On-line pseudo-periodic testing for embedded multiprocessor

Julien Guilhemsang*†, Olivier Heron*, Nicolas Ventroux*
*CEA LIST
PC 94, Gif-sur-Yvette, F-91191 France ;
Email: julien.guilhemsang@cea.fr

Alain Giulieri†
†LEAT
Université de Nice-Sophia Antipolis, CNRS
250, rue Albert Einstein, 06560, Valbonne, France

***Abstract*—Advances in integration technologies have a negative impact on reliability and detecting intermittent faults became a large challenge for complex systems like multicore processors. On-line periodic testing is a good candidate for the detection of intermittent errors but induces lots of preemptions to execute the tests and increase application time. In a multiprocessor implementation, periodic testing has to be aware of busy processors to avoid preemptions. Then the cost of periodic testing can be smaller in a multiprocessor architecture than in a uni-processor. In this case the test of processors become pseudo-periodic and could decrease the detection probability.**

With a probabilistic model, we studied the impact of a pseudo-periodic test on the detection probability. Furthermore, with a simulation of the on-line periodic testing of a multiprocessor architecture, we studied the test intervals variation induced by different implementations of periodic testing. Our results shows that a pseudo-periodic test aware of busy processors induces 20% less preemptions compared to a strict periodic test. We show that on-line pseudo-periodic testing saves performances compare to strict periodic testing with a low impact on the detection probability. On-line pseudo-periodic testing is well adapted to non safety-critical low-cost embedded multiprocessor architectures.

I. Introduction

According to ITRS Roadmap[1], reliability is a more and more critical task for all processor manufacturers in deep-submicron technologies. Reliability is defined as the ability of an IC to work without a failure and within specified performance limits for a specified time and environment[2]. New materials and smaller devices provide benefits for performance and transistor density but, have a negative impact on yield and reliability. Many of billions of transistors in future ICs will be unusable due to extreme static variations. ICs will also experience intermittent errors due to process variations and processor aging[3], [4], [5], combining with dynamic variations of supply voltage and junction temperature[6]. Generally, intermittent errors will move to permanent over the time.

To increase a systems's reliability, fault tolerant concepts must be used. A fault-tolerant system may go through one or more of the following stages[7]: masking, detection, containment, repair/reconfiguration, recovery. In multiprocessor systems, masking is usually not used for low-cost embedded processor, because it requires lots of space. So the first and essential stage to implement is detection. This paper only focuses on this part, but the other stages must be implemented to make the system fault-tolerant.

Hardware reliability techniques such as time and space redundancy offer a very good fault coverage but increase system costs and complexity. On-line periodic testing provides a low-cost detection of intermittent and permanent errors. In this way, the test of a processor is done with hardware-based self-test (HBST) or software-based self-tests (SBST) techniques. Built-in self-test (BIST) techniques are hardware techniques used in production, which provide excellent test quality, at-speed and high fault coverage. But they induce high hardware overhead and high power consumption, which is not compatible with low power embedded processors.

Software-based self-test approaches have been more and more studied by research and development groups during last years [8], [9], [10], [11], [12], [13] for non safety-critical embedded processors. SBST techniques use processor internal resources to perform test generation, test application and results analysis. These techniques are non-intrusive, since it doesn't need to modify the processor and provides high fault coverage.

On-line periodic test using SBST techniques on uni-processors have been studied in [14], [15] and propose interesting alternatives to hardware-based techniques. On-line periodic testing needs to preempt the processor to execute a test. Hence in the uni-processor's implementations the test costs on application duration are directly proportional to the number and the duration of the tests. In multiprocessor architectures, unused processors can be exploited to save performances. However in this case, periodic testing can be subject to test intervals variations, so test becomes pseudo-periodic.

The first part of this paper presents the probabilistic model used to describe intermittent errors, and study the impact of test intervals variation on the detection probability.

The second part presents the multiprocessor model used for our study and shows different implementations of on-line periodic testing of a multiprocessor architecture.

Then the third part presents the simulation results of the different implementations on the performances and on the test intervals variation.

