configuration is only able to achieve, at most, 28.24% of the results from the oracle processor, demonstrating that a dynamic approach is required to efficiently match the hardware to the requirements of the application.

The remainder of this paper is organized as follows. Related work is discussed in Section 2. Next, in Section 3, the adaptive processor implementation is presented. In Section 4 the results are discussed in terms of the trade-off among fault tolerance, energy consumption, and performance. Finally, Section 5 presents conclusions and discusses future directions.

2 RELATED WORK

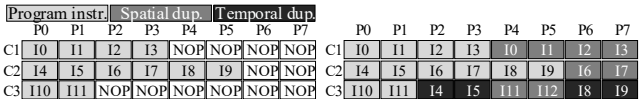
Next, some works that provide adaptive fault tolerance mechanisms are discussed, in which the level of instruction replication is adjusted during runtime, depending on the application behavior. Then, we will discuss other works that also adapt w.r.t. other metrics. NEDA [12] proposes to exploit temporal and spatial duplication in order to provide more flexibility in the instruction replication, as instructions that cannot be replicated in a given bundle are stored in a buffer so they can be executed in a following cycle. The authors evaluate the duplication (for fault detection only) and the triplication of instructions in order to provide fault tolerance. However, no reliability, area, power, and energy results are presented, only performance ones. In addition, the memory latency is not exploited to execute duplicated instructions (i.e., execute instructions while the processor is waiting for the memory). Aaron [3] tackles software and hardware errors by using diversified software components in the CPU spare cycles. Eight methods are used in this diversification. The system load is estimated, and the scheduler chooses the best variant to use the spare cycles (e.g., executing a reliability-oriented variant, which takes longer to execute, but is able to provide fault tolerance). Therefore, whenever load permits, more fault coverage is achieved; but it is only able to detect errors, not correct them.

A few works have proposed to target all three axes (i.e., fault tolerance, energy consumption, and performance): Tricriteria Scheduling Heuristics (TSH) [1] proposes an offline scheduling heuristic that produces a static multiprocessor schedule, based on a given application graph and a given multiprocessor architecture. 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 application 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 schedule length based on the aforementioned techniques. However, all processing and scheduling are done statically.

The authors in [20] propose a multi-objective strategy to choose the best core type considering power efficiency and reliability. Four Alpha processor core configurations are considered, varying voltage; frequency; size of the instruction queue, load-store queue and reorder buffer; and fetch and issue widths. Hardware counters are used to estimate the power dissipation and the Architectural Vulnerability Factor (AVF) of the processor, and with this data, a Cobb-Douglas production function is applied to choose the best core for a given part of the application. Even though reliability is considered, no solution to protect the cores (i.e., fault tolerance mechanism) is proposed.

In [13], a high-level reconfiguration approach is presented, which, based on user-defined constraints, changes the configuration of a heterogeneous multicore processor. Their approach is heavily based on the profiling of the applications before the execution, which must be done for all different processor frequencies in a heterogeneous processor. At every (pre-defined) number of cycles, a reconfiguration is triggered. The reconfiguration engine receives as input the defined reliability level, power budget, and performance counters; then, it defines what will be the frequency and voltage, and if the Error Correction Code (ECC) and the L2 cache should be enabled. The ECC is used to provide additional reliability, and the L2 cache can be disabled to save power. Therefore, this approach relies on static profiling of the application, limiting the dynamic adaptation.

The authors in [17] investigate the trade-off between the aforementioned axes in a Multiple Clustered Core Processor (MCCP). In order to modify the organization of the processor during runtime, some clusters are turned off, therefore, allowing to change between high-performance (2-issue dual core) and moderate performance (single issue dual core) processors (both superscalar processors are homogeneous). To improve fault tolerance, Redundant Multithreading (RMT) technique is used, in which the threads of the application are duplicated and compared. Their contribution is to choose which configuration will be applied to the next part of the application, which is done solely based on the past Instructions Per



(a) Unprotected Processor **(b) Temporal and Spatial Dup.**

Figure 1: Temporal and Spatial Duplication Exec. Example

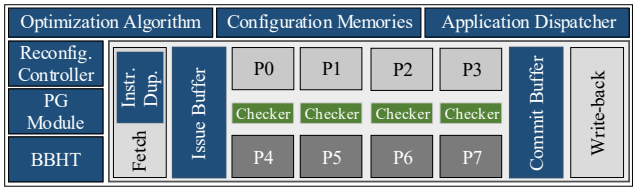


Figure 2: Adaptive Processor Overview

Cycle (IPC). Therefore, power and reliability do not influence the decision mechanism.

