Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 452589, 17 pages
doi:10.1155/2011/452589

Research Article

Dynamic Reconfigurable Computing: The Alternative to Homogeneous Multicores under Massive Defect Rates

Monica Magalhães Pereira and Luigi Carro

Instituto de Informática, Universidade Federal do Rio Grande do Sul, 91501-970 Porto Alegre, RS, Brazil

Correspondence should be addressed to Monica Magalhães Pereira, mmpereira@inf.ufrgs.br

Received 25 August 2010; Accepted 14 December 2010

Academic Editor: Michael Hübner

The aggressive scaling of CMOS technology has increased the density and allowed the integration of multiple processors into a single chip. Although solutions based on MPSoC architectures can increase application's speed through TLP exploitation, this speedup is still limited to the amount of parallelism available in the application, as demonstrated by Amdahl's Law. Moreover, with the continuous shrinking of device features, very aggressive defect rates are expected for new technologies. Under high defect rates a large amount of processors of the MPSoC will be susceptible to defects and consequently will fail, not only reducing yield but also severely affecting the expected performance. This paper presents a run-time adaptive architecture that allows software execution even under aggressive defect rates. The proposed architecture can accelerate not only highly parallel applications but also sequential ones, and it is a heterogeneous solution to overcome the performance penalty that is imposed to homogeneous MPSoCs under massive defect rates.

1. Introduction

The scaling of CMOS technology has increased the density and consequently made the integration of several processors in one chip possible. Many architectural solutions with several cores can be found in the literature in the past decade [1]. These solutions are mainly used to accelerate execution through task level parallelism (TLP) exploitation. However, the speedup achieved by these systems is limited to the amount of parallelism available in the applications, as already foreseen by Amdahl [2]. According to Rutzig et al. [3], current embedded system domain applications present a heterogeneous behavior that includes not only highly parallel applications but also general purpose applications that are also migrating to embedded system domain, such as browsers and high definition video processing.

To cope with this heterogeneous behavior an architecture design is necessary to not only accelerate execution through TLP but also find other ways to accelerate software execution, for example, through ILP (instruction level parallelism) exploitation, or accelerate sequential execution. One solution to overcome Amdahl's law and sustain the speedup of MPSoCs is the use of heterogeneous cores, where each core is specialized in different application sets. An example of a heterogeneous architecture is the Samsung S5PC100 [4] used in Apple's iPhone technology [5]. The Samsung S5PC100 is a multimedia-based MPSoC that has an ARM-based central general purpose processor and five multimedia accelerators targeted to DSP processing. Each accelerator is targeted to a different technology such as JPEG codification and NTSC/PAL/HDMI technologies, [4].

Although the use of heterogeneous cores can be an efficient solution to improve the MPSoC's performance, there are other constraints that must be considered in the design of multicore systems, such as reliability. The scaling process shrinks wires' diameter, making them more fragile and susceptible to break. Moreover, it is also harder to keep contact integrity between wires and devices [6]. According to Borkar [7], in a 100 billion transistor device, 20 billion will fail in the manufacture and 10 billion will fail in the first year of operation.

At these high defect rates, it is highly probable that defects will affect most of the processors of the MPSoC (or even all the processors), causing yield reduction and aggressively

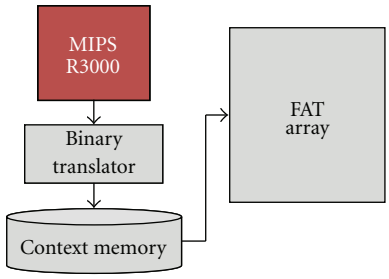


Figure 1: FAT-array system block diagram.

affecting the expected performance. Furthermore, in cases when all the processors are affected, this makes the MPSoC useless. An analysis shown in the next sections demonstrates that under a 20% defect rate a 64-core homogeneous MPSoC would have lost an average of 63 cores, presenting a completely sequential execution. On the other hand, considering a 0.1% defect rate of current technologies, on average only two cores are lost. More details about this analysis are presented in next sections.

To cope with this reliability and consequent loss of performance problem, one solution is to include some fault tolerance approach. Although there exist many solutions proposed to cope with manufacturing defects [8], most of these solutions do not cope with the high defect rates predicted to future technologies. Moreover, the proposed solutions present some additional cost that causes a high impact on area, power, or performance or even in all three [9]. Solutions such as dual and triple modular redundancy (DMR, TMR) presented by HP NonStop architectures [10] and Aggarwal's configurable isolation approach [11] have obvious area and power issues.

In this context, this paper presents a reconfigurable architecture as an alternative to homogeneous multicores that allows software execution even under high defect rates and accelerates execution of parallel and sequential applications. The architecture uses an on-line mechanism to configure itself according to the application, and its design provides acceleration in parallel as well as in sequential portions of the applications. In this way, the proposed architecture can be used to replace homogeneous MPSoCs, since it sustains performance even under high defect rates, and it is a heterogeneous approach to accelerate all kinds of applications. Therefore, its performance is not limited to the parallelism available in the applications.

To validate the architecture, we compare the performance and area of the system to a homogeneous MPSoC with equivalent area. The results indicate that the proposed architecture can sustain execution even under a 20% defect rate, while in the MPSoC all the processors become useless even under a 15% defect rate. Furthermore, with lower defect rates, the proposed architecture presents higher acceleration when compared to the MPSoC under the same defect rates, when the TLP available in the applications is lower than 100%.

The rest of this paper is organized as follows. Section 2 presents the adaptive system. Section 3 details the defect tolerance approach and some experimental results. Section 4 presents a comparison of area and performance between the reconfigurable system and the equivalent homogeneous MPSoC considering different defect rates. Section 5 presents some related work. Finally, Section 6 presents the conclusions and future works.

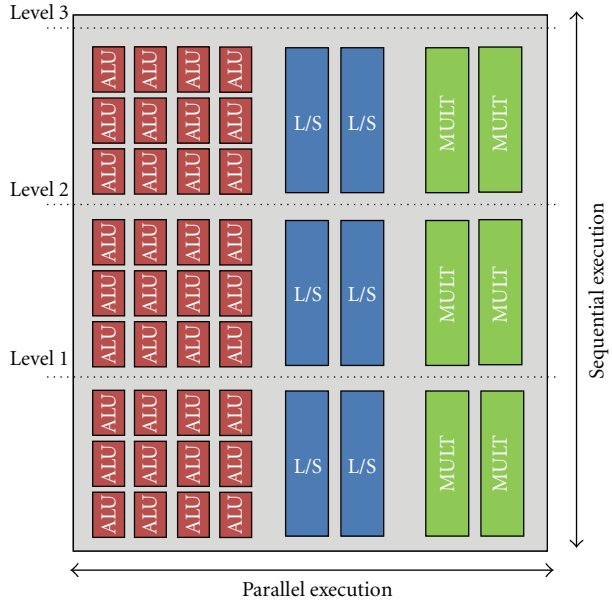


Figure 2: FAT-array block diagram.

2. Proposed Architecture

The FAT-array (fault tolerant array) system consists of a coarse-grained reconfigurable array tightly coupled to an MIPS R3000 processor; a mechanism to generate the configuration, called Binary Translator; and the context memory that stores the configuration [12, 13]. Figure 1 presents the block diagram of the FAT-array system.

The FAT-array consists of a combinational circuit that comprises three groups of functional units: the arithmetic and logic group, the load/store group, and the multiplier group. Figure 2 presents the FAT-array block diagram.

Each group of functional units can have a different execution time, depending on the technology and implementation strategy. Based on this, in this work the ALU group can perform up to three operations in one equivalent processor cycle and the other groups execute in one equivalent processor cycle. The equivalent processor cycle is called level. Figure 2 also demonstrates the parallel and sequential paths of the FAT-array.

The interconnection model, illustrated in Figure 3, is based on multiplexers and buses. The buses are called context lines and receive data from the context registers, which store the data from the processor's register file.

According to Figure 3, the reconfigurable architecture design presents a bottom-up data path. All data that are generated by the functional units flow from the bottom row to the top row through the interconnection model. Moreover,

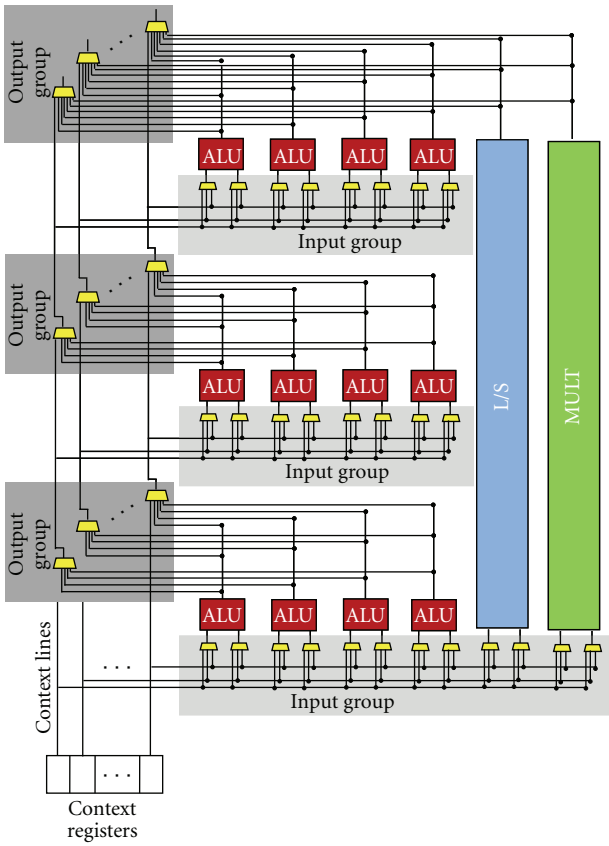
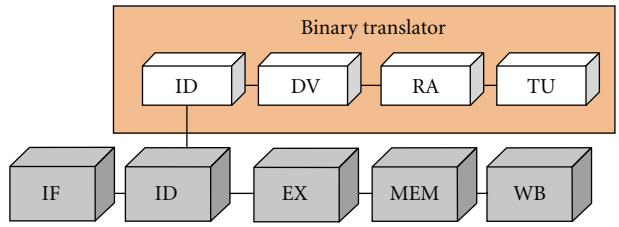


Figure 3: FAT-array interconnection model based on multiplexers and buses.

instructions placed in the same row are executed in parallel, since there are no data flowing from one unit to the other in the same row. Dependent instructions are executed in different rows because data among units flow from bottom to top of the architecture.