The results of the different implementations combined with the theoretical study will help us to determine the best implementation with a good compromise between performances and detection probability.

II. Theoretical study

A. Intermittent error modeling

A processor subjected to intermittent faults can be in two different states: operating or faulty depending on whether the fault is active or not. A common way to model intermittent errors is the use of a two-state continuous-parameter Markov model[15], [16], [14], [17]. This model allows predicting future states of a system with only the knowledge of the present state. The operating state and the faulty state are named state 0 and state 1, respectively. Let λ and μ denote the rates of leaving state 0 and state 1, respectively. Considering the aging of two systems, with the same λ, the system with the smaller μ/λ is the oldest.

The mean time during which the processor is operational or faulty has an exponential distribution with the parameters $1/\lambda$ and $1/\mu$, respectively. The Markov model defines the transition probability $P_{i,j}(t)$ from the state i to the state j after a time t, as below:

$$\begin{aligned} P_{0,1}(t) &= \frac{\lambda}{\lambda+\mu}(1-e^{-(\lambda+\mu)t}) \\ P_{0,0}(t) &= 1-P_{0,1}(t) \end{aligned}$$

Additionally, the steady-state probabilities of being at state 0 or state 1 for an indifferent observation time , are $\pi_1 = \frac{\lambda}{\lambda+\mu}$ and $\pi_0 = \frac{\mu}{\lambda+\mu}$, respectively.

B. Periodic testing with non constant test intervals

As detection system we use on-line periodic testing of which the efficiency to detect intermittent faults has been showed for years[15], [16], [17]. Generally, test intervals are assumed to be equal but we will show in section III that this is not true in our context. Here, we consider a non-perfect periodic testing for which the test intervals can be different and this part explains how to compute the detection probability.

Let T_n denote the instant of the n^{th} test and let ΔT_n denote the interval between T_{n-1} and T_n. Using the Markov model we can compute $P_d(n)$ the probability to detect a fault after n tests. $P_d(n)$ is equal to the probability that the processor stays at the operational state $P_{0,0}$ during $(n-1)$ tests and the probability that the processor moves to the faulty state $P_{0,1}$ for the last test.

Let $P_{fd}(N)$ denote the probability to detect a fault during every one of the N tests. $P_{fd}(N)$ is equal to the probability to detect a fault after the first test or after the second test, etc, or after the n^{th} test.

$$\begin{aligned} P_d(n) &= P_{0,0}(\Delta T_1)\dots P_{0,0}(\Delta T_{n-1}).P_{0,1}(\Delta T_n) \\ P_{fd}(N) &= \sum_{n=1}^{N} P_d(n) = 1-\prod_{n=1}^{N} P_{0,0}(\Delta T_n) \end{aligned} \quad (1)$$

To be more precise equation (1) must take into account the probability that an existing fault becomes active during the observation time and let $q(t)$ denote this probability. Thus the probability that a fault becomes active during the time interval $[0, T_N]$ and is detected with anyone of the N tests is $d(N) = q(T_N).P_{fd}(N)$.

The complementary probability of $q(t)$ is the probability that the processor stays in the operating state (state 0) during the interval $[0, t]$. This probability is equal to the probability to be at state 0 at $t = 0$ and that the fault remains inactive during the time interval $[0, t]$. The first probability is equal to the steady-state probability of being at the operating state: π_0. The second probability is equal to $e^{-\lambda t}$. Consequently, $1 - q(t) = \pi_0 e^{-\lambda t}$. Finally with equation (1) we have:

$$d(N) = (1-\pi_0 e^{-\lambda T_N}).(1-\prod_{n=1}^{N} P_{0,0}(\Delta T_n)) \quad (2)$$

In the specific case where the periodic testing is perfect and where each test interval is equal to the test period T, $d(N) = (1-\pi_0 e^{-\lambda.N.T}).(1-P_{0,0}(T)^N)$.