In [16], the authors propose a processor design that supports instruction duplication with rollback, power gating to reduce the energy consumption, and ILP control to increase the possibility of replicating instructions and applying power gating. Thresholds are defined to aid the decision to choose which techniques are going to be applied to the next phase of the application. The proposed adaptive processor extends the work from [16] by applying both temporal and spatial duplication, increasing the flexibility and coverage of the duplicated instructions. In addition, it exploits both TLP and ILP by using the polymorphic version of the ρ-VEX processor and a optimization algorithm is proposed to dynamically evaluate and change the processor configuration to adapt it to the application that is being executed, considering the trade-off between fault tolerance, energy consumption, and performance.

3 PROPOSED ADAPTIVE PROCESSOR

The adaptive processor is based on the polymorphic version of the ρ-VEX processor. The available dynamic issue-width adaptation was used and extended so the proposed techniques could also dynamically change at runtime. The chosen techniques to be part of this processor, in each of the axes, are detailed next.

3.1 Implemented Techniques

3.1.1 Fault Tolerance - Spatial and Temporal Duplication. Spatial duplication exploits idle resources within the same bundle (i.e., in the same cycle) to execute duplicated instructions (more details in [16]). However, spatial duplication sometimes is not able to duplicate all instructions in a bundle due to lack of empty slots. Thus, temporal duplication takes this technique one step further by allowing duplicated instructions to be executed in subsequent cycles, increasing the duplication flexibility. An example of temporal duplication is depicted in Figure 1. In Figure 1b, the instructions I_4, I_5, I_8, *and* I_9 are duplicated one cycle later than the original bundle. In case of an error, the faulty instruction is re-executed to correct the error. In order to maintain the consistency of the register file and the memory, after all duplicated instructions from a bundle are executed, or after verifying that those instructions will not be duplicated because the buffer is full, the results are committed.

Figure 2 depicts the additional modules for the adaptive processor, that includes two buffers: an issue buffer and a commit buffer (the remaining modules will be discussed later). The issue buffer is responsible for storing those instructions that could not be duplicated when only spatial duplication was applied. In this case, the pending (duplicated) instructions are kept in the buffer until there is a free slot so they can be scheduled. The commit buffer stores the bundles which still have pending instructions to be executed and checked. Once all instructions from a given bundle are verified, the bundle is committed.

The instruction scheduler first applies spatial duplication when each bundle is going to be executed. If there are instructions that could not be duplicated because there were not enough empty slots, these instructions will be stored in the buffers for temporal duplication. On the other hand, if there are still idle slots, the instruction scheduler will get pending instructions from the buffer and fill the bundle.

When the buffer is full, the temporal duplication is not applied to the next instructions until there is space in the buffer. Therefore, those instructions that cannot be stored in the buffer will be committed without verification. Temporal duplication also exploits the memory latency to execute duplicated instructions; whenever there is a cache miss, the instruction scheduler will start using those idle cycles to schedule the instructions that were waiting in the buffer. In the remaining cycles of the cache miss, the core is power gated in order to reduce the energy consumption while the processor is waiting for the memory. A similar approach that applies power gating while the processor is stalled on long memory access is presented by [8]. In order to increase the flexibility for the scheduling of the duplicated instructions, we have added extra functional units in each pipelane, which means that each pipelane is able to execute any instruction: memory, control, or logic. The extra cost of these functional units are taken into account, and the area overhead is further discussed in the results section.

3.1.2 Energy Optimization - Power Gating. A Power Gating (PG) mechanism, which is applied to the functional units of the processor, was implemented so both dynamic and static energy can be reduced. When all functional units from a given pipelane are turned off, the whole pipelane is also turned off to reduce energy consumption even more. A header transistor is used to turn off the supply voltage to the circuit block, which creates a virtual *Vdd*. Considering the technology parameters used in this work, the wake-up time of this transistor is three cycles [7].

To control when PG will be enabled, a small memory, called Basic Block History Table (BBHT) (direct mapped memory to store the PG configurations), is employed. Every time a program Basic Block (BB) ends its execution (reaches a branch instruction), the Program Counter (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), a PG configuration is built. If the entry is found, the processor will turn off the functional units according to the correspondent bits in the PG configuration. Therefore, dynamically profiling the application and detecting phases in which PG can be applied.

3.1.3 Performance - Multiple Applications. As the ρ-VEX processor has eight pipelanes, up to four applications can be executed concurrently (each one executing in a 2-issue core) considering the current implementation of the processor [23], which is similar to the Morphcore processor [9].

