Static Probabilistic Timing Analysis in Presence of Faults

Chao Chen
École Polytechnique de Montréal
chao.chen@polymtl.ca

Luca Santinelli
ONERA, Toulouse
luca.santinelli@onera.fr

Jerôme Hugues
ISAE, Toulouse
jerome.hugues@isae.fr

Giovanni Beltrame
École Polytechnique de Montréal
giovanni.beltrame@polymtl.ca

***Abstract*—Accurate timing prediction for software execution is becoming a problem due to the increasing complexity of computer architecture, and the presence of mixed-criticality workloads. Probabilistic caches were proposed to set bounds to Worst Case Execution Time (WCET) estimates and help designers improve system resource usage. However, as technology scales down, system fault rates increase and timing behavior is affected. In this paper, we propose a Static Probabilistic Timing Analysis (SPTA) approach for caches with evict-on-miss random replacement policy using a state space modeling technique, with consideration of fault impacts on both timing analysis and task WCET. Different scenarios of transient and permanent faults are investigated. Results show that our proposed approach provides tight probabilistic WCET (pWCET) estimates and as fault rate increases, the timing behavior of the system can be affected significantly.**

I. Introduction

A time-critical computing system, such as a satellite on-board computer, requires accurate timing prediction of software execution. If events are not managed within a certain time frame, the result may be catastrophic. A conservative estimation on execution time for traditional deterministic architecture will place the Worst Case Execution Time (WCET) far away from the actual maximum time used by the application [1].

To help predicting timing behavior, probabilistic real-time systems were introduced and such systems have very few pathological cases [2]. One method to realize probabilistic system is to modify the behavior of the cache – a bridge between processor and main memory – and make it random [2], which provides overall tighter bounds due to the lack of pathological cases. Two timing analysis techniques are proposed for systems with random caches: the Measurement Based Probabilistic Timing Analysis (MBPTA) and the Static Probabilistic Timing Analysis (SPTA). While MBPTA is based on repeated testing of an application for estimating its timing probability distribution, the SPTA uses detailed knowledge of software and hardware for obtaining a precise timing analysis with safe timing bounds.

As transistor size decreases, circuits become more sensitive to transient faults [3] which affect system timing behavior. Transient faults might might be caused by high local temperatures or radiation effects, such as package alpha decay or galactic cosmic rays impacts, even at the ground level [4]. Furthermore, wear-out effects can introduce permanent faults throughout the lifetime of a device. Such effects are also exacerbated by technology scaling [5]. As a result, reliability to permanent and transient faults has to be considered for timing analysis.

In this paper, we present an SPTA methodology for instruction caches with random replacement policy, that takes both transient and permanent faults into consideration. Our methodology takes single-path program memory traces as inputs and computes probabilistic WCET (pWCET), i.e. exceedance probabilities with respect to execution time (the number of processor cycles in our simulations). The calculation is performed using state space techniques, and it is based on a non-homogeneous Markov chain model [6]. At every step, the current status of the system can be represented as a vector containing the probability of each state. The status of next step is computed using a transition matrix. To perform timing analysis, timing distribution vectors – which are used for timing representation and analysis – are assigned to each state. Transient and permanent fault effects are addressed as probabilistic models using fault injection. We employ an online fault detection mechanism for both faults and modify the system state at each step for accounting of faults. Our results show that by our approach, we can obtain tight pWCET estimates. In addition, different scenarios of faults are studied. We can see that permanent faults have a significant impact on performance degradation.

The rest of the paper is organized as follows: related work is discussed in Section II; Section III introduces system model based on Markov chain; the methodology using the model is explained in Section IV; fault models and their impacts on the system are demonstrated in Section V; real-world benchmarks are evaluated in Section VI; and finally Section VII draws come concluding remarks.

II. Related Work

Several works on SPTA have been proposed for caches with random replacement policy. Zhou [7] proposes a cache hit formula using reuse distance – the number of memory addresses accessed between two consecutive references to the same memory address – which simplifies computational complexity significantly. The probabilities for each cache access are made independent, and the final result is the convolution of all cache accesses. However, Cazorla *et al.* [8] and Altmeyer *et al.* [9] have found his methodology unsound. Quinones *et al.* [2] and Kosmidis *et al.* [10] give other formulae for random caches, while Cucu-Grosjean *et al.* [11] and Cazorla *et al.* [8] perform probabilistic timing analysis using these formulae. However, the formulae in [10] may overestimate the cache hit ratio [12].