C. Test interval variation

Let the test instant variation v denote the difference between the theoretical test time and the effective test time. Let T_n denote the theoretical instant of the test n and T denote the test period, then the effective time of the test n is included in the interval $[T_n; T_n + v.T]$, and the interval between two tests is included in the interval $[(1-v).T; (1+v).T]$.

Figure 1 shows an example of test interval variation for different variation values. The case where $v = 0$ is equivalent to the perfect periodic test. Considering the detection probability, in the example, the case with a variation of 50% corresponds to the worst case. In this case half the test intervals are separated by $(1-v).T$ and the other half by $(1+v).T$. This case corresponds to a test intervals distribution with the mean value equal to T and with a standard deviation of $v.T$. Then if N is the total number of tests, using equation (2) we can compute the detection probability for a given variation in the worst case:

$$d_{wc}(N) = q(N.T).(1-P_{0,0}((1-v).T).P_{0,0}((1+v).T))^{N/2} \quad (3)$$

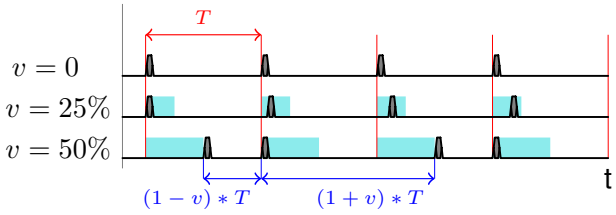

Fig. 1. Test interval variation examples for different variation values

D. Numerical example

This section shows how the detection probability of the periodic testing decreases with the variation of test intervals.

For the Markov model, intermittent faults are defined by two parameters, $1/\lambda$ is the mean time before a fault appears and $1/\mu$ is the mean activation time of the fault. As numerical example we consider three cases: $\lambda = 0.01\ ms^{-1}$ and $\mu/\lambda =$

$100, 10$ and 1 . $\mu/\lambda = 100$ means that the processor is in the operating state 100 times more than in the faulty state.

Given μ and λ, the period of the tests can be computed by different methods, for example [16] and [15] suggest deriving a cost function to determine the optimal test period. Here we define $(1-\varepsilon)$ as requirement for the detection probability after a certain duration time. That duration time must be greater than $1/\lambda$, otherwise the probability to observe the errors is too weak, so here we choose $10/\lambda$. The table II-D shows the test period for different cases, for example, if $\lambda = 0.01\ ms^{-1}$, $\mu = 1\ ms^{-1}$ and $\varepsilon = 10^{-3}$, then the test period required to have $d(N) = 0.999$ after a duration time of $1000\ ms$ is $T = 0.79\ ms$.

The figure 2 shows the impact of the worst case intervals variation on the detection probability for different cases of intermittent errors. For $v = 0$ there is no intervals variation, so the detection probability is maximum and equal for all the cases. If the periodic testing is not perfect and the test intervals variation is equal to 40%, then the detection probability can decrease down to 0.9986 which is less than the requirement of 0.999. If we are able to measure the test interval variation on the processor then we can adapt the test period to take into account the variation and achieve the requirements. For example, if the detection probability requirement is 0.9986, then the periodic test tolerates intervals variations up to 40%.

The next part explains how to implement on-line periodic testing on a multiprocessor architecture. Depending on the implementations and depending on whether performances or detection probabilities have the priority, the test intervals can suffer from variations. This theoretical study evaluates the impact of the test intervals variations on the detection probability, so this will help us determine the best implementation.

Cases	μ/λ	$\mu\ (ms^{-1})$	T (ms)
1	100	1	0.78
2	10	0.1	7.63
3	1	0.01	65.57

TABLE I
CASE STUDY WITH $\boldsymbol{\lambda = 0.01\ ms^{-1}}$ AND $\boldsymbol{\varepsilon = 10^{-3}}$

III. ON-LINE FUNCTIONAL SELF-TESTING OF CMPS

The previous section allows determining a test period for a given intermittent error (λ and μ). This section shows different implementations of a non concurrent on-line periodic test using functional software-based self test in a chip multiprocessor architecture.

