ANALYSIS OF ON-LINE SELF-TESTING POLICIES FOR REAL-TIME EMBEDDED MULTIPROCESSORS IN DSM TECHNOLOGIES

O. Heron[*], *J. Guilhemsang*[*] *N. Ventroux*[**] *A. Giulieri*[***]

(*) CEA, LIST, Embedded System Reliability Laboratory, PC 94, Gif-sur-Yvette, F-91191, France
(**) CEA, LIST, Embedded Computing Laboratory, PC 94, Gif-sur-Yvette, F-91191, France
email: olivier.heron@cea.fr
(***) LEAT, Univ. de Nice-Sophia Antipolis, Valbonne, F-06560, France. email: alain.giulieri@polytech.unice.fr

ABSTRACT

Advances in DSM technologies have a negative impact on yield and reliability of digital circuits. On-line self-testing is an interesting solution for detecting permanent and intermittent faults in non safety critical and real-time embedded multiprocessors. In this paper, we describe and evaluate three scheduling and allocation policies for on-line self-testing. We show that a policy that periodically applies a test procedure to the different processors in a way that considers idle times, test history of processors and task priorities offers a good trade-off between performance and fault detection probability.

1. INTRODUCTION

ITRS Roadmap [1] predicts that integrated circuits (IC) reliability will become a critical task for semiconductor industry in Deep Sub-Micron (DSM) technologies (sub-45nm). Reliability is defined as the ability of an IC to work without a failure and within given performance limits for a specified time and environment [2]. New materials and smaller devices provide benefits for performance, power consumption and transistor density but have a negative impact on yield and reliability. Many billions of transistors in ICs will be unusable due to extreme static variations. ICs will also encounter frequent intermittent errors due to their increasing sensitivity to external neutrons, alpha particles, transient voltage variations (such as IR drop) and junction temperature variations (such as thermal hotspots) [3][4][5]. Transistors, vias / contacts and wires will age and degrade faster over time, causing intermittent errors due to the parameter variations and even more permanent errors [6], thus shortening IC lifetime. For all the above reasons, reliability becomes a key issue in architecture design.

Design techniques using fault tolerance concepts help to overcome yield and reliability related errors in digital IC. A fault-tolerant IC may go through one or more of the following stages: error masking, error detection, error correction / containment, IC repair / reconfiguration, and IC recovery [7]. Several fault tolerance techniques for multiprocessors were proposed in the past, such as [8][9]. Most of these solutions do not address explicitly error detection; they rather focus on the recovery procedure. In addition, most of them are not suitable for our purpose that concerns non-safety critical and real-time applications such as Media and Telecommunication in embedded multiprocessors. In this context, the multiprocessors rather integrate hundreds of RISC processors that have a less complex micro-architecture (scalar pipeline and non-multithreading support). All the processor cores are occupied by a SW task most of the time (no spare) therefore optimizing performance and power efficiencies (mW/um^2 and Gops/s/um^2).

The design of a cost-effective error detection solution for embedded multiprocessors still appears to be an open problem and hence a critical focus in DSM technologies. Some past works focused on the on-line error detection problem in single RISC processors. “Razor” [10] allows detecting and recovering from delay errors, with the aid of special pipeline latches. “Diva” [11] and “Argus” [12] integrate checkers in the micro-architecture for detecting control and data flow errors. The “Bulletproof pipeline” [13] inserts BIST modules for testing faults in the pipeline stages during stall times. These solutions prevent from committing most or all errors but to the detriment of micro-architecture design effort, area and power consumption overheads.

Real-time and non safety critical applications do not need on-line error detection solutions that would track HW errors cycle by cycle in the most aggressive scenario. Conversely, if an active error escapes from the faulty resource and propagates through the architecture, then it may not always cause a fatal multiprocessor failure. Such applications can naturally tolerate the occurrence and propagation of one or more undetected errors, such as infrequent computing errors. Depending on the expected reliability level, the circuit may only be checked at boot time and sometimes during the execution. A boot checking mainly addresses permanent errors while a checking during execution can also detect intermittent errors that can cause bursts of computing errors or fatal errors (e.g. program counter errors in a processor).