There are two groups of multiplexers: the input group that selects data used by each functional unit and the output group that selects the context line that receives the result from the operations performed by each functional unit. Moreover, each output multiplexer also has as input the context register. This input is used when the multiplexer needs only to bypass the previous data without the need of any functional unit.

The different execution times presented by each group of functional units allow the execution of more than one operation per level. Therefore, the FAT-array can perform up to three arithmetic and logic operations that present data dependency among each other in one equivalent processor cycle, consequently accelerating the sequential execution. Moreover, the execution time can be improved through modifications in functional units and with technology evolution, consequently increasing the acceleration of intrinsically sequential parts of a code. Even nonparallel code can have a better performance when executed in the structure illustrated in Figure 2, as shown in [12]. Moreover, the amount of functional units is defined according to area constraints



IF	Instruction fetch	ID	Instruction decode
ID	Instruction decode	DV	Dependency verification
EX	Execution	RA	Resource allocation
MEM	Memory access	TU	Table update
WB	Write back		

Figure 4: Binary translator's pipeline coupled to the processor's pipeline.

and/or an application set demand (given by a certain market, e.g., portable phones).

The Binary Translator (BT) unit implements a mechanism that dynamically transforms sequences of instruction to be executed on the array. The transformation process is transparent, with no need of instruction modification before execution, preserving software compatibility. Furthermore, the BT works in parallel with the processor's pipeline, presenting no extra overhead to the processor. Figure 4 illustrates the Binary Translator's steps attached to the MIPS R3000 pipeline.

2.1. Configuration Generation. To generate the configuration, the first stage of BT's pipeline (ID) consists in searching for sequences of instructions that can be executed in the FAT-array. The second stage (DV) checks data dependency among the current instruction and the previous ones that were already analyzed. Based on data dependency analyzed in previous stage, in the next stage (RA) BT searches for available functional units to perform resource allocation. Both data dependency and resource availability verification are performed through the management of tables that are updated in the last stage (TU). At the end of BT's pipeline a configuration is generated and stored in the context memory indexed by the program counter (PC) value of the first instruction from the sequence.

To check data dependency BT uses two tables to indicate which registers are used in each instruction. One table indexes the input registers and the other indexes the output registers. In this way, BT can verify if the registers used in the current instruction are also used in the previous one, finding RAW (read after write) dependencies.

To perform resource allocation BT uses a resource map that represents the functional units. Considering a defect-free architecture, when BT starts a configuration, all positions of the resource map are set as free. Every time an instruction is placed in a functional unit, its relative position in the resource map is set as busy. Therefore, to select a functional unit, after checking data dependency, BT needs to search in the resource map a free unit and generate the configuration bits for that unit and its multiplexers. If there

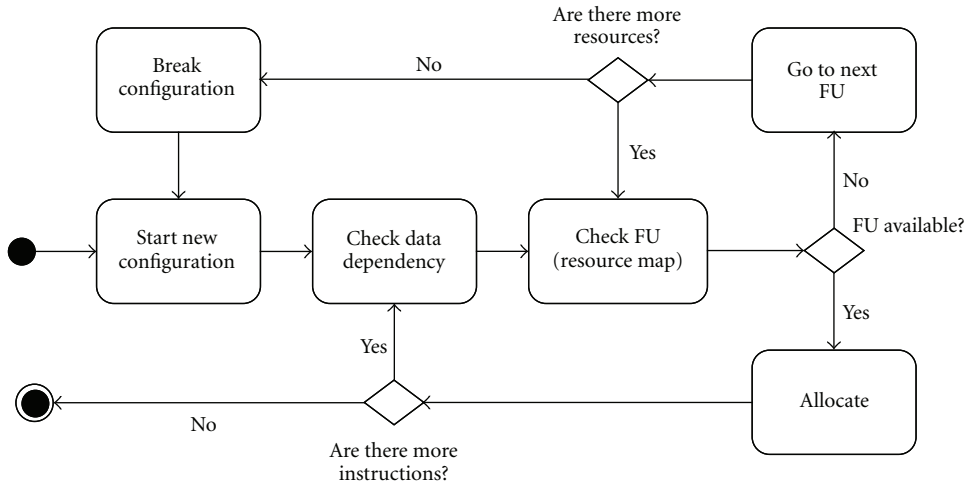


Figure 5: Configuration generation algorithm implemented in Binary Translator (defect-free approach).

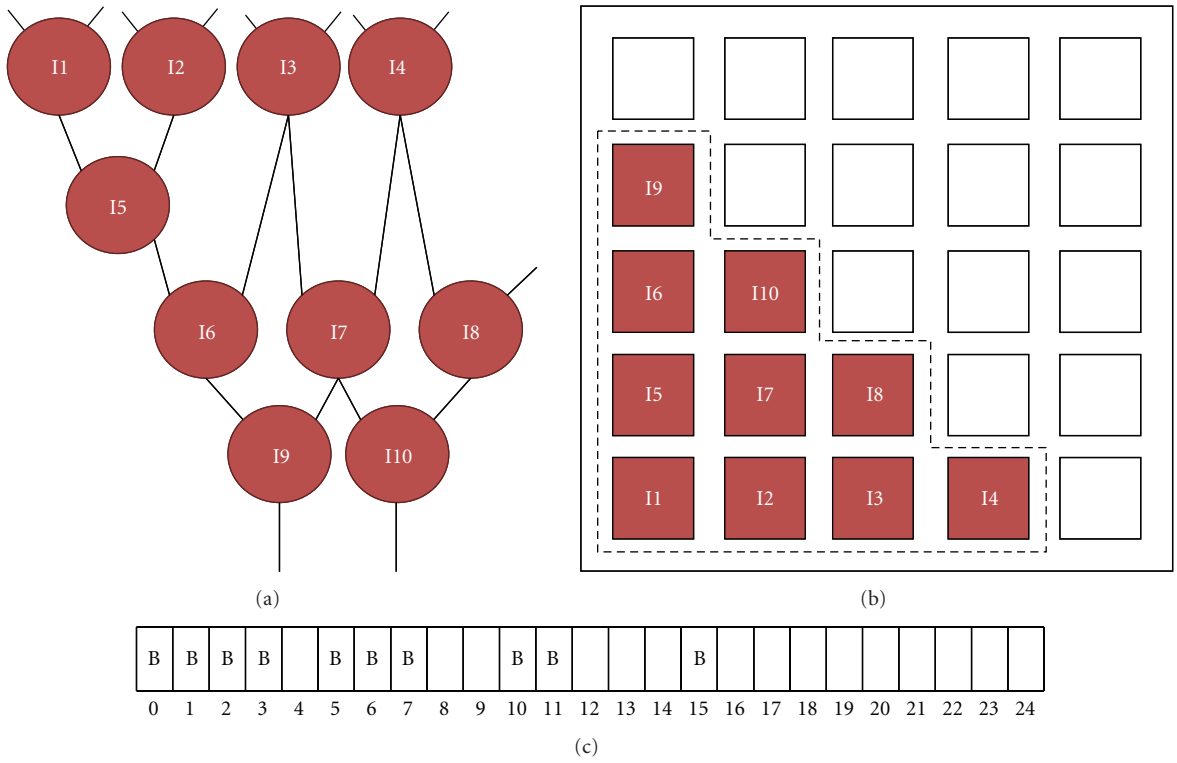


Figure 6: Defect-free resource allocation: (a) data dependency graph, (b) FAT-array resource allocation based on data dependency, and (c) allocated units set as busy in resource map.

are no more available units, then BT breaks the configuration and starts a new one with all units set as free. Figure 5 presents an activity diagram describing the configuration generation algorithm.

Figure 6 presents an example to illustrate how the resources are allocated using the resource map. Figure 6(a) presents the data dependency graph of the instruction sequence mapped to the FAT-array. Figure 6(b) represents the FAT-array with 25 identical functional units, and Figure 6(c) presents the resource map. For clarity's sake, in this example we are considering only arithmetic and logic units. As can be observed, the resource map has 25 positions, each one representing one resource of the array.

As already explained, the parallel instructions are placed in the same row of the FAT-array, and the dependent instructions, which must be executed sequentially, are placed in different rows. To select the functional units to execute the instructions, the mechanism starts by allocating the first available unit (bottom-left). The next instructions are placed according to data dependency and resource availability.

Based on Figure 6(a) the instructions 1, 2, 3, and 4 do not have data dependency among each other; hence they are all placed in the same row of Figure 6(b). On the other hand, instruction 5 is dependent on the result of instructions 1 and 2. Hence, instruction 5 is placed in the first available unit of the second row. Continuing the allocation, the inputs of instruction 6 are the outputs of instructions 3 and 5. Since instruction 5 is in second row, instruction 6 must be placed in third row. Instructions 7 and 8 depend on instructions 3 and 4 that are placed in the first row. Thus, they can be placed in second row. Moreover, since instruction 9 depends on instructions 6 and 7 that are in the third and second rows, respectively, it must be placed in fourth row. Finally, instruction 10 depends on instructions 7 and 8 placed in second row; hence it must be placed in the third row. Therefore, as one can observe in Figure 6(c), the positions of the resource map that correspond to the allocated functional units are set as busy. In this example all the positions are already set as busy but in real execution the positions are filled one per time during run-time.