We focus on a Chip Multiprocessor Architecture (CMP), made up of a control part, processors and memories. For example, this kind of architecture is used in IBM Cell processor [18] and is a good compromise between performance and power efficiency.

Contrary to symmetric multiprocessor architectures (SMP), the control of CMPs architectures is not executed on the processors used for the application. The control part can be implemented in a processor or in a dedicated hardware. Task scheduling is done dynamically depending on the scheduling algorithm implemented in the control. In our study, the test of the processors has to be transparent for the tasks scheduling. The scheduling of the tasks is evaluated periodically, so at each scheduling tick, tasks can be preempted or moved on any processor. When preemption occurs, the context of the processor is saved in local memories until the task can be processed on any processor.

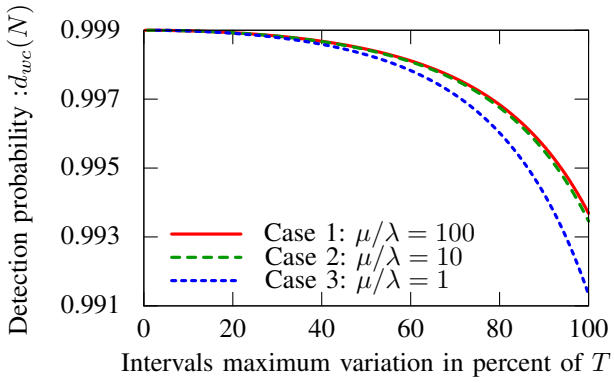

Fig. 2. Test intervals variation impact on the detection probability ($d_{wc}(N)$) for $\varepsilon = 10^{-3}$ after a time duration of $1000\ ms$

Since the use of software-based self-test is targeted the test is considered as a task and is scheduled with non-test applications. Indeed software-based self-test techniques consists in executing self-test programs from on-chip memory to diagnose a processor, so tests are executed at the speed of the processor and responses are stored in the on-chip memory. The studied architecture consists of identical processors, so the self-test routine is the same for all the processors.

Figure 3 shows the scheduling procedure for test tasks and application tasks. Let n_p denote the number of processors in architecture, at each scheduling tick, if a test is running on a processor or a test has to be launched on a processor, then the scheduling of application tasks is done on $(n_p - 1)$ processors. If no test is running and no processor has to be tested, then the scheduling of application tasks is done normally on n_p processors. The choice of the processor to be tested can be done in accordance with different strategies, which are presented in the next section.

A. On-line test scheduling strategies

Software-based Self Test (SBST) [14], [10], [15] is used to test the processors. Test of the processors is done by test tasks which are issued by the control part of the architecture, so test scheduling has to deal with task scheduling algorithm. Moreover processors are tested periodically with a period T. Lets n_p denotes the number of processors in the architecture, then each time instant T/n_p a processor is tested.

Each time instant T/n_p a test tick occurs and a new test task is scheduled on a new processor. But tests must be done on specific processors, so test tasks must be scheduled before application tasks. At each test tick a processor is chosen

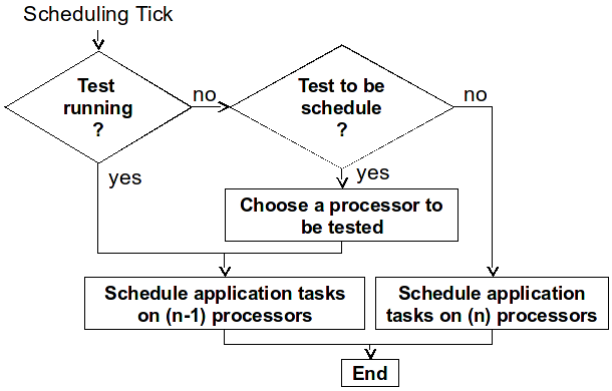

Fig. 3. Scheduling procedure

and this choice has an impact on performances and detection probabilities. Three strategies are presented for the test tasks scheduling.