Davis *et al.* [13] develop a formula using reuse distance only for evict-on-miss caches, and Altmeyer *et al.* [9] prove it to be optimal when only reuse distance is known. Multi-path programs are also analyzed by assuming that they are bounded. Besides, maximum preemption effects during program execution are taken into account for timing analysis. Altmeyer *et al.* [9] propose an exhaustive analysis approach. To reduce its computational complexity, this exhaustive approach can be combined with simplified formulae [14], resulting in an improved algorithm for SPTA. Griffin *et al.* [15] propose a methodology from the field of Lossy Compression and compare it with the method in [9]: by using *May* and *Must* Analysis, the result is more accurate with appropriate parameters. Lesage *et al.* [16] propose an SPTA for multi-math programs by using a conservative approach: cache states upper-bounds are calculated and paths are reduced according to worst-case execution path expansion. To demonstrate the impact of random caches, Abella *et al.* [17], Altmeyer *et al.* [14] and Lesage *et al.* [16] have done comparisons between caches using LRU and random replacement policy.

There are few studies on Probabilistic Timing Analysis (PTA) in the presence of faults. Slijepcevic *et al.* [18] study fault-tolerant systems, and combine it with PTA. *Degraded Test Mode* is proposed for random caches, which specifies requirements for hardware design and test. By using *Degraded Test Mode*, real time systems can be analyzed with probabilities, and the pWCET is performed using MBPTA. Slijepcevic *et al.* extend the work in [19]. They propose an approach taking account of timing impacts of error detection, correction, diagnosis, and reconfiguration (DCDR) and degraded performance due to faults. They verify the timing behavior with different fault scenarios on critical real-time embedded systems and their work is based on MBPTA. Hardy and Puaut [20] present an SPTA-based methodology to calculate pWCET for instruction caches that contains only manufacturing permanent faults with LRU replacement policy. Permanent faults are detected by tests and cache blocks with permanent faults are disabled. A fault-free pWCET and miss probability distribution due to faults are initially computed separately. Then they are combined to form the pWCET with permanent faults. This method does not consider permanent faults that occur during program executions.

Our approach is the first method that calculates the timing behavior of caches with random replacement policy in presence of both transient and permanent faults. We consider permanent faults that happen during execution and are caused by device wear-out effects. This approach is based on SPTA and provides safe and tight pWCET estimates.

III. SYSTEM MODEL

In this section, we present our SPTA model for random caches based on Markov chains. Our model is applied to a fully associative cache and it can be generalized to a set associative cache in which the analysis of each cache set can be performed separately as a fully associative cache. The model uses a memory trace as the input, and obtains a pWCET as the output. It is based on system states, which is similar to other accurate SPTA approaches [9], [15] for random caches. Different heuristics are applied for these approaches, and we apply an adaptive heuristic in the proposed approach.

A Markov chain is a mathematical framework that describes how a system moves from one state to another. If the future state of a system depends uniquely on the current state, such a system forms a Markov chain. The current state describes the status of the system, and the transition matrix explains how the system transits into the next state.

For a cache with evict-on-miss random replacement policy, every time a cache miss happens, a cache block is randomly selected and replaced with the new data (the term *data* is used to refer to the content of a memory address). As a result, there may be different data in the cache at different times, i.e. the memory layout of the cache changes with time. To describe the status of the system, s_i is defined as the memory layout of the cache and $|s_i|$ is the number of elements in this state. Example 3.1 shows how to construct states from a trace of memory access by a task.

Example 3.1: Suppose there is a task τ and a 2-way cache. The memory accesses from τ are a, b, c, a, b. Then we can define the state space as $s_0 = \emptyset$, $s_1 = \{a\}$, $s_2 = \{b\}$, $s_3 = \{c\}$, $s_4 = \{a, b\}$, $s_5 = \{a, c\}$, $s_6 = \{b, c\}$. We can see that all memory layouts are included in the states, and $|s_0| = 0, |s_i| = 1 : i = 1, 2, 3, |s_i| = 2 : i = 4, 5, 6$.

Each program step can be seen as an access to a new memory address. Every time a memory address is accessed, the system advances by 1 time step and the system state may change. Each possible state of the system, i.e. memory layout, at a given step is associated with a probability.

The state occurrence probability vector $\overline{S}$ is defined as:

$$\overline{S} = [Pr(s_0), Pr(s_1), \cdots], \tag{1}$$

where $Pr(s_i)$ is the probability of the state s_i. With m being the number of different memory addresses in the program, and l being minimal value between cache associativity and m, the number of states can be calculated as:

$$\sum_{k=0}^{l} \begin{pmatrix} m \\ k \end{pmatrix}. \tag{2}$$

In addition to $\overline{S}$, we introduce the transition matrix $\overline{P}$, which describes how one state varies from the current step to the next. It is represented as:

$$\overline{P} = \begin{pmatrix} p_{0\to 0}, & p_{0\to 1}, & \cdots \\ p_{1\to 0}, & p_{1\to 1}, & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}, \tag{3}$$

where $p_{i\to j}$ is the probability for the system to go from state s_i to state s_j. In our model, $p_{i\to j}$ varies constantly, because it depends on the current system state and the memory accesses. At each step, the system may access different memory addresses and its state may change. Consequently, the transition probability $p_{i\to j}$ may change and this is a non-homogeneous Markov chain model.