3.2 Optimization Algorithm

The runtime adaptation process for the aforementioned techniques comprises: modify the buffer size, enable or disable the power gating mechanism, and change the issue-width. Figure 2 also depicts the additional modules for the optimization algorithm, which comprises two main phases: learning and runtime (detailed in the next subsections). The buffers contain several banks, which can be shutdown to dynamically re-size the available buffer size. The bank size can be configured during design time, and each bank can be powered on/off independently by a bypass switch that enables or disables the power supply to a given bank [11, 22]. The PG mechanism is enabled and disabled through a control flag, and the hardware modules responsible for these functions are shut down when these mechanisms are disabled. Finally, the polymorphic ρ-VEX provides the issue-width adaptation through a switch network.

The *Configuration Memories* are two small memories that are used to store (1) the list of configurations that are going to be evaluated by the optimization algorithm and (2) the best configuration (for each benchmark, identified by its Process ID (PID)) after the learning phase is complete. The *Application Dispatcher* is responsible for scheduling the applications considering the available issue slots and the configurations that were dynamically chosen by the optimization algorithm, therefore, minimizing the idle hardware. Finally, the *Reconfiguration Controller* is responsible for changing the configuration of the processor during runtime. The whole process of reconfiguring the polymorphic processor takes 8 cycles, which is the time required to flush the pipeline, decode the new configuration and start the execution [6].

3.2.1 Learning Phase. A learning algorithm was implemented to evaluate different configurations and find which one delivers the best trade-off considering fault tolerance, energy consumption, and performance. This learning is done by changing the hardware configuration (polymorphic behavior) at runtime so each execution the application's kernel can be evaluated with a different configuration. For this, the kernel of the application is identified by using *pragmas* (that can be inserted by a simple script), so the compiler is able to generate a code that will inform the hardware which are the regions that need to be evaluated. Therefore, the basic idea of the learning algorithm is to evaluate a different hardware configuration each time a given kernel is executed until the best one is found. This means that the application may run with a sub-optimal configuration until the algorithm converges to the ideal. However, as we will demonstrate in the results section, this learning phase can be performed with minimal overhead.

In order to evaluate the trade-offs among the aforementioned axes and choose the best configuration, the Mean Work Per Unit of Energy to Failure (MWPUETF) metric is proposed. This metric is composed of the Mean Work to Failure (MWTF), which was adapted from [14], and the energy consumption of the application. The MWTF equation is presented in (1), where the *core utilization* is the ratio between the number of program instructions and the total number of instructions (program instructions plus No-Operations (NOPs)). With that, it is possible to capture the trade-off between performance and fault tolerance. This allows to evaluate the reliability of different issue-widths after removing the influence of the NOPs and the difference in execution time when the issue-width is changed. To obtain the failure rate, a fault injection campaign was conducted and faults were injected at the design's gate-level, using the Simbah-FI framework [15]. The metric MWPUETF is depicted in (2), which is the ratio between the MWTF and the energy consumption. Each of these metrics is obtained at runtime by using hardware counters, except the failure rate, which is obtained from a previous fault injection campaign.

$$MWTF = \frac{amount\ of\ work}{number\ of\ errors} = \frac{core\ utilization}{(failure\ rate) \times (exec.\ time)} \quad (1)$$

$$MWPUETF = \frac{MWTF}{energy\ consumption} \quad (2)$$

Algorithm 1 presents the learning phase. It has a list of configurations to be evaluated that can be selected during design time (and stored in one of the configuration memories), and it receives the *kernelList* as a parameter, which is the identification of the target applications' kernels.

The following optimization decisions are made regarding the learning flow:

- The issue-width is evaluated in the following order: 2, 4, then 8-issue. In addition, the number of evaluations is reduced if the MWPUETF decreases when increasing the issue-width. As an example, if the 4-issue core results in a worse MWPUETF when compared to the 2-issue, the 8-issue core will not be evaluated, because it will result in a MWPUETF that is lower than the 2-issue. This is explained by the following reasoning, the 8-issue spends more energy (which negatively affects MWPUETF), but the improvement in performance in most cases is much lower than the theoretical 2x (when going from 4-issue to 8-issue) because the compiler is not able to fill all bundles with independent instructions. Therefore, the energy consumption weighs more than the performance in this scenario.
- For the temporal duplication, there are a few parameters that can be modified to trade-off performance, reliability, and energy consumption, such as the buffer size, and the use of PG. During the learning, the buffer size stops being evaluated when the MWPUETF gets worse than for the previous buffer size. When this happens, it means that increasing the energy consumption (larger buffer) does not outweigh the increase in the number of duplicated instructions (if any, because the previous buffer size could be enough to duplicate all instructions that were stored in the buffer).

Therefore, by applying these two optimization strategies, it is possible to reduce the number of steps performed by the learning algorithm, since those additional steps would not improve the result.