As an alternative to the solutions listed above, on-line self-testing seems to be an interesting candidate that can fit well with our error detection requirements. Test is a common technique used for detecting faults in IC [20]. It is generally applied during manufacturing process (offline testing), at boot time or periodically during the IC lifetime. For the last case, we talk about on-line self-testing for which the test is automatically applied by the circuit itself [18]. This approach requires a lower design effort, induces lower area and power consumption overheads and can detect several fault models with a high coverage, but in detriment of error detection latency [16]. Under test mode, the circuit is configured in a way where the resources under test are logically isolated from the rest of the system. When the test is performed concurrently with the application, the latter should continue its execution on the remaining resources. The control part of a multiprocessor architecture should be able to dispatch task applications and tests over the architecture in a way that minimizes the performance penalties and maximizes the test efficiency.

In this paper, we explain how on-line self-testing can be integrated in an embedded multiprocessor architecture and we describe three possible self-test scheduling and allocation policies for detecting intermittent and permanent faults. With the help of simulation, we evaluate the impact of these policies on performance and fault detection probability in an asymmetric multiprocessor.

The paper is organized as follows. In Section 2, we motivate our approach. In Section 3, we describe the principles of on-line self-testing in an embedded multiprocessor. In Section 4, we detail a self-test configuration. Section 5 describes three self-test policies. Section 6 presents the results about performance and fault detection probability for an asymmetric architecture. Finally, Section 7 concludes the paper.

2. MOTIVATIONS

In a multiprocessor context, on-line self-testing consists in testing frequently a sub-part of the architecture while the rest of the resources still run the jobs, concurrently. Li and al. [16] propose a solution for detecting stuck-at and delay faults in OpenSPARC T1 multiprocessor. High quality test patterns are generated (offline) and stored in an external non volatile memory. An HW support automatically loads and applies the test patterns to every core periodically while others remain in a normal mode. In addition, the proposed solution enables the prediction and diagnosis of such “hard” failures. One limitation the solution is that they do not address explicitly the scheduling and allocation problems for decreasing the performance and power penalties induced by the test. In [17], authors reserve a spare processor for test purpose so as, when the operating system activates a test session on a processor that is running a task, the task is preempted and migrated to the spare processor. This solution can reduce the performance penalty depending on the preemption and migration costs but to the detriment of performance and power efficiencies. In a general way, most of papers address the integration and application problems of the on-line self-testing. To our knowledge, there are no papers that address explicitly the questions: at which rate the test should be applied? And which processor is chosen at each test time? In this paper, we focus on the configuration and scheduling and allocation policies for on-line self-testing in real-time embedded multiprocessors.

The design of an optimal on-line self-testing technique for embedded multiprocessors should offer a good trade-off between fault coverage, fault detection probability, HW design effort and performance and power consumption overheads. Fault coverage is the number of faults that a given test pattern sequence can detect among the total number of possible faults, for different fault models [20]. High fault coverage for several fault models increases the likelihood for detecting a multiprocessor failure, but results in a long test pattern sequences. On-line self-testing should address both permanent and intermittent faults. While permanent faults are mainly due to aging phenomenon such as NBTI, HCI, TDDB, etc. [14], intermittent or temporary faults are more complex failure modes. Such faults are caused by the extreme variation of timing margins of long paths above the limit thus causing a timing violation of the register setup and hold times [3][4][15]. In very DSM technology, the root cause may be due to a transient variation of the junction temperature of the processor that reaches a certain value (hotspot) in a aggressive operating mode (voltage, frequency) or a transient variation of the power supply voltage resulting from the power-on of a neighbour processor that shares the same power lines (IR drop). Note that a test can only detect reproducible faults, thus transient faults cannot be addressed by it.

The fault detection probability at time t is the probability of detecting an active fault with any of the n tests applied till time t. The probability for detecting a fault varies depending on the test period and both activation duration and occurrence frequency probabilities of the fault (we assume a high test quality). As an example, for a given fault detection probability, an intermittent fault with a low activation duration probability and a low occurrence frequency probability results in a lower test period than a fault with a high activation duration probability and a low occurrence frequency probability – that tends to behave as a permanent fault. In [6], authors present an analytical solution for designing the optimal on-line self-testing period in uni-processor for detecting a processor failure with a predefined fault detection probability.