The first strategy aims at preserving the test period, so test tasks have priority compared to application tasks. At each test tick a new processor is chosen and processors are tested always in the same order. If the selected processor is busy, the task it was executing is preempted and the test task is executed. The figure 4(a) shows an example of test task scheduling for the first strategy. The scheduling algorithm is the following:

```
For p=1 to Number of processor in architecture
Do
  If processor_not_tested(p) Then test(p) End_if
  Else_if all_processors_tested() Then
    init_tests()
  Else try_next_processor(p)
  End_if
End_for
```

The first strategy doesn't take in account the freedom of the processors. The second strategy tries to execute tests on free processors before preempting busy processors. This strategy aims at reducing the number of preemptions compared to the first strategy, but there is an impact on the detection probability. As we can see on the figure 4(b) test intervals can suffer variations. The scheduling algorithm is the following:

```
For p=1 to Number of processor in architecture
Do
  If processor_not_tested(p) Then
    If is_free(p) Then test(p)
    Else
      If no_free_processor() Then test(p) End_if
  Else_if all_processors_tested() Then
    init_tests()
  Else try_next_processor(p)
  End_if
End_for
```

In embedded systems critical tasks are scheduled with common tasks, and there are different impacts if the preempted task has high priority or not. The last strategy takes into account the freedom of the processors and tasks priority. If no processor is free, the task with the minimum priority, among untested processors, is preempted. Test intervals can suffer from the same variations as the second strategy. An example of test task scheduling is shown on the figure 4(b), and the scheduling algorithm is the following:

```
For p=1 to Number of processor in architecture
Do
  If processor_not_tested(p) Then
    If is_free(p) Then test(p)
    Else
      If no_free_processor() Then
  test_untested_processor_with_min_priority(p)
      End_if
  Else_if all_processors_tested() Then
    init_tests()
  Else try_next_processor(p)
  End_if
End_for
```

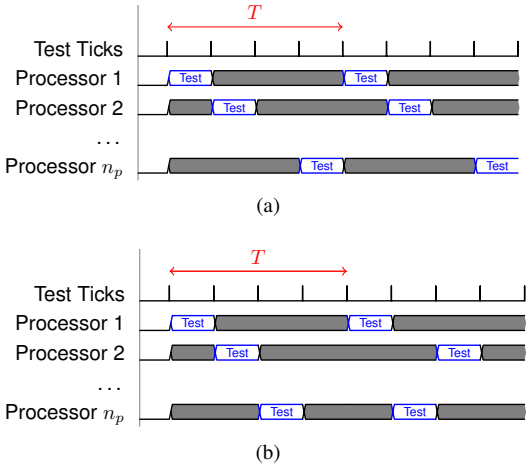

Fig. 4. Test scheduling example for the first strategy (a) and for the second and third strategies (b)

IV. Experimental results

The previous part presents different implementations of the on-line periodic testing on an asymmetric multiprocessor architecture. This part shows the impact of the implementations on performances and on the test intervals variations. These results combined with the theoretical study will give the impact on the detection probability and will help us to determine the best implementation.

The simulation of the architecture is done with SystemC [19] and includes eight MIPS-32 processors with local memories and a control part. Processors are implemented with ArchC [20] and the control part in hardware. The control part uses the Enhanced Least-Laxity-First [21] scheduling algorithm and at each scheduling tick, tasks can be preempted or moved on the different processors.

The error targeted by the test is the same as the error studied in the section II-C in case 2 with $\mu/\lambda = 10$. The number of tests executed on a processor during the application is 131, corresponding to a test period of $T = 7.63\ ms$ during an application time of $1000\ ms$.

In order to analyse the behaviour of the different strategies, simulations are done for different values of architecture occupation. The higher the architecture occupation is, and the smaller the number of free processors over the time is.

A. Impact on performances

On-line periodic testing has an impact on application duration for two reasons: tests duration and preemptions due to tests. The tests duration and the number of tests are the same for all the strategies, so the impact on application duration is exactly the same. Difference comes with the number of preemption induced by the strategies, so we focus on the number of preemptions during the application to compare the impact of the strategies on the performances.