After the first kernel of the queue is obtained in Algorithm 1, the optimization module verifies if there are enough idle slots to schedule this kernel with a given configuration (*l. 4-5*), if there is, the defined configurations are evaluated, one for each execution

Algorithm 1: Optimization Algorithm - Learning Phase

```
1  optQueue ← kernelList
2  while kernel in optQueue do
3      currentKernel ← get first application of optQueue
4      if currentKernel was not tested in this time-slot and there are idle slots
         then
5          if currentKernel fits in the current unused issue-slots then
6              for testConfig in configList do
7                  if testConfig = "TemporalDuplication" then
8                      for bufferSize in bufferSizeList do
9                          for TDconfig in TDconfigList do
10                             Execute currentKernel with
                                 testConfig, TDconfig, bufferSize
11                             if currentResult > previousResults
                                 then
12                                 Save currentResult
13                         if resultCurrentBuffer <
                             resultPreviousBuffer then
14                             Break to the next testConfig
15                 else
16                     Execute currentKernel with testConfig
17                     if currentResult > previousResults then
18                         Save currentResult
19             if currentResult is better than the previous issue-width then
20                 if current issue-width < max(issue-width) then
21                     currentKernel.nextIssue ← next issue-width
22                     Put currentKernel back in the optQueue
23                 else
24                     Save currentResult as the best one
25             else
26                 Mark the previous result as the best one
27         else
28             Put currentKernel back in the queue
29     else
30         Put currentKernel back in the queue
31         if there are idle slots then
32             Apply PG on idle slots
33         Wait for an application to end and update time-slot
34 return the best configuration for each application
```

of the given kernel. While testing the temporal duplication, if the result for the current buffer (*resultCurrentBuffer*) is better than the result for the previous buffer, it is saved in the *configuration memory* (*l. 12*), otherwise, it stops testing further buffer sizes (*l. 14*). Then, the result for the current issue-width of this kernel is compared to the one from the previous issue-width (*l. 19*), if it is better, the next issue-width is evaluated (until it tests the maximum issue-width), otherwise, the best result is the *currentResult*. This is repeated for all kernels that fit in the available issue-slots (*l. 4*). For example, in a given moment, four kernels running on 2-issue mode can be executed in parallel. When all issue-slots are occupied, the optimization module waits for a kernel to end, so it can evaluate the next kernel to be scheduled (*l. 30-34*). In case there are still idle slots after all kernels tried to be scheduled (because they did not fit in the available slots), PG is applied to those slots, to reduce the energy consumption (*l. 31-32*).

3.2.2 Runtime Phase. In the runtime phase, the best configuration was already determined by the learning mechanism, and it

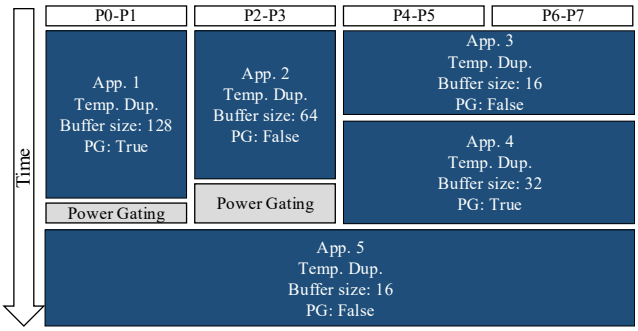


Figure 3: Application Scheduling Example

was stored in the *configuration memory*. Therefore, the *application dispatcher* is responsible for continuing the execution of the applications while trying to schedule as many applications as possible in parallel. The execution time of each kernel was already saved by the learning phase, so in the runtime phase the applications are sorted by the descending order of the execution time (in order to allow more flexibility in the scheduling for the other benchmarks) and the applications are scheduled to fill the available slots.

An execution example is depicted in Figure 3, in which the *App. 1* and *App. 2* continue the execution after the learning phase in the best configuration for each of these applications, which is the 2-issue processor with temporal duplication and buffer size of 128 with PG for *App. 1* and a buffer of 64 without PG for *App. 2*. At the same time, the other four pipelanes are occupied with *App. 3* having a buffer size of 16 without PG. After *App. 3* finishes the execution, *App. 4* is scheduled, maintaining the 4-issue configuration with another buffer size and PG configuration. Finally, *App. 5* is scheduled in the 8-issue mode, occupying all pipelanes. Note that power gating is applied to the empty slots that cannot be used to fit another application (between *Apps. 1 and 5*, and *Apps. 2 and 5*).