HW design effort for integrating on-line self-testing resources in the architecture is relatively minimal and quasi non intrusive in the processor core design. Actually, the test resources used for manufacturing test can be re-used such as scan chains, JTAG, BIST, test pattern decompression and test patterns [16][18]. A global test controller should be built on the top of these features for selectively configuring and activating the needed test resources depending on the scheduling and allocation decisions. In the paper, we especially focus on the test of processors. It can be performed with hardware-based self-test (HBST) using Built-in-self-test (BIST) techniques and JTAG support, or software-based self-test (SBST) techniques [21]. Due to paper size limitation, both test implementation and application problems will be not addressed here. Note that the test of processors with SBST techniques can also detect faults in the resources used by test (memories and interconnect) but with a lower fault coverage.

Performance and power consumption overheads caused by the test require an important attention. Low-power aware test pattern generation and design-for-test technique can reduce power consumption and temperature impacts [19]. Relatively to energy consumption, the duration of a test task is often lower than that of application tasks. In the paper, we explicitly focus on the application performance problem even though we will consider power consumption problem in the next work steps.

On-line self-testing can be viewed as a single or multiple SW tasks that can run periodically and possibly concurrently with a user application. A test task is characterized by a length (number of cycles) and a period at which the test task should be applied. Compared to a running application task, a running test task will be never stopped before its normal end and it will never share the resources under test with the other tasks. Moreover, the resources under test should be logically isolated from the rest of the system for preventing the propagation of an active error.

Different “application vs. test” execution scenarios can appear, depending on the required reliability level. Firstly, the test tasks can be only executed at boot time, without any user application. Secondly, the test tasks can be executed at runtime as well, but after stopping the user application. Thirdly, as an alternative to the previous one, the test can be executed concurrently during the user application execution. The last scenario enables a higher fault coverage than the two first ones because it enables the detection of intermittent faults that can cause bursts of computing errors or fatal errors in the control. Compared to the second scenario, the last one also reduces the performance penalty because the application continues its execution, even if it will run in a degraded mode. In that case, the test tasks have to be scheduled and allocated to processors concurrently with the user application. The performance penalty will depend on the rate at which processors will be tested for the highest fault detection probability and the test length. Low test period and length values imply a low performance penalty but also low fault detection probability and fault coverage levels.

3. ON-LINE SELF-TESTING IN MULTIPROCESSORS

We address the problem of on-line self-test scheduling and allocation policies in an asymmetric architecture. Note that the principles we will develop in this section can be extended to any multiprocessor architecture. This architecture is composed of P processors with local memories, a shared memory and a programmable network, such as a multi-bus, that allows any processor to read/write data in the shared memory or peripherals with an identical time (uniform memory access).

A centralized control part manages tasks execution, memory allocation and communication between the tasks. Only one task can run on a processor at a time (no multi-threading support). The control is able to preempt a running task and even more it is able to migrate a running task from one processor to another one, especially for real-time and power consumption considerations. In this paper, we only consider the case of data-flow applications. However, this limitation does not restrict the scope of the following demonstration. An application control graph is loaded in the control part. It represents the data / control dependencies between the tasks. A task is a sequence of instructions that can run to completion independently of other tasks. A task is either in the suspended state (not scheduled) or runnable state (task is ready to run) or running state (it is running on a processor core).

We now describe the principle of the on-line self-testing control in this type of architecture. It goes through test configuration, test task scheduling and allocation, and test observation. First of all, a Test configuration function sets the length and period values of the test task. The period value is derived from the expected fault detection probability required by the application, as it will be explained in Section 4. The test length can be a unique value for all of the processor cores or a different value for each one. The test length can be predefined at design time, with the help of ATPG and fault simulation tools [20]. In addition, the value can be updated during the lifetime of the architecture. In that case, the value is a parameter of the control part that is loaded at boot time.