The approach to avoid and replace defective resources is implemented in the resource allocation step. Section 3 details this approach.

2.2. FAT-Array Reconfiguration and Execution. While BT generates and stores the configuration, the processor continues its execution. Next time a PC from a configuration is found, the processor changes to a halt stage, the respective configuration is loaded from the context memory, and the FAT-array's datapath is reconfigured. Moreover all input data are fetched. Finally, the configuration is executed and the registers and memory positions are written back.

It is important to highlight that the overhead introduced by the FAT-array reconfiguration and data access are amortized by the acceleration achieved by the FAT-array. Moreover, as mentioned before, the configuration generation does not impose any overhead. More details about the reconfiguration process can be found in [12].

3. Defect Tolerance

3.1. Defect Tolerance Approach. Reconfigurable architectures are strong candidates to defect tolerance. Since they consist essentially of identical functional elements, this regularity can be exploited as spare-parts. This is the same approach used in memory devices and has demonstrated to be very efficient [14]. Furthermore, the reconfiguration capability can be exploited to change the resources allocation based on the defective and operational resources.

In addition, dynamic reconfiguration can be used to avoid defective resources and generate new configuration at run-time. Thus, there is no performance penalty caused by the allocation process, nor extra fabrication steps are required to correct each circuit.

Finally, as it will be shown, the capability of adaptation according to the application can be exploited to amortize the performance degradation caused by the replacement of defective resources by working ones.

Since the defect tolerance approach presented in this paper handles only defects generated in the manufacturing process, the information about the defective units is generated before the FAT-array starts its execution, by some classical testing techniques. Therefore, the solution to provide defect tolerance is transparent to the configuration generation.

To demonstrate the approach to tolerate defective functional units Figure 7 presents two examples based on the example of Figure 6.

In a defective FAT-array, the allocation algorithm is exactly like the one described in the example of Figure 6. The only difference is that to allocate only the resources that are effectively able to work, before the FAT-array starts and after the traditional testing steps, all the defective units are set as permanently busy in the resource map, like if they had been previously allocated. With this approach, no modification in the binary translation algorithm itself is necessary.

Figures 7(a) and 7(b) demonstrate the resource allocation considering defective functional units. In Figure 7(a) the configuration mechanism placed the instruction in the first available unit, which in this case corresponds to the second functional unit of the first row. Since the first row still has available resources to place the four instructions, the FAT-array sustains its execution time. In this case the presence of a defective functional unit does not affect the performance.

Figure 7(b) illustrates an example where defective functional units affect the performance of the FAT-array. In this example, the first row has only three available functional units. In this case, when there are not enough resources in one row, the instructions are placed in the next row, and all the data-dependent instructions must be moved upwards. In Figure 7(b), instruction 5 is dependent on instruction 4. Hence, instruction 5 was placed in the next row, and the same happened with other instructions (6 to 10). In this example, because of the defective units it was necessary to use one more row of the FAT-array. Since the sequential path of the FAT-array flows from the bottom row to the top row, the use of one more row leads to the increase of execution time, consequently affecting performance. Figures 7(c) and 7(d) illustrate the resource map of both Figures 7(a) and 7(b) examples, respectively. As one can observe, the defective units have their positions in the resource maps set as busy.

To avoid defective multiplexers from the interconnection model of Figure 3 the strategy can be different depending on which group of multiplexer is affected.

In case of a defective input multiplexer, one of the inputs of the respective functional unit will have invalid data; consequently the result of the operation will be incorrect. Therefore, to keep the defect tolerance approach simple and avoid introduce overhead to the configuration mechanism, the strategy is to consider the multiplexer and its respective functional unit as defectives and place the instruction in the next available functional unit. Figure 8 illustrates the approach to tolerate defective input multiplexers.

As in the case of functional units, it is assumed that all defective multiplexers are detected before BT starts. According to Figure 8, the defective multiplexer invalidates the FU input (dashed line) and this invalidates the functional unit

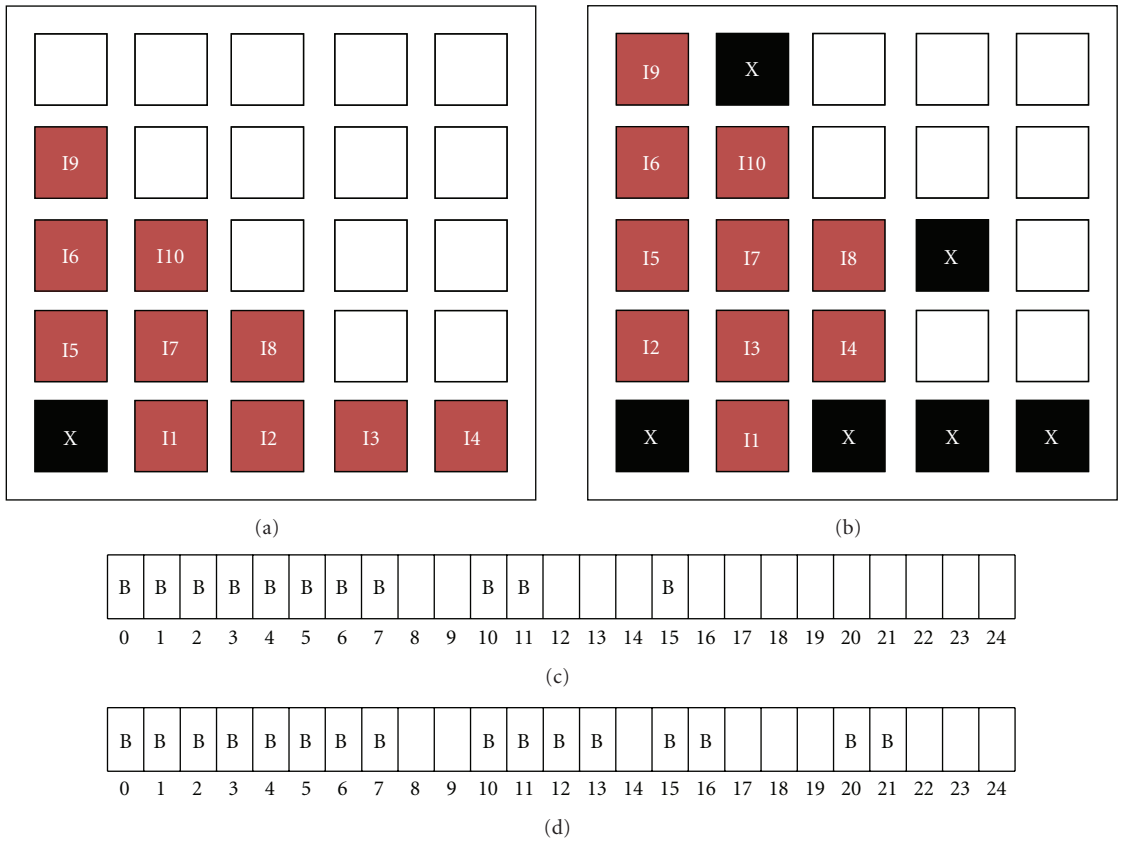


Figure 7: Resource allocation considering defective functional units: (a) not affecting performance, (b) affecting performance, (c) resource map of example (a), and (d) resource map of example (b).

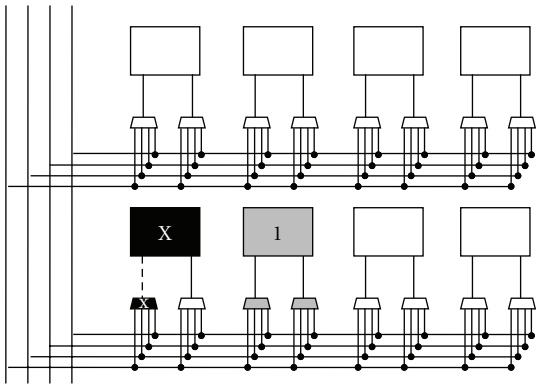


Figure 8: Defective input multiplexer tolerance approach—the defective input multiplexer invalidates the functional unit.

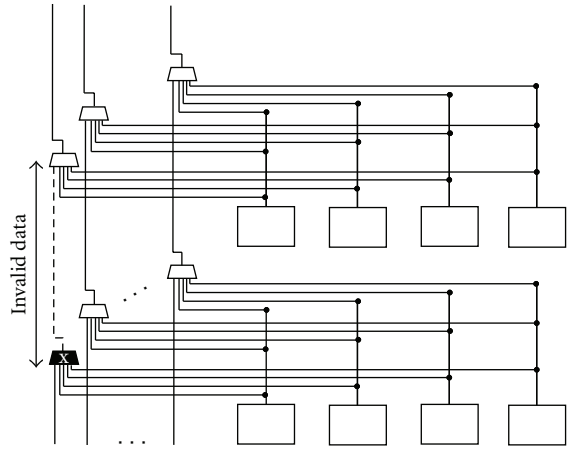


Figure 9: Example of a defective output multiplexer—the defective output multiplexer invalidates the data in the respective context line.

execution. Therefore, the approach is to avoid this FU and to place the instruction in the next available FU (grey modules).

On the other hand, a defective output multiplexer invalidates part of the respective context line. Figure 9 illustrates a defective output multiplexer. For clarity's sake the input multiplexers were omitted. As can be observed in Figure 9, the data remain invalid from defective output multiplexer until the next valid output. The dashed line indicates that this part of the context line has invalid data. The context line will have valid data again when an instruction placed in any row, positioned after the defective multiplexer, writes valid data in this context line.