4 RESULTS

4.1 Methodology

The Cadence Encounter RTL compiler was used to obtain power dissipation and Application-Specific Integrated Circuit (ASIC) area, using a 65nm Complementary Metal-Oxide-Semiconductor (CMOS) cell library from STMicroelectronics (the operating frequency was set to 200MHz). CACTI-P [10] was used to estimate the area and energy consumption of the following modules:

- BBHT: 256 lines of 64 bits each, one write and one read port.
- Buffer sizes for the temporal duplication: buffers of 16, 32, 64, and 128 entries.
- *Configuration memories*: Comprises two small memories, the first with 32 read-only entries for storing the list of configurations that are going to be evaluated by the optimization algorithm (27 configurations are used in the current evaluation); the second with 16 entries to store the best result and the corresponding configuration for each benchmark during the learning phase (11 entries are used as we are evaluating 11 benchmarks).

- Cache: the instruction and data caches have 16KB for the 8-issue configuration (the 4-issue and 2-issue will access 8KB and 4KB, respectively, therefore, maintaining the 16KB for the whole polymorphic processor). The caches are 4-way associative with a block size of 4 bundles (128B for the 8-issue) and a *next-line* prefetch mechanism, which will also fetch the following line whenever a cache miss occurs.
- Main memory: a 512MB memory is used, having a miss latency of 30 cycles.

All the previous memories and buffers have ECC enabled. For the area and energy consumption evaluation, all additional modules that were implemented are taken into account, which include those that were synthesized and the memories simulated with CACTI.

In addition, the polymorphic core requires that the number of registers to be quadrupled in order to have four contexts (4x2-issue) of 64 registers each, resulting in a total of 256 registers. The same reasoning applies to the buffer sizes: each configuration supports up to 128 buffers, so a total of 512 entries is required. For the energy consumption estimation, when there is no switching activity, only static power is considered. Otherwise, the average switching activity of the circuit is considered to be 30% [4] for the dynamic power dissipation.

The benchmark set is composed of 16 applications from the WCET [5] and Powerstone [18] benchmark suites. In-house simulators were used to simulate the polymorphic processor and the proposed techniques, while hardware modules were implemented and synthesized to estimate power and energy.

4.2 Design Space Exploration

In this section, static configurations using the aforementioned techniques are evaluated. We show that each benchmark has a specific configuration in terms of the parameters for the temporal duplication, PG, and issue-width that results in the best trade-off between fault tolerance, performance, and energy consumption. Therefore, motivating the use of a dynamic adaptation technique to cope with these different application characteristics. In Section 4.3, the dynamic behavior of the core will be further explored.

Figure 4 (note that the *Y* axis is in logarithmic scale) shows the MWPUETF for each issue-width (2, 4, and 8). The first observation is that it is not possible to find a single configuration (in terms of issue-width) that will be always better than the others considering all axes and benchmarks. *TD* means Temporal Duplication, and *PG* Power Gating. We use the 8-issue as the baseline, in order to evaluate the relationship between different issue-widths. For each configuration of the temporal duplication, there are two extra options: enable or disable the power gating, and the buffer size. For instance, for the *CJPEG*, the 8-issue delivers the best MWPUETF, the same goes to the *CRC* with the 4-issue and the *DFT* with the 2-issue. By applying temporal duplication, the MWPUETF can be improved by up to 2K times (*Sums* with a buffer of 128) when compared to the unprotected processor.

By increasing the buffer size, more instructions can be stored for duplication, however, more power will be dissipated by these buffers. Therefore, each benchmark will have an ideal buffer size considering its behavior, increasing the buffer after the point where the benchmark effectively uses it will only result in increased power dissipation, while reducing the size of the buffer may affect the reliability as fewer instructions will be stored for the temporal duplication. Similar reasoning can be applied to the PG mechanism, which will turn off more hardware, at the cost of reducing the number of duplicated instructions in some cases.

Table 1: Best Configuration for Each Benchmark

Buffer	PG	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
16	F						8-i					4-i
	T								4-i			
32	F											
	T											
64	F		4-i					8-i				
	T	8-i										
128	F					4-i						
	T			2-i	2-i					2-i	2-i	