At each control tick period, a Scheduling function determines the priority between the runnable tasks and the running tasks. If one or more test tasks are runnable, the function can set a higher priority to the test task(s) or a lower priority than the application tasks, depending on the implemented policy. In the former case, a test task will be next allocated to a processor. One consequence is that a runnable application task that might have run is here delayed and will be re-scheduled at the next tick period. In the latter case, the test task will be delayed to the next control ticks. This will decrease the likelihood for detecting an active fault.

Thirdly, an Allocation function allocates the HW resources to the scheduled tasks from the task having the highest priority to the lowest. The number of allocated tasks is equal to the number of processor cores (fault-free). The policy is based on a simple algorithm, as follows. The allocator first selects the task having the highest priority. If the task was not running, it tries to allocate it to a free processor, if available. If no free processor is available, the allocator preempts the running task that has the lowest priority and loads the candidate task. The algorithm is illustrated in Figure 1.a. Relatively to the test, the allocator does not allocate it to a processor already under test. The allocation policy should distribute the test tasks over the different processors in a way that enables the same fault detection probability between the processors. Note that it should not allocate it to a processor core that was tested at the previous test period. The function will also control the logical isolation of the resources under test from the rest of the system and the test application (test program and data loading in memories, test application start/stop). In a general way, the Allocation function also allocates memory segments and sets the predefined processor power mode (voltage & frequency) that can be viewed as task parameters, as well.

Fourthly, when the test normally ends, a signal coming from the processor under test triggers the Test observation function that gathers and analyzes the test results. The control part of the architecture will next reallocate the resources under test to the application tasks (PASS result) or applies confinement / repair actions (FAIL result) and even more recovering actions (out of scope of this paper).

4. SELF-TEST CONFIGURATION

In this section, we define the self-test configuration that is an important parameter for reliability control. Self-test configuration determines the period at which the tasks should be scheduled for execution and the length of the test which is proportional to the number of test patterns.

As a starting point of our work and for reader convenience, we assume that all the processors are identical relatively to the probability of a fault occurrence, independently of the way the application is dispatched in the architecture. All the processors have to be tested with identical test period and length. Therefore, we consider that only one periodic test task is scheduled at a time. The rate at which the test task is scheduled depends on the expected fault detection probability on each processor. Let's call T_p the ideal test period of each processor for a given detection probability objective. The period T of the test task will be equal to T_p/P, where P is the total number of processors.

The value of the test period T_p depends on the expected fault detection probability. The analytical equation of this probability is derived in [6][24]. Here, we only summarize the main steps. Firstly, we define a processor failure model. A processor subject to faults can be either in operating or faulty state, depending on whether the fault is active or not. We assume that the occurrence of the first fault causes a processor failure. The two-state continuous-parameter Markov model allows predicting future states of a system with only the knowledge of the present state. The operating and faulty states are named state 0 and state 1, respectively. Let λ and μ denote the rates of leaving state 0 and state 1, respectively. The mean time during which the processor is operational (faulty) has an exponential distribution with the parameters $1/\lambda$ ($1/\mu$). $1/\lambda$ represents the occurrence frequency probability and $1/\mu$ represents the activation time probability of a fault (faulty state). The Markov model defines the transition probability $P_{i,j}(t)$ from the state i to the state j after at a time t, as follows:

$$P_{0,1}(t) = \frac{\lambda}{\lambda+\mu} * \left(1 - e^{-(\lambda+\mu)*t}\right) \qquad \text{[eq. 1]}$$

$$P_{0,0}(t) = 1 - P_{0,1}(t) \qquad \text{[eq. 2]}$$

Where $P_{0,0}(t)$ is the probability to remain in the operating state (0). Additionally, the steady-state probabilities of being at state 1 (0) for an indifferent observation time is $\pi_1 = \lambda/_{\lambda+\mu} \left(\pi_0 = \mu/_{\lambda+\mu}\right)$.

Let's now denote T_n as the begin time of the nth test and $\Delta T_n = T_n - T_{n-1}$ as the time interval between the nth and n-1th tests. The fault detection probability at time T_n can be expressed as follows. It is the probability that a fault becomes active during the time interval $[0\ ,\ T_n]$ and the probability that the active fault is detected with any of the n tests applied since time 0:

$$d(T_n) = \left[1 - \pi_0 * e^{-\lambda * T_n}\right] * \left[1 - \prod_{i=1}^{n} P_{0,0}(\Delta T_i)\right] \qquad \text{[eq. 3]}$$

Where the first term represents the probability that the processor reaches the faulty state (state 1) at time T_n, assuming it was in the operating state at time 0. The second term is the probability of detecting a fault with any of the n tests, assuming that the fault coverage of a test is ideally equal to 100%.