Figure IV-A shows the preemptions for the different test scheduling strategies and for different values of architecture occupation.

As expected, the first strategy shows the most important number of preemptions, since busy processors are not taken into account. With a low architecture occupation, the number of preemptions is approximately equal to the number of tests executed on all the processors in the architecture.

Application parallelism is adapted to the number of processors in the architecture, then if a test is executed on a processor, the application has to deal with one processor less. When occupation increases, applications tasks preempt themselves so the total number of preemptions increases also.

The number of preemptions for the other strategies is equal and saves 20% of preemptions compared to the first strategy. So these strategies achieve our needs concerning performances.

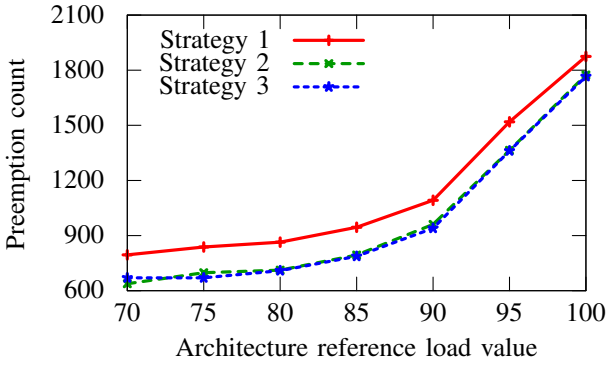

Fig. 5. Preemption number in function of test scheduling strategies

The standard deviation of the test instants is a measure of the variability of the test instants. If tests are strictly periodic then the standard deviation is equal to 0, otherwise the test is pseudo-periodic and there is an impact on the detection probability, as explained in the section II-C.

The figure 6 shows the higher standard deviation between all the processors for the three strategies as a function of the architecture occupation.

As we can see the first strategy induces no variability on the test instants whatever the architecture occupation, and on all the processors. This strategy represents a strict periodic test and the detection probability is not affected by this strategy.

Concerning the third strategy, we notice that the maximum standard deviation on all the processors is less than 50% of the test period, if the architecture occupation is less than 90%, which correspond to normal operating conditions. This means that the time between two tests fluctuates between $T + T/2$ and $T - T/2$, which can be compared to the worst case of detection probability studied in section II-C. If the architecture occupation if higher than 90%, the standard deviation grows up to 66%.

The second strategy induces more variability than the third. The maximum standard deviation on all the processors is up to 82%, which represents a large variability. This strategy has a more important impact on the detection probability than the third strategy.

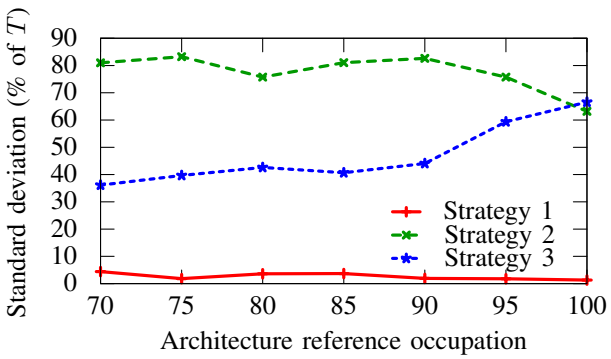

Fig. 6. Higher test instant variability between all the processors

B. Impact on detection probability

Since we focus on periodic testing, the detection probability depends on the number of tests executed and on the distribution of the test intervals during the simulation. For each simulation, the number of tests executed is the same, so we focus on the test intervals distribution to compare the impact of the strategies on the detection probability.

At the end of each simulation and on each processor, the mean time between two tests and the standard deviation of the test instants are computed. The mean time between two tests must be as close as possible to the test period. In all our simulations, the difference is always less than 1% so this parameter has a very low impact on the detection probability.