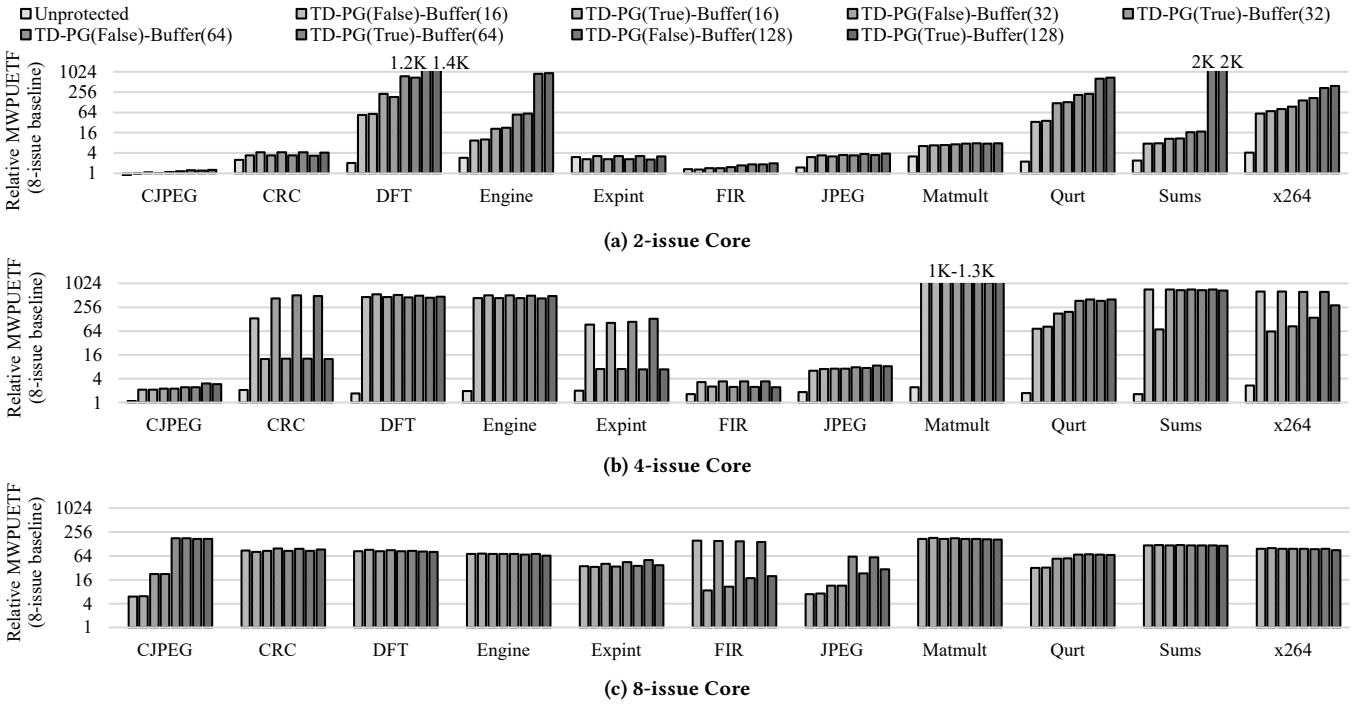
Due to space constraints, each of the axes will be only briefly discussed next. The duplication ratio varies from 9% to 100% depending on the application and processor configuration. The 2-issue requires a large buffer for most benchmarks in order to duplicate all, or almost all instructions. While in the 4-issue, high duplication ratio is achieved with intermediate buffer sizes and in the case of the 8-issue, small buffers are able to provide high duplication levels. Note that even for applications that are able to duplicate 100% of the instructions, the checker remains vulnerable to faults and such vulnerability is considered in the reliability evaluation. When the temporal duplication is used without applying PG and with a buffer size of 128, the average performance overhead is 27.80%, 7.39%, and 1.25% for the 2, 4, and 8-issue, respectively. Finally, when the PG is turned on, the overhead goes to 31.29%, 15.47%, and 12.78%. On average, the temporal duplication increases the energy consumption by 65.04%, 30.17%, and 43.30% for the 2, 4, and 8-issue. The 4-issue presents the lowest energy overhead because the 2-issue has high performance overhead, while the 8-issue has elevated power dissipation, both influencing the energy consumption.

Table 1 depicts the best configuration for each benchmark considering the MWPUETF metric (the *2-i*, *4-i*, and *8-i* represent the 2, 4, and 8-issue processors). Each application has an ideal configuration and issue-width that needs to be applied in order to provide the best trade-off among all axes. Therefore, an adaptive processor that dynamically adjusts itself to the application is required to execute each benchmark in the best possible configuration (discussed next).

4.3 Optimization Algorithm Evaluation

In this subsection, first, the number of steps of the learning algorithm required by the adaptive processor to find the best configuration for each application will be evaluated. Then, its ability to cope with different applications' behavior will be assessed through the MWPUETF metric.

4.3.1 Number of Steps to Find the Best Configuration. Considering that we are evaluating a total of 27 possible configurations (unprotected: 3 issue-widths, and temporal duplication: 4 buffer sizes, 3 issue-widths, and enable or disable the PG), the maximum number of steps that the optimization algorithm may perform is



(a) 2-issue Core

(b) 4-issue Core

(c) 8-issue Core

Figure 4: MWPUETF Comparison (8-issue baseline)

Table 2: Number of Kernel Executions and Number of Steps to Find the Best Configuration

	CJPEG	CRC	DFT	Engine	Expint	FIR	JPEG	Matmult	Qurt	Sums	x264
Num. of steps	19	19	23	19	21	17	19	19	19	19	23
Num. of iterations	215500	400	44440	200	40000	600	200	250	2000	138500	1000
Tests-Iterations ratio	0.01%	4.75%	0.05%	9.50%	0.05%	2.83%	9.50%	7.60%	0.95%	0.01%	2.30%

27. Table 2 presents the number of executions for each kernel, and the number of steps required to find the best configuration for each application. For these benchmarks, the number of steps of the learning algorithm varies from 17 to 23. Thus, the optimization algorithm is able to find the best configuration without evaluating all possible scenarios. The average number of steps is 19.65.