5. SELF-TEST POLICIES

Relatively to the self-test configuration described above, we derive three typical self-test policies that offer different trade-offs between the performance penalty and fault detection probability. Let's denote T_k as the tick period of the control part i.e. the period at which all of the tasks are scheduled and allocated (application and test tasks). Let's denote T as the period at which a test task is woken up (runnable state). T is necessarily an integer multiple of T_k.

5.1. Aggressive self-test policy

This policy periodically applies a test to the different processors with a constant and identical period value, whatever the processor usage. This strategy guarantees the expected period for all of processors to the detriment of the performance.

At each tick $T_k = a*T$ (integer $a > 1$), a test task is in a runnable state and must be allocated to a processor according to a round-robin policy. For that, let's consider that the scheduler provides an allocation table of size P that stores the application tasks to be allocated to processors. The tasks are ordered from the highest priority to the lowest priority by the scheduler. If processor 'i' must be tested, the application task stored in the i^{th} table index is replaced by the test task. Next, all non test tasks are (re-)allocated to the P processors. If the replaced task in the table was running, then it is pre-empted; its execution is stopped and its context is saved before loading the test task. If the task was only in a runnable state, then it will be re-scheduled in the next control tick T_k.

Figure 1 illustrates a scenario example with three processors when no test is applied (Fig 1.b) and when a test task is allocated at a rate of T (Fig 1.c). Note that the real test period of each processor T_p is equal to T*P, where P is the number of processors. For preventing the pre-emption of application tasks with a high priority, the application tasks in the table are re-ordered according to the test history of the processors. The application task with the highest priority is stored in the table index corresponding to the most recently tested processor, and so on.

5.2. Idleness aware self-test policy

Compared to the previous scenario, this policy uses the idle state of the processors to execute a test task before preempting a running application task.

The control checks if the allocation table contains an empty index (e.g. index j). If there is one, it next checks if the corresponding processor j was already tested since it was in the idle state. Actually, let's assume that an application only uses P-1 processors among P. The P^{th} processor is always free (idle state). If no verification is performed, then it will happen that the free processor will be always tested while the others will remain untested. If no free processor can be found, the test task is inserted in the allocation table according the procedure described in 5.1. Figure 1.d illustrates a scenario example with three processor cores. Note that the application task allocation is here modified according to the description of Section 5.1.

The real test period of each processor may vary over time. The time difference between two consecutive tests of a same processor will be greater or equal to Tp. Consequently, the resulting fault detection probability may differ from one processor to another one. In this scenario, the test of a free processor may be also extended to performance and power characterizations.

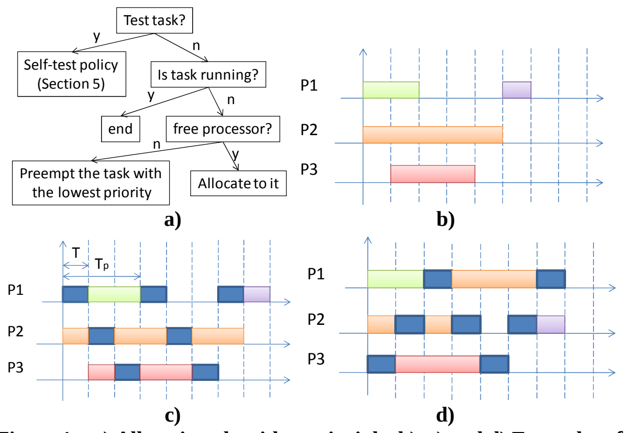

Figure 1 – a) Allocation algorithm principle. b), c) and d) Examples of self-test policy application. Blue blocks represent test tasks while the other blocks are application tasks.