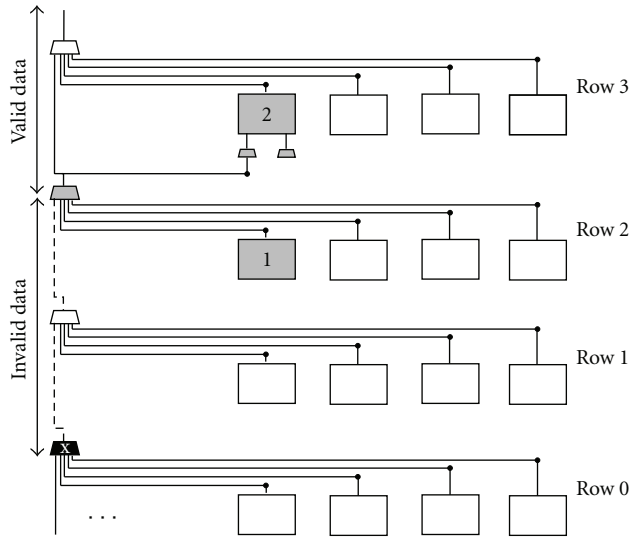


Figure 10: Defective output multiplexer/reading case solution—since instruction 1 has written in context line after defective multiplexer (after row 0) and before instruction 2 reads from this context line (before row 3), then, instruction 2 can read the correct data.

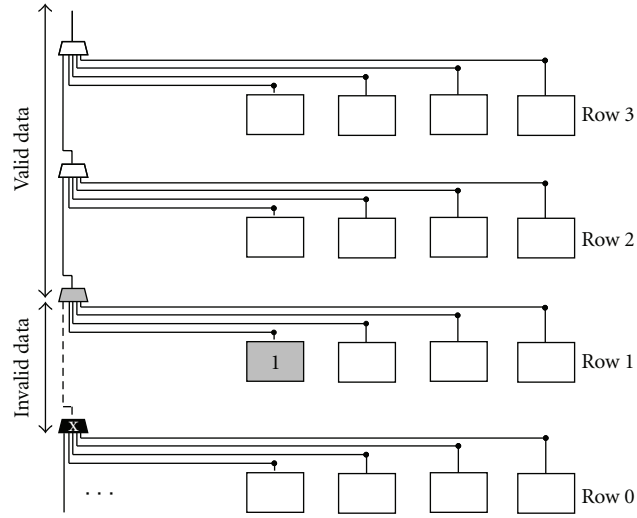


Figure 11: Defective output multiplexer/writing case solution—instruction 1 is placed in the next available row with nondefective output multiplexer (row 1).

Therefore, if an output multiplexer is defective, it is necessary to evaluate if the functional unit needs to read or write in this multiplexer.

When a functional unit needs to read data from a context line, and this same context line has a defective output multiplexer in any previous row, BT must check whether the context line is valid or not. Figure 10 presents an example to illustrate this situation, called reading case.

In the example of Figure 10 instruction 2 that reads from the context line must be placed in row 3. However, the output multiplexer from row 0 is defective. Before placing the instruction in row 3, BT must check if any instruction placed in the rows between row 3 and row 0 wrote in this context line. From Figure 10 it can be observed that instruction 1 placed in row 2 wrote in the context line making this line valid again. Thus, the instruction 2 can be placed in row 3 because it will use the output data from row 2.

In case there is no FU that writes in the context line, this context line remains invalid from the defective output multiplexer until the last row of the array. Thus, the instruction cannot be placed in any functional unit and the configuration must be broken in two configurations. One configuration includes all the instructions that can be placed before the defective multiplexer position and the other configuration will have the other instructions placed in a completely different manner, avoiding the defective multiplexer. Since resource allocation is performed dynamically, there is no predefined order to store data in the context registers. This is the reason that the second configuration can be generated avoiding the defective multiplexer.

The other defect tolerance solution to cope with output defective multiplexers is used when a functional unit needs to write in a context line and the output multiplexer is defective. This case is called writing case and it is illustrated in Figure 11.

The solution to tolerate defective output multiplexers in the writing case consists in placing the instruction in the next available FU. However, since each row has only one output multiplexer for each context line, the strategy in this case is to place the instruction in the next available FU of the next available row. Furthermore, it is also necessary to check if the output multiplexer from the next row is operational. Thus, the instruction can be placed in the next row (as demonstrated in Figure 11) or in any other row with defect-free multiplexer. For example, if the multiplexer from row 1 was also defective, the instruction would be placed in the row 2 and all the instructions dependent on this instruction would be placed from row 3 upwards.

Figure 12 presents the activity diagram summarizing the configuration generation algorithm including the defective unit management. The algorithm is the same presented in Figure 5 with additional steps (grey boxes).

As can be observed in the diagram, the additional steps are responsible for managing only the defective output multiplexers. This is the only case that requires modifications in the algorithm. Moreover, the information about defective functional units and input multiplexers is transparent to BT. This information is translated into the resources map that indicates which functional unit is available and which one is busy. Therefore, the FU that cannot be allocated due to defects is permanently set as busy.

Although some modifications are required to the defective output multiplexer approach, the new steps included in BT are part of the same algorithm already implemented and do not increase the number of cycles required to generate the configuration. Therefore, BT continues working

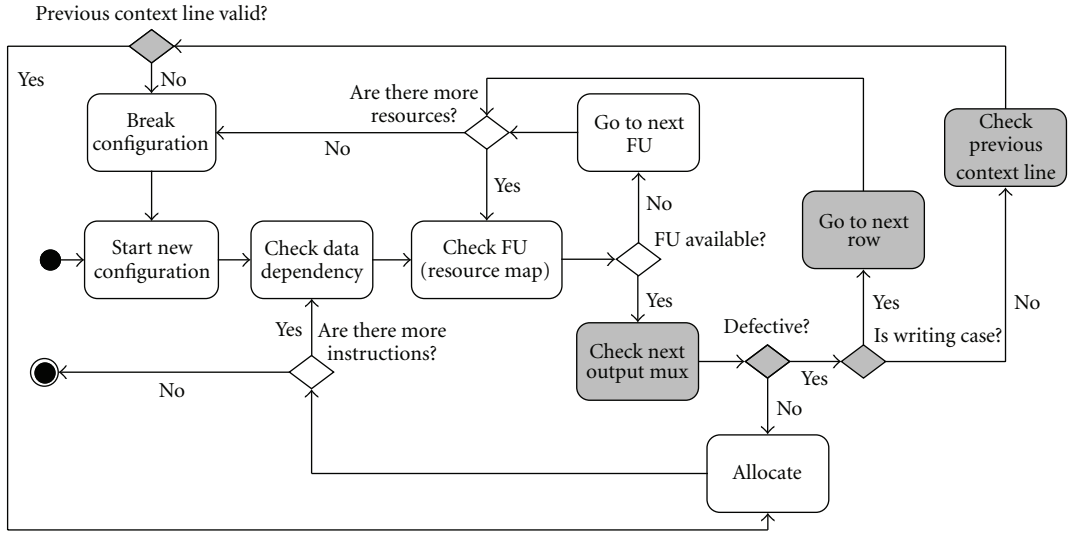


Figure 12: Configuration generation algorithm implemented in BT considering defective units—same steps of defect-free resource allocation (Figure 5) with additional steps for managing defective output multiplexers.

in parallel with the processor's pipeline even with the defect tolerance approach. More important, since BT works in parallel to the processor's pipeline, no performance overhead is incurred while making the binary translation, since it takes fewer cycles than the processor's pipeline, even when defect tolerance is in place.

The defect tolerance approach for functional units and interconnection model was already proposed in [15], where more details about the defect tolerance of interconnection model and experimental results can be found. These details are not in the scope of this paper, since the main focus of the current paper is to demonstrate how the FAT-array with the defect tolerance approach presented in [15] can be an efficient alternative to homogeneous multicores under high defect rates.

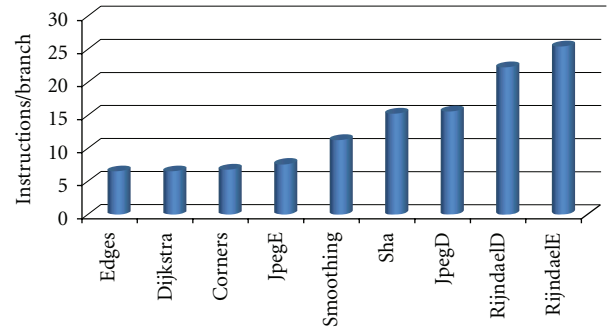


Figure 13: Average of number of instructions executed per branch—*edges* is the most control-flow application and *rijndaelE* is the most data-flow application.

3.2. Experimental Results. To evaluate the proposed approach and its impact on performance, we have implemented the FAT-array in an architectural simulator that provides the MIPS R3000 execution trace. Furthermore, we have chosen a software workload that reflects an embedded system environment. The selected workload is a subset of MiBench suite [16] and covers a wide range of application behaviors. Figure 13 presents the average number of instructions executed between two branch instructions, demonstrating the heterogeneous behavior of the applications. This metric is correlated to the amount of instruction level parallelism available. Therefore, according to Figure 13, *edges* is the most control-flow application, presenting, on average, five operations per branch, and *rijndaelE* is the most data-flow one, with almost 25 instructions per branch.

To include defects in the FAT-array a tool was implemented to randomly select functional and interconnection units as defective, based on several different defect rates. The tool's input is the information about the amount of resources available in the array and its sequential and parallel distribution, as well as the defect rate. Based on this, the tool's output has the same resources information, but now with the randomly selected functional units set as busy and some multiplexers set as defective. This information was used as input to the architectural simulator. In this study we used five different defect rates (0.01%, 0.1%, 1%, 10%, and 20%), and the reference design was the defect-free FAT-array.

Since the functional unit selected as defective must be eliminated from the resource pool, this can affect the performance achieved by the execution of each application. To ensure that the achieved performance was not influenced by one specific defect, we performed several simulations using the same array structure and defect rates, but changing the random defects selection. Thus, the performance results of each set of application, array, and defect rate are actually an average of the performance results of all the simulations that used the same set.