The same table also shows the ratio between the number of steps and iterations, which vary from 0.01% to 9.5%. The latter occurs in the *Engine* application: 19 out of 200 kernel executions are performed by the learning phase, then, the other 181 iterations are executed in the best configuration. As each execution is already an iteration of the kernel and the learning phase contributes to the final result of the application, a given iteration is not executed twice after finding the best configuration. In addition, all overheads for the configuration switching are taken into account, which is lower than 0.003% in terms of performance for all benchmarks.

4.3.2 MWPUETF Comparison - Adaptive Processor. In order to evaluate the proposed adaptive processor, we compared it to the best static configuration *on average* from the configurations evaluated in Section 4.2, for each issue-width. In addition, each static configuration is able to execute multiple applications concurrently. An Oracle processor is also used for comparison and each of these configurations is detailed next.

- ***Temporal Duplication 2-2-2-2*** : On average, the best 2-issue configuration is the 128 buffer with PG. Also, four applications can be executed in parallel in this configuration.
- ***Temporal Duplication 4-4*** : The best configuration is the 128 buffer without PG and it runs two applications.
- ***Temporal Duplication 8*** : It has a 64 buffer without PG as the best configuration, running one application.
- ***Adaptive Processor*** : The adaptive processor is able to switch between different issue-widths and configurations considering fault tolerance, energy optimization, and performance. Each benchmark is dynamically evaluated and the best configuration is found according to the aforementioned optimization algorithm (Algorithm 1).
- ***Oracle Processor*** : The oracle processor executes all applications in the best possible configuration for each one, providing an upper bound for comparison.

Figure 5 presents the MWPUETF results normalized to the ***Oracle Processor***. The ***Adaptive Processor*** is able to get from 86.01% to 99.99% of the oracle's result, having an average of 94.88%. This means that the ***Adaptive Processor*** is able to deliver almost the

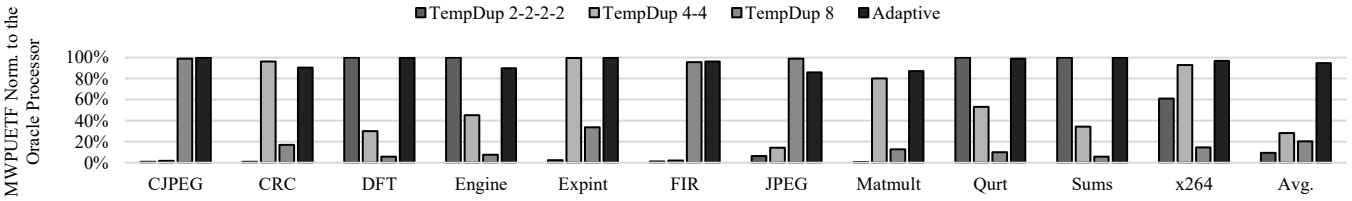


Figure 5: MWPUETF Normalized to the Oracle Processor

same result as the ***Oracle Processor***, but in a completely transparent, automatic, and dynamic manner. The small decrease in the final MWPUETF is due to the learning phase, which executes the application in a sub-optimal configuration. On the other hand, the best static configuration (***Temporal Duplication 4-4***) is only able to achieve 28.24% of the result from the ***Oracle Processor***, followed by the ***Temporal Duplication 8*** with 20.24%, and ***Temporal Duplication 2-2-2-2*** with 9.21%.

Finally, all modules for the adaptive processor, including the fault tolerance and PG mechanisms, result in an area overhead of 26.82% when compared to the unprotected processor. In addition, if one decides to restrict the steps of the learning algorithm to configurations that are most likely to result in higher improvements, for example, by eliminating the evaluation of the unprotected configurations, the ***Adaptive Processor*** can get even closer to the ***Oracle Processor***, as fewer steps will be performed to find the best configuration.

Therefore, the proposed ***Adaptive Processor*** is able to dynamically choose the most appropriate processor configuration for each application, and it provides results close to the ***Oracle Processor*** when the trade-off among fault tolerance, energy consumption, and performance is considered.