5.3. Idleness&Priority aware self-test policy

Compared to the previous policy, the application task priorities are considered when the test task is inserted in the allocation table, instead of the round-robin algorithm. When no free processor can be tested, the application task with the lowest priority in the allocation table is replaced by the test task. The assignation of the test task to the processor depends on the scheduling policy used for deriving the allocation table with the ordered application tasks.

6. RESULTS

In this section, we compare the impact of the three self-test policies relatively to the performance penalty and the resulting fault detection probability of each processor. The evaluations are made with a functional simulation platform that simulates an asymmetric multiprocessor composed of eight MIPS-like Instruction Set Simulators [22] and a control part that implements the Enhanced Least-Laxity-First scheduling policy [23]. The architecture is modelled with SystemC language. The simulator accounts for the number of executed instructions in each processor and the number of preemptions.

The analysis is performed with a virtual application that allows us to control the architecture load from 30% to 95%. It is the average percentage of the occupation of all processors by the application tasks (without test) over the time. For a given application duration equal to 1s, we consider two types of task lengths: a long task length (125ms) that corresponds to 8 tasks per processor and a short task length (10 ms) that corresponds to 100 tasks per processor. We consider the case a fault characterized by μ/λ=10 (μ= 0.1ms^{-1}). Application requires that this type of fault must be detected with a probability of 99.9% at the end of execution. From equation 3 (Section 4), T_p= 7.69ms (ideal test period of processors) and T= 961.53us. The total number of tests per processor will be equal to 130.

For each self-test policy, we consider two different test lengths: L1= 0.15ms and L2= 0.3ms that are approximately the test length of a MIPS processor with a SBST approach [21]. Note that this value can be back-annotated from ATPG tools and fault simulation [20] after gate level synthesis. Conversely, the designer may explore the impact of different self-test lengths and may express an acceptable maximum test length to test engineers, as an input. We perform several simulations. Each simulation corresponds to a triplet (self-test policy, architecture load, test length). The 1st policy corresponds to the one described in Section 5.1, the 2nd one to Section 5.2 and 3rd one to Section 5.3.

Figures 2.a) and 2.b) show the application duration overhead in % over the architecture load relatively to the three different self-policies of Section 5. Figure 2.a) shows the results for short tasks and the two different test lengths while Figure 2.b) considers long tasks. The 1st self test policy causes a highest penalty than the 3rd one. The difference increases significantly with long tasks. When the architecture load remains below 85%, the overhead remains almost constant. When an application task is pre-empted by a test task, the control finds most of the time a free processor on which it can migrate the application task. Above 85%, the overhead grows with a steeper slope. The number of idle states is not enough for mitigating the effect of the preemption, thus causing a higher penalty.

Figures 2.c) and 2.d) show the total number of preemption occurrences. Figure 2.c) considers short tasks while Figure 2.d) considers long ones. When no test runs, the number of preemptions is zero. In both figures, whatever the self-test policies, the number of preemption occurrences increase with the load value. The preemption of a task with a high priority (due to test) implies most of the time, the preemption of the one with the lowest priority. This phenomenon is exacerbated when the number of tests per task increases with the architecture load growth.

For an identical test length in each policy, the 1st policy causes the highest pre-emption count while the 3rd policy causes the lowest one, whatever the task length. In both figures, the impact of the test length on the pre-emption count is more significant on long tasks than short ones. When the duration of tests per task increases, the number of preemptions increases. This phenomenon is exacerbated with the architecture load growth.

Figures 2.e) and 2.f) show the fault detection probability of the architecture (average of the fault detection probability value of each processor) relatively to architecture load and self-test policy. The results are computed at the time of the last test execution (T_n). Figure 2.e) shows the results with short tasks while Figure 2.f) shows the results with long ones. For each load value, the vertical bars on the curves represent the maximum deviation between the average value and the probability value obtained on the eight processors.

In both figures, the fault detection probability remains close to the reference probability (99.9%). The maximum deviation below the reference is equal to 0.3% (Figure 2.f). 1st self-test policy remains strongly constant. The variations of the average value over the load and the deviations are due to the variation of the real test period on each processor over the time.