The size of the FAT-array was based on several studies varying the amount of functional units and their parallel and sequential distribution. The studies considered large

arrays with thousands of functional units and also small ones with only dozens functional units. The chosen FAT-array is a middle-term of the studied architectures. It contains 512 functional units (384 ALUS, 96 load/stores, and 32 multipliers). The area of this architecture is equivalent to 10 MIPS R3000.

It is important to highlight that despite the performance degradation presented by the FAT-array system under a 20% defect rate, the performance was still superior to the standalone processor's performance. Hence, Figure 14 presents the speedup degradation of the FAT-array system, instead of the performance degradation.

According to Figure 14, the highest acceleration penalty was presented in the execution of *jpegE*, with 6.5% of speedup reduction under a 20% defect rate. Nevertheless, the FAT-array system is still 2.4 times faster than the standalone processor.

The mean speedup achieved by the defect-free FAT-array in the execution of MiBench applications was 2.6 times. Under a 20% defect rate the mean speedup degraded to 2.5 times. This is less than 4% of speedup degradation.

These results demonstrate that even under a 20% defect rate, the FAT-array combined with the on-line reconfiguration mechanism is capable of not only ensuring that the system remains working but also accelerating the execution when compared to the original MIPS R3000 processor.

3.3. Considerations about Power and Energy Consumption of FAT-Array System. Although technology scaling increases integration capability and consequently performance, another consequence of miniaturization process is the increase of leakage power [17]. According to Sery et al. [18], the leakage power is around 40% of the total power in today's high-performance microprocessors.

Many solutions to increase power efficiency of reduced feature size devices consist in scaling down the supply voltage. However, in order to maintain performance, it is also necessary to reduce the transistor threshold voltage, which in turn increases the subthreshold leakage current. Due to the exponential relationship between the transistor threshold voltage and the subthreshold leakage current, the consequence is the increase of leakage power [19].

To control leakage power the solutions rely on static and dynamic techniques. Static techniques work during the circuit design phase and do not change during operation. On the other hand, dynamic techniques work when the circuit is in idle or in the standby state. One of the most effective dynamic techniques is power-gating. This technique uses sleep transistors to shut down the power supply of parts of the design that are in the idle or standby mode [20].

To cope with power dissipation in the FAT-array architecture, Pereira and Carro [21] propose the inclusion of sleep transistors in each functional unit of the reconfigurable fabric. The approach consists in using the Binary Translator to control the sleep transistors and, at run-time, shut down the idle and defective functional units.

To validate this approach, Pereira and Carro [21] present simulation results evaluating energy consumption of the reconfigurable system with and without sleep transistors. Therefore, a comparison among the MIPS R3000, the full reconfigurable system (without sleep transistors), and the reconfigurable system with sleep transistors in each functional unit is presented. The results demonstrate that the execution of MiBench benchmarks on the reconfigurable system without sleep transistor resulted in an average of 59% of energy saving compared to the execution on MIPS R3000. Moreover, an average of 32% additional energy saving is obtained by implementing sleep transistors in each functional unit.

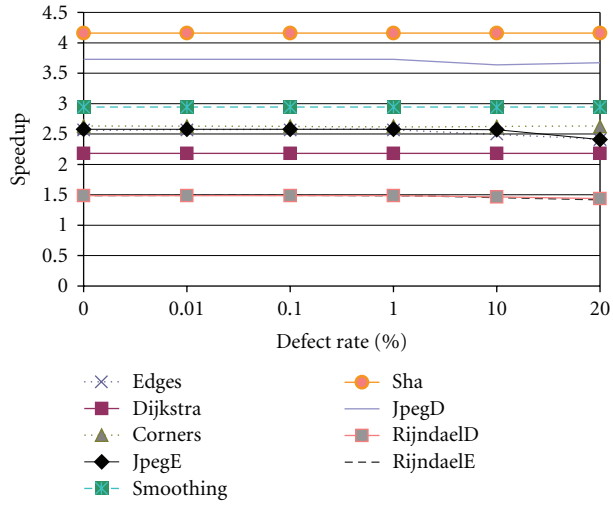


Figure 14: Speedup degradation of the FAT-array system under different defect rates—under 20% defect rate *jpegE* presents the highest speedup degradation with 6.5% of speedup reduction.

It is important to highlight that to control the sleep transistors the Binary Translator has to generate a few more bits to the configuration. Therefore, no significant performance overhead is introduced in the Binary Translator to control the transistors. Moreover, the total area overhead corresponds to two transistors for each functional unit.

4. Adaptive System X Homogeneous MPSoC

4.1. Area and Performance. To demonstrate the efficiency of the proposed architecture this section presents a comparison between the adaptive system and a homogeneous MPSoC with the same area.

As mentioned before, the area of the FAT-array system is equivalent to 10 MIPS R3000 processors, including data and instruction caches. Furthermore, the mean speedup achieved by the reconfigurable system is 2.6 times for the MiBench suite.

The homogeneous MPSoC used to compare area and performance consists of ten MIPS R3000. In this analysis the communication and memory access overheads are not considered. Although the interprocessors communication is not considered, its impact would certainly be higher in the case of the MPSoC; hence all presented results are somewhat favoring the MPSoC.

Table 1: Acceleration as a function of f, $n = 10$.

f	Speedup
0.10	1.099
0.15	1.156
0.20	1.220
0.25	1.290
0.30	1.370
0.35	1.460
0.40	1.563
0.45	1.681
0.50	1.818
0.55	1.980
0.60	2.174
0.65	2.410
0.70	**2.703**
0.75	3.077
0.80	3.571
0.85	4.255
0.90	5.263
0.95	6.897
0.99	9.174
1.00	10.000

As mentioned before, according to Amdahl's law, the speedup achieved by the MPSoC is limited to the execution time of the sequential portion of the application. Equation (1) repeats Amdahl's law for parallel systems:

$$\text{Speedup}(f, n) = \frac{1}{(1 - f) + (f/n)}, \quad (1)$$

where f is the fraction of the application that can be parallelized and n is the number of cores.

Since the MPSoC has ten cores, by varying f and fixing n in 10 (to have the same area of the FAT-array and hence normalize results by area), from Amdahl's law we obtain the results presented in Table 1, where one can see the speedup as a function of f in (1), the part that can be parallelized.

Since communication and memory accesses overheads are not considered, with 10 cores it is possible to achieve a speedup of 10 times if 100% of the application is parallelized.

According to Table 1 it is necessary that 70% of the application be parallelized to achieve a speedup of 2.7 times, which is around the acceleration obtained by the FAT-array system.

Now, one can fix the speedup and vary f to find the number of processors to achieve the required speedup. From (1), varying f from 0.1 to 1 we have that when $f \geq 0.65$, we can achieve the speedup of 2.6. When $f < 0.65$, it is not possible to find a number of cores to achieve an acceleration of 2.6 times. Thus, even with hundreds or thousands of cores, if the application has less than 65% of parallelism, it will never achieve the speedup of 2.6, the same of the FAT-array. Nevertheless, with 65% there would be necessary 19 cores to achieve speedup of 2.6 times, as shown in Figure 15.

Table 2: Sequential acceleration as a function of f.

f	Speedup	AP	AS
0.1	2.60	10	2.402
0.2	2.60	10	2.194
0.3	2.60	10	1.974
0.4	2.60	10	1.741
0.5	2.60	10	1.494
0.6	2.60	10	1.232
0.7	2.60	10	0.954
0.9	2.60	10	0.339
0.99	2.60	10	0.035
1	**2.60**	**10**	**0.000**

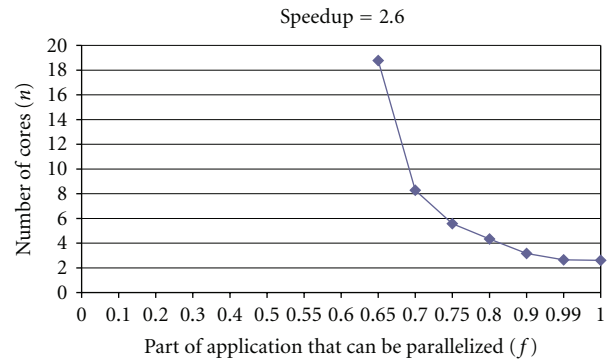


Figure 15: Number of cores as a function of f—according to Amdahl's law, to achieve speedup of 2.6 it is required at least 65% of application parallelism and 19 cores.

One solution to cope with this is to improve the homogeneous core's performance to increase the speedup of the sequential execution. Therefore, one can rewrite Amdahl's law to take this into account, as it is demonstrated in (2). This solution was discussed in [2], where the authors presented the possible solutions to increase performance of a homogeneous MPSoC. They conclude that more investment should be done to increase the individual core performance even at high cost:

$$\text{Speedup}(f, \text{AP}, \text{AS}) = \frac{1}{((1 - f)/\text{AS}) + (f/\text{AP})}. \quad (2)$$

Equation (2) is an extension of Amdahl's law and reflects the idea of improving the MPSoC's overall performance by increasing core performance through acceleration of sequential portions. In (2), AS is the speedup of the sequential portion and AP is the speedup of the parallel portion. Table 2 presents values for AS, fixing the speedup in 2.6 (acceleration given by the FAT-array) and AP in 10 (homogeneous multicore and FAT-array have the same area), while varying f. As one can see in Table 2, only when $f = 100\%$ that AS $= 0$, which means that this is the only case that does not require sequential acceleration. This acceleration cannot be achieved by the homogeneous MPSoC; however as explained in Section 2, the FAT-array can accelerate sequential portions of the application.