5 CONCLUSIONS AND FUTURE WORK

In this work, an adaptive processor was developed to trade-off the axes of fault tolerance, energy consumption, and performance, by implementing specific mechanisms for each of these axes. The multi-objective optimization algorithm dynamically evaluates the applications that are being executed and chooses the best configuration of the processor to maximize the MWPUETF, on average achieving 94.88% of the results of an oracle. This demonstrates that the optimization algorithm is able to quickly and accurately select the best configuration for each benchmark. In addition, the application dispatcher maximizes the number of applications executing in parallel to exploit the available pipelanes.

As future work, additional techniques for these axes may be integrated into the proposed processor, e.g., apply PG to the register file as well. In addition, the sensitivity of the instructions will be assessed, so critical instructions can be prioritized in the temporal duplication (in this work, a first-in-first-out approach was applied).

REFERENCES

[1] Ismail Assayad, Alain Girault, and Hamoudi Kalla. 2011. Tradeoff exploration between reliability, power consumption, and execution time. In *Computer Safety, Reliability, and Security*. Springer, 437–451.

[2] Antonio Carlos Schneider Beck, Carlos Arthur Lang Lisbôa, and Luigi Carro. 2012. *Adaptable embedded systems*. Springer Science & Business Media.

[3] Marc Brünink, André Schmitt, Thomas Knauth, Martin Süßkraut, Ute Schiffel, Stephan Creutz, and Christof Fetzer. 2011. Aaron: An adaptable execution environment. In *Dependable Systems & Networks (DSN), IEEE/IFIP 41st International Conference on*. IEEE, 411–421.

[4] Bibiche Geuskens and Kenneth Rose. 2012. *Modeling microprocessor performance*. Springer Science & Business Media.

[5] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. *WCET* 15 (2010), 136–146.

[6] J Hoozemans, J Johansen, J V Straten, A Brandon, and S Wong. 2015. Multiple contexts in a multi-ported VLIW register file implementation. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6.

[7] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. 2004. Microarchitectural techniques for power gating of execution units. In *Low power electronics and design, int. symp. on*. ACM, 32–37.

[8] Kwangok Jeong, Andrew B Kahng, Seokhyeong Kang, Tajana S Rosing, and Richard Strong. 2012. MAPG: Memory Access Power Gating. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*. 1054–1059.

[9] K Khubaib, M Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N Patt. 2012. Morphcore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *Microarchitecture (MICRO), 45th Annual IEEE/ACM International Symposium on*. IEEE, 305–316.

[10] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Int. Conf. on Computer-Aided Design*. 694–701.

[11] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. 2001. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Washington, DC, USA, 90–101.

[12] R Psiakis, A Kritikakou, and O Sentieys. 2017. NEDA: NOP Exploitation with Dependency Awareness for Reliable VLIW Processors. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 391–396.

[13] T Ramírez, E Herrero, N Axelos, J Carretero, N Foutris, D Sanchez, and X Vera. 2012. Mitigating lower layer failures with adaptive system reconfiguration. In *Proceedings of the 19th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES*. 109–114.

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

[15] A L Sartor, P H E Becker, and A C S Beck. 2017. Simbah-FI: Simulation-Based Hybrid Fault Injector. In *VII Brazilian Symposium on Computing Systems Engineering (SBESC)*. 94–101.

[16] A L Sartor, P H E Becker, J Hoozemans, S Wong, and A C S Beck Filho. 2017. Dynamic Trade-off among Fault Tolerance, Energy Consumption, and Performance on a Multiple-issue VLIW Processor. *IEEE Transactions on Multi-Scale Computing Systems* In Press, 99 (2017).

[17] Toshinori Sato and Toshimasa Funaki. 2008. Dependability, power, and performance trade-off on a multicore processor. In *Asia and South Pacific Design Automation Conference (ASPDAC)*. 714–719.

[18] Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. 1998. Designing the Low-Power MCORE Architecture. In *Power driven microarchitecture workshop*. 145–150.

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

[20] Sudarshan Srinivasan, Israel Koren, and Sandip Kundu. 2015. Online mechanism for reliability and power-efficiency management of a dynamically reconfigurable core. In *Computer Design (ICCD), 33rd International Conference on*. IEEE, 327–334.

[21] A Tiwari, M A Laurenzano, L Carrington, and A Snavely. 2012. Modeling Power and Energy Usage of HPC Kernels. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 990–998.

[22] H Wang, I Koren, and C M Krishna. 2011. Utilization-Based Resource Partitioning for Power-Performance Efficiency in SMT Processors. *IEEE Transactions on Parallel and Distributed Systems* 22, 7 (2011), 1150–1163.

[23] Stephan Wong, Thijs Van As, and Geoffrey Brown. 2008. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In *ICECE Technology, Int. Conf. on*. IEEE, 369–372.