In both figures, the 1st policy provides the highest average fault detection probability than the 3rd one, until a load of 80%. Above this value, the benefit of the 2nd and 3rd policies is quasi identical. As we can show, long tasks have a more significant impact on the fault detection probability than short tasks. We can also show that the length of tests has a similar effect (not shown here for clarity considerations). The reason is due to the application duration overhead relatively to the number of tests. While the number of tests remain quasi constant, the time difference between the tests is greater with when long task (or long test lengths) and so, the time Tn (from eq. 3).

For measuring the trade-off between performance and fault detection probability, we compute the ratio between the preemption count and the fault detection probability for each architecture load (short task length). Figure 3 plots the obtained ratio over the architecture load for the three self-test policies, for short tasks. Whatever the architecture load, we note that the 3rd policy (Priority&Idleness aware policy) always has the lowest value. For long tasks, the position of the curves will remain identical, at least with load values below 80%. As a result, a multiprocessor that periodically applies a self-test procedure to the different processors in a way that considers the idle times, task priorities and test history of processors offers a good trade-off between performance and fault detection probability.

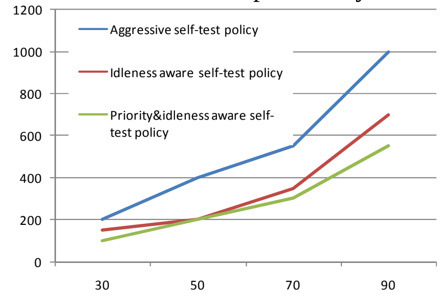

Figure 3 – (preemption count)/(detection probability)

7. CONCLUSION

In this paper, we described how on-line self-testing can be controlled in a real-time embedded multiprocessor for non safety critical applications. We analyzed the impact of three on-line self-testing policies in terms of performance penalty and fault detection probability in a simulated asymmetric architecture. It was shown that a policy that periodically applies a test to each processor in a way that accounts for the idle states of processors, the test history and the task priority offers a good trade-off between the performance and fault detection probability. The evaluations performed in this paper considered actually the particular case of an asymmetric architecture with predefined scheduling and allocation policies. However, the principle and methodology can be generalized to other multiprocessor architectures.

8. REFERENCES

[1] Intl Sematech, "Critical Reliability Challenges for the International Technology Roadmap for Semiconductors", 2003.

[2] G. Gielen et al., "Emerging yield and reliability challenges in nanometer CMOS technologies", DATE'08, pp. 1322-1327, 2008.

[3] Shekhar Borkar, "Microarchitecture and Design Challenges for Gigascale Integration", Int. Symp on MicroArch., 2004.

[4] Shekhar Borkar et al., "Parameter variations and impact on circuits and microarchitecture", DAC'03, pp. 338-342, 2003.

[5] C. Constantinescu, "Impact of deep submicron technology on dependability of VLSI circuits", IEEE DSN, pp. 205-209, 2002.

[6] N. Kranitis et al., "Optimal Periodic Testing of Intermittent Faults In Embedded Pipelined Processor Applications", DATE '06, pp. 1-6, 2006.

[7] Victor P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts", Computer, vol. 23, pp. 19-25, 1990.

[8] Philip M. Wells et al., "Adapting to intermittent faults in multicore systems", ASPLOS '08, pp. 255-264, 2008.

[9] M. Prvulovic et al., "ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors", Int. Symp. on Computer Architecture, pp. 111-122, 2002.

[10] D. Ernst et al., "Razor: a low-power pipeline based on circuit-level timing speculation", MICRO 36, pp. 7-18, 2003.

[11] T.M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design", MICRO'99, pp. 196-207, 1999.

[12] A. Meixner etal., "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores", IEEE MICRO, vol. 28, pp. 52-59, 2008.

[13] Smitha Shyam et al., "Ultra low-cost defect protection for microprocessor pipelines", ASPLOS'06, pp. 73-82, 2006.

[14] Renesas Tech., "Semiconductor Reliability Handbook", Nov. 2008.

[15] Kypros Constantinides et al., "Architecting a reliable CMP switch architecture" ACM Trans. Archit. Code Optim., vol. 4, 2007.

[16] Y. Li, S. Makar, and S. Mitra, "CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns", DATE'08, pp. 885-890, 2008.