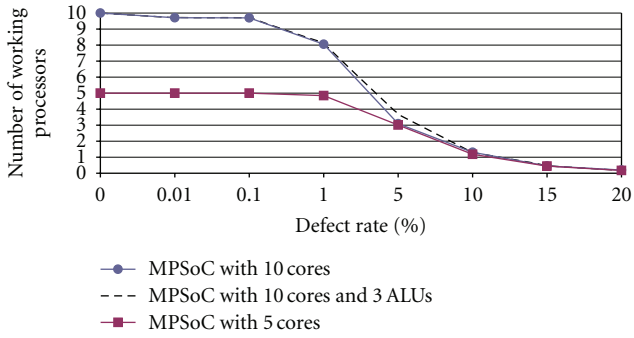


Figure 16: Defects simulation in the MPSoC—simulations performed on a 10-core MPSoC and no fault tolerance solution, on a 10-core MPSoC and 3 ALUs per core, and on a 5-core MPSoC with two processors per core. All cores fail under 15% defect rate.

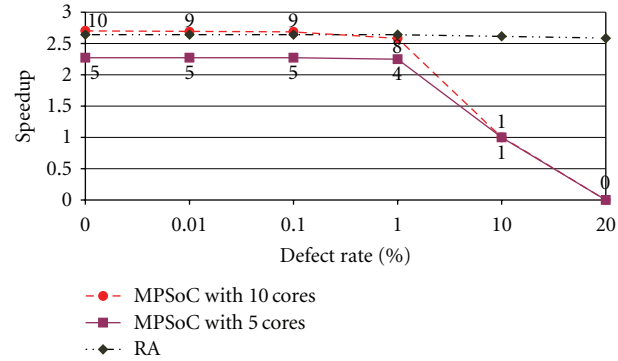


Figure 17: Performance degradation of the MPSoC—considering an application with 70% of available parallelism. Both MPSoC solutions, without fault tolerance approach and with two processors per core, present sequential execution under 10% defect rate and completely fail under a 20% defect rate.

The next section presents a comparison between the MPSoC and the FAT-array considering the fault tolerance capability. The results are normalized by area and speedup.

4.2. Defect Tolerance. This section presents an analysis of performance degradation of both, FAT-array and MPSoC, caused by the presence of defects. This analysis was done through a performance simulation varying the defect rate.

To simulate the defects in both, MPSoC and FAT-array, a tool was implemented to randomly insert defects in both architectures. To ensure that the defect position was not affecting the results, thousands of simulations were performed and in each simulation a new random set of defects was generated. Moreover, the defects generated had the same size (granularity) to both MPSoC and reconfigurable architecture.

In the first analysis we normalized FAT-array and MPSoC by area. In the second analysis we increased the numbers of cores of the MPSoC to evaluate the tradeoff between area and fault tolerance capability.

4.2.1. MPSoC and FAT-Array with Same Area. Figure 16 illustrates the number of cores that remain operating in function of the defect rate in three different studies. The first analysis was performed in a homogeneous MPSoC with 10 MIPS R3000 processors without any fault tolerance approach. According to the results, when the defect rate is 15% or higher, less than 1 core is operating; that is, the whole MPSoC system fails under a 15% defect rate.

The second and third analyses were performed considering that the MPSoC has some kind of fault tolerance solution implemented. In the second analysis, the fault tolerance solution consists in replicating the processor in each core. In this case, instead of having 10 cores with 10 processors, the MPSoC has 5 cores with 2 processors in each core. The second processor works as spare that is used only when the first processor fails. This solution was proposed for two main reasons. First there is no increase in area. Thus, the MPSoC still has the same area of the FAT-array. Second, even with half of the number of cores, the MPSoC still presents higher speedup than the array when the application presents 100% of parallelism.

The solution proposed in the third analysis is also based on hardware redundancy. However, in this case instead of replicating the whole processor, only critical components of the processor are replicated, for example, the arithmetic and logic unit. Therefore, this solution considers that each processor has 3 arithmetic and logic units (ALUs), where 2 ALUs are used as spare. This solution presents lower area cost compared to the solution of the second analysis. However, it can be more complex to implement, since each processor must have an extra unit to implement the fault tolerance approach.

As can be observed in Figure 16, in both second and third analyses, under a 15% defect rate all cores failed. This means that even with fault tolerance solutions, the MPSoC tends to fail completely under high defect rates.

Figure 17 presents the performance degradation of the MPSoC when the number of cores is reduced due to the presence of defects. To obtain the speedup there was used Amdahl's law represented in (1), the MPSoC with ten cores (without fault tolerance), and the one with 5 cores and 2 processors per core. Again, we considered no communication costs, and hence real results tend to be worse. The chart also presents the mean performance degradation of the FAT-array system in the execution of MiBench applications.

This analysis was performed using $f = 0.70$ (the portion of the application that can be parallelized). This number was used because according to Table 1, the speedup achieved by the MPSoC when 70% of the application can be parallelized approaches the speedup achieved by the FAT-array system. The numbers next to the dots in the chart represent the amount of cores that are still working in the MPSoC in function of the defect rate.

As can be observed in Figure 17, the performance of the MPSoC degraded faster than the degradation presented by the FAT-array system, even when the MPSoC presented

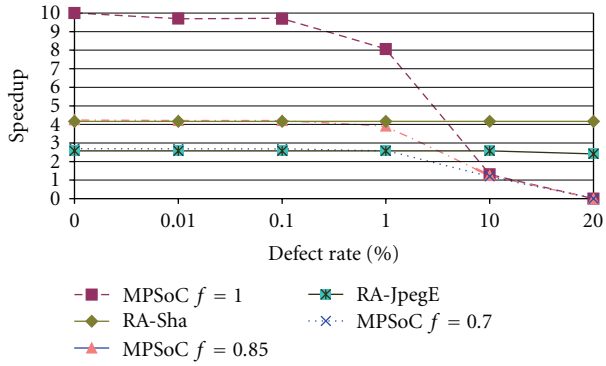


Figure 18: Graceful degradation of *sha* and *jpegE* applications—comparison performed considering applications with different degrees of parallelism running on the 10-core MPSoC without fault tolerance approach.

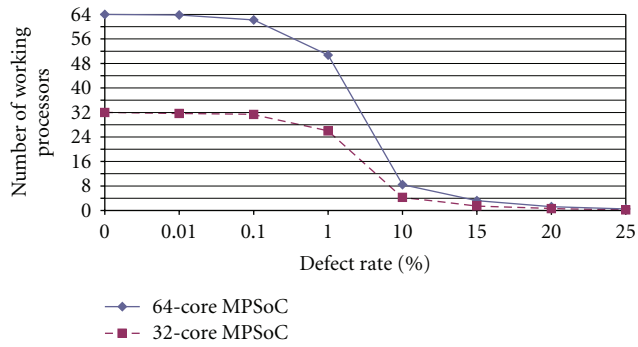


Figure 19: Defect simulation in 32-core and 64-core MPSoC—under 20% defect rate all cores fail in the 32-core MPSoC and only one core remains in the 64-core MPSoC.

higher speedup in a defect-free situation (10-core MPSoC). This happens because when a defect is placed in any part of the processor, the affected processor cannot execute properly. Moreover, even in the MPSoC with five cores and one spare processor in each core the system still degraded faster than the FAT-array. This means that a much more sophisticated approach is necessary to increase fault tolerance of the MPSoC. Nevertheless, this does not guarantee that the system works under high fault rates expected to future technologies. Besides, the more sophisticated is the fault tolerance approach, the more complex it will be, possibly introducing a high area, power, and/or performance overheads.

On the other hand, when a defect is placed in any functional unit or interconnection element of the FAT-array, the run-time mechanism selects another unit (or element) to replace the defective one. According to Figure 17 the execution of MiBench applications by the FAT-array system presented less than 4% of speedup degradation even under a 20% defect rate.

It is important to highlight that in these analyses there was not considered the impact on area and performance that the implementation of the fault tolerance strategies in the MPSoC should introduce. Again the presented results are somewhat favoring the MPSoC. Moreover, the choice of these fault tolerance solutions was based on the idea of causing the minimal impact on the area of the system to maintain both FAT-array and MPSoC equivalent in area.

Figure 18 presents the graceful degradation of applications *sha* and *jpegE*. These applications were selected because the first one achieved the highest speedup by the FAT-array system among all the applications from the MiBench suite and the second presented the highest speedup degradation.

According to Figure 18 the execution of application *sha* by the FAT-array system presented less than 1% of speedup degradation even under a 20% defect rate. On the other hand, even considering that the application was 100% parallelized ($f = 1$), with an initial acceleration higher than the one achieved by the FAT-array system, the 10-core MPSoC stopped working under a 20% defect rate. This same behavior was observed when the amount of parallelism available was reduced ($f = 0.85$ and $f = 0.70$). In these cases, not only did the whole system stop working under a 20% defect rate but also the initial acceleration was equal and lower, respectively, than the one achieved by the FAT-array system.

Moreover, as can be observed in Figure 18, the same behavior was detected in *jpegE* results. However, in this application execution the MPSoC presented higher speedup with $f = 0.85$ than the FAT-array, which rapidly decreased to 0 with the defect rate higher than 1%. On the other hand, the FAT-array sustained acceleration even under a 20% defect rate that presented a speedup degradation of 6.5%.

4.2.2. Increase MPSoC Core Number. Since the FAT-array consists in a large amount of identical functional units that can be easily replaced, the same idea was proposed to the MPSoC: increase the number of cores to increase reliability. Thus, this solution consists in adding more cores to the MPSoC to allow software execution under higher defect rates.

As one can observe in Figure 19, the MPSoCs with 32 and 64 cores still execute under a 15% defect rate. However, in case of 32 cores the execution is completely sequential (one core left under 15% defect rate), and in case of 64 cores, the MPSoC has only 3 cores left under a 15% defect rate. Moreover, under a 20% defect rate the 32-core MPSoC completely fails and the 64-core MPSoC presents a completely sequential execution with only one core left.