C. Discussions

Choosing a strategy is always a compromise between performances and detection probability. In our case the first strategy preserves the test period on all the processors but produces 20% more preemptions than the other strategies. The third strategy produces less preemptions than the first strategy but infers more variability on the test instants. These variations are always less than 50% of the test period, in normal operating conditions, which corresponds to a reduction of 0.1% of the detection probability. So the third strategy is a good candidate as compromise between performances and detection probability.

V. Conclusion

First, with a probabilistic model of intermittent fault, we studied the impact of a pseudo-periodic test on the detection probability. We showed that pseudo-periodic testing has a low impact on the detection probability.

Secondly, with a simulation of the on-line periodic testing of a multiprocessor architecture, we studied the test intervals variation induced by different implementations of periodic testing. Our results show that a pseudo-periodic test aware of busy processors induces 20% less preemptions compare to a strict periodic test.

We show that on-line pseudo-periodic testing saves performances compared to strict periodic testing with a low impact on the detection probability. On-line pseudo-periodic testing is well adapted to non safety-critical low-cost embedded multiprocessors.

References

[1] Intl. Sematech, "Critical Reliability Challenges for The International Technology Roadmap for Semiconductors," Tech. Rep. Tech. Transfer 03024377A-TR, 2003.

[2] G. Gielen, P. De Wit, E. Maricau, J. Loeckx, J. Martin-Martinez, B. Kaczer, G. Groeseneken, R. Rodriguez, and M. Nafria, "Emerging yield and reliability challenges in nanometer cmos technologies," *Design, Automation and Test in Europe, DATE'08*, pp. 1322–1327, 2008.

[3] S. Borkar, "Microarchitecture and design challenges for gigascale integration," in *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 3–3.

[4] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *DAC '03: Proceedings of the 40th annual Design Automation Conference*. New York, NY, USA: ACM, 2003, pp. 338–342.

[5] C. Constantinescu, "Impact of deep submicron technology on dependability of vlsi circuits," *IEEE Dependable Systems and Networks*, pp. 205–209, 2002.

[6] ——, "Impact of intermittent faults on nanocomputing devices," *Proc. IEEE/IFIP DSN-2007 (Supplemental Volume), Edinburgh, UK*, pp. 238–241, 2007.

[7] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[8] F. Corno, M. Reorda, G. Squillero, and M. Violante, "On the test of microprocessor IP cores," pp. 209–213, 2001.

[9] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *DAC '03: Proceedings of the 40th annual Design Automation Conference*. New York, NY, USA: ACM, 2003, pp. 548–553.

[10] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *Computers, IEEE Transactions on*, vol. 54, no. 4, pp. 461–475, April 2005.

[11] S. Guramurthy, S. Vasudevan, and J. Abraham, "Automated mapping of pre-computed module-level test sequences to processor instructions," in *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, Nov. 2005, pp. 10 pp.–303.

[12] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, p. 398.

[13] P. Parvathala, K. Maneparambil, and W. Lindsay, "Frits - a microprocessor functional bist method," in *Test Conference, 2002. Proceedings. International*, 2002, pp. 590–598.

[14] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 1, pp. 88–99, Jan. 2005.

[15] N. Kranitis, A. Merentitis, N. Laoutaris, G. Theodorou, A. Paschalis, D. Gizopoulos, and C. Halatsis, "Optimal periodic testing of intermittent faults in embedded pipelined processor applications," *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, pp. 1–6, March 2006.

[16] T. Nakagawa and K. Yasui, "Optimal testing-policies for intermittent faults," *Reliability, IEEE Transactions on*, vol. 38, no. 5, pp. 577–580, Dec 1989.

[17] S. Su, I. Koren, and Y. Malaiya, "A continuous-parameter markov model and detection procedures for intermittent faults," *Computers, IEEE Transactions on*, vol. C-27, no. 6, pp. 567–570, June 1978.

[18] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation cell processor," pp. 184–592 Vol. 1, Feb. 2005.

[19] Initiative, O.S.C., "SystemC community," *World Wide Web Document*, 2006.

[20] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, 2005.

[21] J. Hildebrandt, F. Golatowski, and D. Timmermann, "Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems," in *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, 1999, pp. 208–215.