[17] H. Inoue, Yanjing Li, and S. Mitra, "VAST: Virtualization-Assisted Concurrent Autonomous Self-Test", ITC 2008, 2008, pp. 1-10.

[18] M. Nicolaidis and Y. Zorian, "On-Line Testing for VLSI - A Compendium of Approaches", J. Electron. Test., vol. 12, pp. 7-20, 1998.

[19] Patrick Girard, "Survey of Low-Power Testing of VLSI Circuits", IEEE Design and Test of Computers, vol. 19, pp. 82-92, 2002.

[20] Samiha Mourad and Yervant Zorian, "Principles of testing electronic systems", Wiley, ISBN 978-0-471-31931-3, 2000.

[21] D. Gizopoulos et al., "Systematic Software-Based Self-Test for Pipelined Processors", Trans. on VLSI Sys., vol. 16, pp. 1441-1453, 2008.

[22] J.L. Hennessy et al., "Computer architecture: a quantitative approach", Morgan Kaufmann, 2003.

[23] J. Hildebrandt et al., "Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems", EUROMICRO, pp. 208-215, 1999.

[24] S.Y.H. Su et al., "A Continuous-Parameter Markov Model and Detection Procedures for Intermittent Faults", IEEE Transactions on Computers, vol. C-27, pp. 567-570, 1978.

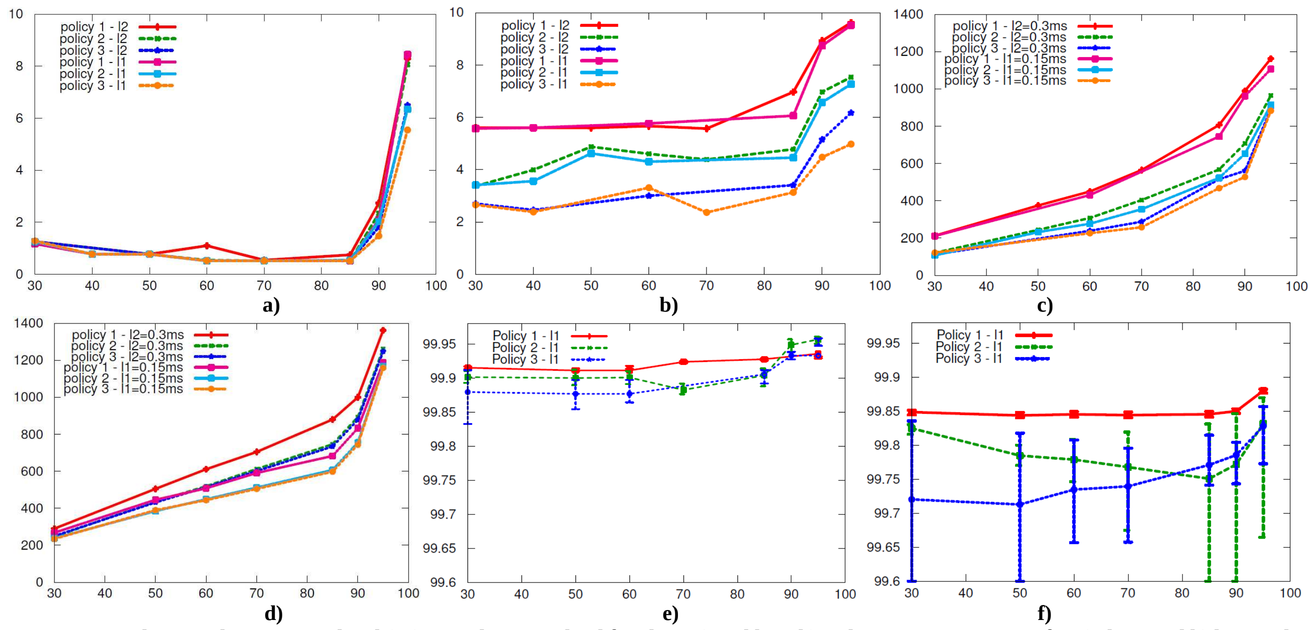

Figure 2 – Application duration overhead (%) vs architecture load for short a) and long b) tasks. Preemption count for a) short and b) long tasks. Fault detection probability of architecture (%) for e) short tasks and f) long tasks.