The speedup results presented in Figure 20 also demonstrate the rapid decrease in the MPSoC speedup with 32 and 64 cores while the FAT-array sustains acceleration in both *sha* and *jpegE* even under a 20% defect rate.

Based on this result, one can conclude that simply replicating the cores it is not enough to increase the defect tolerance of the system to tolerate high defect rates that new technologies should introduce. Moreover, adding a defect tolerance approach can be costly in area and performance.

To evaluate the area of the system with the reconfigurable architecture used in this work we estimated the size of

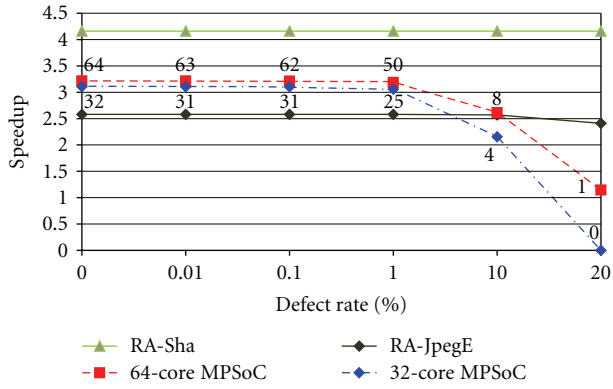


Figure 20: Graceful degradation of 32-core and 64-core MPSoC—under different defect rates and comparing with FAT-array graceful degradation when executing *sha* and *jpegE* applications.

the chip based on the gates number presented by the LeonardoSpectrum from Mentor Graphics [22]. Table 3 presents the chip size (in mm^2) in the 90 nm technology.

Since the reconfigurable fabric is highly tolerant to defects, as demonstrated in the analysis presented above, we also estimated the area of the FAT-array system in scaled technologies that present high defect rates, using the transistor density presented in ITRS roadmap [23]. In this case, since the processor is not tolerant to defects, shrinking its area would increase the probability of this unit presenting defects, and consequently, precluding the operation of the system. Thus, the processor remained the same size in all technologies. The same approach was used with BT, once it is a critical part of the system and it is not tolerant to defects. Therefore, only the reconfigurable fabric was scaled to reduced feature size technologies. Table 3 presents the area estimated according to the technology (in mm^2). It is clear that with technology evolution the area overhead of the reconfigurable fabric gets much smaller, and by using the proposed approach, its reliability does not get compromised by the scaling process.

According to Table 3, the 90 nm system is around 7 times larger than the 11 nm system. This means that if one uses the area occupied by the 90 nm system, it is possible to have 7 cores each one consisting of an MIPS R3000, a BT mechanism and a FAT-array. The combination of MPSoC and reconfigurable architecture allows task level parallelism exploitation through the use of several cores and instruction level parallelism by the reconfigurable architecture. Moreover, sequential arithmetic and logic units can also be accelerated by the FAT-array. This is a very recent research topic. Rutzig et al. [3] proposes an MPSoC architecture that combines general-purpose processors and reconfigurable architectures. However, the study of fault tolerance in this system was not addressed yet.

The analyses presented in this section demonstrate that to future technologies with high defect rates, homogeneous MPSoCs may not be the most appropriate solution. The main reasons are the fact that a defect in any part of the processor invalidates this processor. Thus, the higher

Table 3: Estimated chip size (mm^2).

	90 nm	32 nm	22 nm	11 nm
MIPS R3000	0.2162	0.2162	0.2162	0.2162
BT	0.1216	0.1216	0.1216	0.1216
FAT-array	2.1619	0.0269	0.0135	0.0034
TOTAL	2.4997	0.3647	0.3513	0.3412

is the defect rate, more aggressive is the performance degradation, leading to a completely fail of the system under defect rates already predicted to futures technologies, such as nanowires [6]. Furthermore, solutions to provide fault tolerance in homogeneous MPSoCs under high defect rates can be costly, both in area and performance.

Another disadvantage of homogeneous MPSoCs is the fact that they can only exploit task level parallelism, depending on the parallelism available in each application. Therefore, only a specific application set that is highly parallelized can benefit from the high integration density and consequently the integration of several cores in one chip [24].

There are two main solutions to cope with this. The first one is to use heterogeneous MPSoCs, where each core can accelerate a specific application set [4]. The main problem of this solution is that like the homogeneous multicore, the heterogeneous one must also have some fault tolerance approach to cope with high fault rates, and this can increase area and performance costs.

The other possible solution is to increase the speedup of each core individually. With the improvement of each core it is possible to accelerate sequential portions of code and consequently increase the overall performance of the system. An example of this approach is to change the MIPS R3000 cores for superscalar MIPS R10000 [25]. However, this strategy can result in a significant area increase. According to [3], an MIPS R10000 is almost 29 times larger than the MIPS R3000.

The analyses also demonstrate that the proposed reconfigurable architecture ensures software execution and also accelerates the execution of several applications even under a 20% defect rate. Moreover, the FAT-array system is a heterogeneous solution that accelerates parallel and sequential code. Thanks to this approach, the proposed architecture even exposed to high defect rates predicted to future technologies can still accelerate code, since the parallelism exploitation is not the only way to accelerate execution.

5. Related Work

This section presents some works related to the two main topics of this paper. The first one is fault tolerance approach in reconfigurable architectures. In this context, this section presents some relevant works found in the literature that use reconfigurable architecture as a solution to increase fault tolerance of the systems.

The second topic presented in this section is fault tolerance in MPSoCs. Since this paper proposes to replace a homogeneous multicore for a reconfigurable architecture

as a solution to increase fault tolerance of the system, it is relevant to show that there are many solutions to increase reliability of MPSoCs. However, as mentioned before, the efficient solutions present high overheads, such as increase in area and power and decrease in performance due to the fault tolerance approach; for example, DMR and TMR approaches add at least 2 and 3 times more area and power just to duplicate or triplicate the system, respectively. The area and power consumption are even higher when considering the voters. Moreover, the solutions that focus on a low-cost approach are not as efficient as the more complex ones.

5.1. Fault Tolerant Reconfigurable Architectures. Most works that try to connect reconfiguration with fault tolerance have been developed targeting commercial and fine-grained FPGA devices.

Redundancy exploitation, graceful degradation, and yield enhancement are examples of techniques used in FPGAs, such as the one proposed by Hatori et al. [26]. This work introduced the first fault tolerance solution using redundancy specifically to FPGAs. The authors propose the inclusion of extra rows of logic blocks and extra wiring to be used as spare-parts. Moreover, in this approach reconfiguration is performed during manufacture. Therefore, there is no time overhead due to reconfiguration. However, it tolerates only one fault per row and the strategy consists in eliminating an entire column for one fault. As a consequence, the fault tolerance is low.

Hanchek and Dutt [27] adopt a node-covering method to replace faulty elements. In this approach chains of PLBs (nodes) are created and the last PLB from the chain is used as spare (cover node). When a fault occurs, the node is replaced by shifting all nodes along the row, one position at a time, from the faulty node until the end of the chain. Additional segments (cover segments) are also included to ensure that the moved local routing will be reconnected. Each chain corresponds to a row in an FPGA. Therefore, the approach can tolerate one fault per row with the overhead of one additional column of PLBs.

Lach et al. [28] present an offline approach that partitions the physical design into tiles and replaces the affected tile with a functionally equivalent one. Besides part of the system, each tile contains spare logic and interconnection elements. To replace the tile several precomputed configurations are stored in memory, each one with the spare logic placed in a different position. If a fault occurs, the current configuration from this tile is replaced by a configuration that does not use the faulty resource.

Abramovici et al. [29] propose an online testing, diagnosis, and fault tolerance approach that uses self-testing areas (STARs) of the FPGA while the rest of the device continues its normal operation. In this work the STARs are gradually roved across the FPGA until the entire chip is tested. Although this work proposes a solution that does not require stopping the system, the reconfiguration is performed while the STAR area is nonoperational. Therefore, there is still a performance overhead to reconfigure the parts of the FPGA [30].

Gebelein et al. [31] propose the combination of hardware and software approaches to mitigate radiation effects in FPGAs. At the bottommost FPGA configuration bit layer, the proposed solution consists in combining scrubbing with application of DMR in the CPU. The CPU consists in a soft-core compatible with MIPS-R3000 architecture and instruction set. This solution was implemented in VHDL and tested through live beam tests. Considerations about the use of ECC BRAM blocks and memory refresh scrubber as well as configuration of software layers like operating system and software application to mitigate radiation effects are also presented. Moreover, the reconfiguration capability and area increase due to DMR are used specifically to mitigate radiation effects. They do not allow acceleration of software execution.

Besides the overhead introduced to reconfigure fine-grained FPGAs, some of the presented works involve some form of redesign, which forces the application to stop while reconfiguration is taking place, even if the application is being configured in some other part of the FPGA. Regarding the solutions that use spare-parts, they require the addition of extra hardware or require the user to perform an alternative placement and routing scheme to reconfigure the FPGA. Moreover, solutions that use precomputed configurations are usually limited to a small amount of faults that the approach can cover or require a large amount of different configurations to cope with all the possible faults combination.

The solution presented in this paper manages all these issues during the execution phase, hence it does not require any redesign, and the application is not required to stop. First, the system uses a coarse-grained reconfigurable fabric, which reduces the overhead introduced by the amount of information needed to reconfigure the system. Moreover, to replace defective functional units the same mechanism implemented to dynamically configure the reconfigurable fabric is used; that is, the system is self-adaptive. Thus, it is not necessary to implement several backup algorithms at design time or add extra hardware to specifically perform fault tolerance. Since the reconfiguration is performed at run-time, it is also not necessary to use precomputed configurations as a solution to avoid fault elements. Finally, the fault tolerance approach does not require the addition of spare-parts, since the system will continue its execution, only with less acceleration. The redundancy characteristic of the reconfigurable fabric is used to replace defective elements for operational elements with the cost of a slightly lower acceleration penalty, as demonstrated in last sections.

5.2. Fault Tolerant MPSoCs. Redundancy is the main solution to increase reliability of processors. There are four main redundancy strategies: hardware, software, information, and time redundancy [8]. Since this work proposes a fault tolerance approach through hardware redundancy, the related works presented in this section also use hardware redundancy to increase reliability. To see more details about the other types of redundancy and related work on this topic please refer to [8].

One of the most relevant works in hardware redundancy is DIVA architecture [32]. DIVA uses two processors in the same die to perform fault detection and fault tolerance. The processors are an out-of-order superscalar that executes the leading thread and a simple in-order processor that works as a checker to verify the correctness of all the computations performed by the superscalar. The checker receives all data through the reorder buffer (ROB) of the processor's pipeline. For this reason, the checker does not need to calculate addresses and perform branch prediction, among other tasks, becoming simpler than a processor. When the checker finds inconsistency, it triggers a recovery action. The problem of this approach is the assumption that the checker is error-free, which is acceptable for current technologies but in future technologies this assumption probably cannot be taken. Moreover, the static nature of its hardware prevents it from gaining additional performance when reliability is not warranted.

HP NonStop [10] system was the first commercial system designed to achieve high availability. The main applications of NonStop systems are as follows: credit card authorization, emergency calls, electronic mail, among other applications that require high level of availability, data protection, or scalability [10]. The HP NonStop system was first introduced by former Tandem Computers, Inc in 1976 before it became an HP product [10]. The current system from HP is the NonStop Advanced Architecture (NSAA) that consists in a massively parallel cluster where each independent processor, a 4-way or 8-way SMP Itanium2, runs a copy of the operating system. The system relies on dual or triple modular redundancy (DMR or TMR). According to the authors, the main advantage of the NSAA over previous versions is that the NSAA implements the loose lockstep approach, where each processor can run the same application instruction stream in different clock rates. This allows each processor to do error retries or fix up routine, and to hit or miss caches at different points in time [10]. The NSAA also relies in checksums and error correction codes in network messages and disk data. All implemented techniques cope with transient and permanent faults to provide a correct result. Otherwise the faulty component will stop working and remove itself from the system.

Another fault tolerant approach in multicore systems to cope with permanent faults is the configurable isolation proposed by Aggarwal [11]. This approach consists in splitting the resources in groups, represented by colors, and any fault resource in one group will affect only the cores from that group (with the same color). To avoid that one failure in one group affects resources in the other group, an isolation mechanism is implemented for interconnection, caches, and memory controllers. According to the authors the architecture can be used in different configurations, such as DMR, TMR, or even N-MR configuration.

Many other fault tolerance strategies in processor and multicores architectures can be found in the literature [33–37]. Although some present efficient strategies, such as NonStop [10], and others present low-cost approaches such as DIVA [32], all works that rely on hardware redundancy present area and power overhead. Moreover, many works also present a performance penalty due to the fault tolerance approach. On the other hand, the FAT-array presented in this paper does not present any overhead due to the defect tolerance approach. Moreover, all the redundant hardware included in the system is used to accelerate software execution, and only in case of defects the system can suffer a performance penalty. Nevertheless, as already demonstrated, even under a 20% defect rate the system still remains accelerating.

6. Conclusions

Advances in CMOS technology scaling have increased the integration density, and consequently allowing the inclusion of several cores in one single chip (multicore solutions).

The MPSoC (Multiprocessor System on Chip) architectures allow the acceleration of application execution through task level parallelism exploitation. However, the drawback of this approach is the fact that it is limited to the amount of parallelism available in each application, as demonstrated by Amdahl's law.

One of the solutions to overcome this limit is using heterogeneous MPSoCs, where each core is specialized in different applications set. Another possible solution consists in increasing the speedup of each core individually. These approaches can improve performance but cannot handle high defect rates presented in future technologies.

As a solution to cope with high defect rates and sustain performance this paper presented a run-time reconfigurable architecture to replace homogeneous MPSoCs. The system consists of a coarse-grained reconfigurable array and an on-line mechanism that perform defective functional unit replacement at run-time without the need of extra tools or hardware.

The Fault-Tolerant array (FAT-array) design allows the acceleration of parallel and sequential portions of applications and can be used as a heterogeneous solution to replace the homogeneous MPSoCs and ensure reliability in a highly defective environment.

To validate the proposed approach several simulations were performed to compare the performance degradation of the FAT-array system and the MPSoC using the same defect rates, normalizing the architectures by area and speedup. According to the results, the FAT-array system sustains execution even under a 20% defect rate, while the MPSoC with equivalent area (10-core MPSoC) has all the cores affected under a 15% defect rate. Moreover, analyses considering 32-core and 64-core MPSoCs demonstrated that under a 20% defect rate the former completely fails and the latter presents a sequential execution with only one core left.

Future works include the possibility of coping with transient faults in the FAT-array system.

References

[1] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (MPSoC) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[2] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[3] M. B. Rutzig, F. Madruga, M. A. Alves et al., "TLP and ILP exploitation through a reconfigurable multiprocessor system," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, April 2010.

[4] Samsung Electronics Co., Ltd, "Samsung S5PC100 ARM Cortex A8 based Mobile Application Processor," 2009.

[5] iPhone, http://www.apple.com/iphone/.

[6] A. DeHon and H. Naeimi, "Seven strategies for tolerating highly defective fabrication," *IEEE Design and Test of Computers*, vol. 22, no. 4, pp. 306–315, 2005.

[7] S. Borkar, "Microarchitecture and design challenges for gigascale integration," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO '04)*, p. 3, December 2004.

[8] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*, Morgan Kaufmann, San Francisco, Calif, USA, 2007.

[9] S. K. Shukla and R. I. Bahar, *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[10] D. Bernick, B. Bruckert, P. D. Vigna et al., "NonStop advanced architecture," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 12–21, July 2005.

[11] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*, pp. 470–481, June 2007.

[12] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1208–1213, March 2008.

[13] A. C. S. Beck and L. Carro, "Transparent acceleration of data dependent instructions for general purpose processors," in *Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI-SoC '07)*, pp. 66–71, October 2007.

[14] E. Stott, P. Sedcole, and P. Y. K. Cheung, "Fault tolerant methods for reliability in FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 415–420, September 2008.

[15] M. M. Pereira and L. Carro, "A dynamic reconfiguration approach for accelerating highly defective processors," in *Proceedings of the 17th IFIP/IEEE International Conference on Very Large Scale Integration*, October 2009.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst et al., "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the 4th IEEE International Workshop on Workload Characterization*, pp. 3–14, IEEE Press, December 2001.

[17] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305–327, 2003.

[18] G. Sery, S. Borkar, and V. De, "Life is CMOS: why chase the life after?" in *Proceedings of the 39th Design Automation Conference*, pp. 78–83, June 2002.

[19] J. P. Halter and F. N. Najm, "Gate-level leakage power reduction method for ultra-low-power CMOS circuits," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 475–478, May 1997.

[20] K. Shi and D. Howard, "Challenges in sleep transistor design and implementation in low-power designs," in *Proceedings of the 43rd Annual Conference on Design Automation*, pp. 113–116, ACM Press.

[21] M. M. Pereira and L. Carro, "Dynamically adapted low-energy fault tolerant processors," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 91–97, IEEE Press, August 2009.

[22] MENTOR GRAPHICS, "LeonardoSpectrum™," http://www.mentor.com/products/fpga/synthesis/leonardo_spectrum/.

[23] ITRS, "International Technology Roadmap for Semiconductors," http://www.itrs.net/reports.html.

[24] K. Olukotun, L. Hammond, and J. Laudon, *Chip Multiprocessor Architecture*, Mark D. Hill, 2006.

[25] K. C. Yeager, "Mips R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, 1996.

[26] F. Hatori, T. Sakurai, K. Nogami et al., "Introducing redundancy in field programmable gate arrays," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 711–714, May 1993.

[27] F. Hanchek and S. Dutt, "Node-covering based defect and fault tolerance methods for increased yield in FPGAs," in *Proceedings of the 9th International Conference on VLSI Design*, pp. 225–229, Bangalore, India, January 1996.

[28] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Efficiently supporting fault-tolerance in FPGAs," in *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 105–115, Monterey, Calif, USA, February 1998.

[29] M. Abramovici, C. Stroud, C. Hamilton, S. Wijesuriya, and V. Verma, "Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications," in *Proceedings of the ITC International Test Conference (ITC'99)*, pp. 973–982, Atlantic City, NJ, USA, September 1999.

[30] J. M. Emmert, C. E. Stroud, J. A. Cheatham et al., "Performance penalty for fault tolerance in roving STARs," in *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL '00)*, pp. 545–554, Villach, Austria, August 2000.

[31] J. Gebelein, H. Engel, and U. Kebschull, "FPGA fault tolerance in radiation susceptible environments," in *Radiation Effects on Components and Systems Conference (RADECS '10)*, Längenfeld, Austria, September 2010.

[32] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '99)*, pp. 196–207, November 1999.

[33] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger, "Exploiting microarchitectural redundancy for defect tolerance," in *Proceedings of the 21st International Conference on Computer Design (ICCD '03)*, pp. 481–488, October 2003.

[34] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin, "Tolerating hard faults in microprocessor array structures," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 51–60, July 2004.

[35] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Exploiting structural duplication for lifetime reliability enhancement," in *Proceedings of the 32nd Interntional Symposium on Computer Architecture (ISCA '05)*, pp. 520–531, June 2005.

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

[37] K. Constantinides, S. Plaza, J. Blome et al., "BulletProof: a defect-tolerant CMP switch architecture," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pp. 3–14, February 2006.