Assuming $\overline{S}_k$ and $\overline{P}_k$ are the state probability vector and the transition matrix at step k, respectively, then we have

$$\overline{S}_{k+1} = \overline{S}_k \overline{P}_k. \tag{4}$$

We can see that the state of the system for next step only depends on current state and the transition matrix.

IV. Methodology

In this section, we demonstrate how to perform SPTA using the proposed system model based on Markov chains.

A. Transition Matrix Calculation

In our Markov chain model, Equation (4) is used to describe the system behavior. Given an initial state, we can calculate the transition matrix at each step and obtain the system state.

ALGORITHM 1: Transition matrix calculation

```
Data: State prob. vector $\overline{S}$, memory address $a$
Result: Transition matrix $\overline{P}$
1  $n \leftarrow |\overline{S}|$; //number of states in $\overline{S}$;
2  for $i \leftarrow 0$ to n-1 do
3      for $j \leftarrow 0$ to n-1 do
4          $p_{i\to j} \leftarrow 0$; //initialize transition matrix
5      end
6      if $Pr(s_i) = 0$ then
7          go to next $i$; //state $s_i$ does not exist
8      end
9      if $s_i = \emptyset$ then
10         $p_{i\to m} \leftarrow 1$; //$s_m = \{a\}$, cache miss for $s_i$
11         go to next $i$;
12     end
13     if $a \in s_i$ then
14         $p_{i\to i} \leftarrow 1$; //cache hit
15         go to next $i$;
16     end
17     $ind0 \leftarrow \emptyset$; //indexes of transitions by replacement
18     $ind1 \leftarrow \emptyset$; //indexes of transitions by new cache block
19     $q \leftarrow |s_i|$; //number of addresses for state $s_i$
20     for $j \leftarrow 0$ to n-1 do
21         $p \leftarrow |s_j|$; //number of addresses for state $s_j$
22         if p-q=0 then
23             $l \leftarrow |s_i - s_j|$; //number of different addresses
24             if l=1 and $a \in s_j$ then
25                 Add $j$ to $ind0$; //replaces existing address
26             end
27         end
28         if p-q=1 then
29             if $s_i \subset s_j$ and $a \in s_j$ then
30                 Add $j$ to $ind1$; //add new address
31             end
32         end
33     end
34     $N \leftarrow$ cache associativity;
35     for $x \in ind0$ do
36         $p_{i\to x} = 1/N$; //replacement prob.
37     end
38     for $x \in ind1$ do
39         $p_{i\to x} = (N-q)/N$; //new address prob.
40     end
41 end
```

Algorithm 1 takes two inputs: the state occurrence probability vector $\overline{S}$ and the incoming memory address, and produces one output: the transition matrix $\overline{P}$. The algorithm checks all states and generates the transition matrix elements accordingly:

- Line 4: All transition probabilities for state s_i are first initialized to 0. They may be modified later depending on current state and incoming address.
- Line 6: $Pr(s_i) = 0$ means that state s_i is an impossible state at the current step. Therefore we have $\forall m, p_{i\to m} = 0$, i.e. one cannot exit from an impossible state.
- Line 9: If s_i corresponds to an empty cache, a cache miss is inevitable, and there is only one possible transition from the empty cache state to the cache state with the incoming memory address.
- Line 13: If the requested memory address is in the cache, there is a cache hit. In this case, the cache will not change its state with probability 1, i.e. $p_{i\to i} = 1$.
- Line 20: If the requested memory address is not in the cache, there is a cache miss and the transition matrix is computed. This is the most complex case: the new memory address may replace an existing cache block, or it may be put into a new cache block and probabilities have to be computed accordingly. In our target cache, the probability of replacing an existing cache block is $1/N$ (see Line 36), where N is the cache associativity. This is because we consider an evict-on-miss random cache, and a cache block is randomly selected for replacement with probability $1/N$. The probability for a memory address to be placed in an empty cache block is $(N-q)/N$ (see Line 39), where q is the number of blocks in use for the current state s_i. This is due to the fact that if the new memory address does not cause a replacement, it can only be put into an empty cache block. The number of empty cache blocks is $N-q$, and they are chosen from N ways. Therefore the probability is $(N-q)/N$.

Example 4.1 shows how to obtain the state at a given step for Example 3.1 using the transition matrix using Equation (4) and Algorithm 1 .

Example 4.1: Let $p_{i\to j} = 0$ at the beginning of each step, and assuming the cache is initially empty at the beginning of step 1, then we have $\overline{S}_1 = [1, 0, 0, 0, 0, 0, 0]$.

At step 1: For $\overline{P}_1$, all elements are 0 except $p_{0\to 1} = 1$. $\overline{S}_2 = \overline{S}_1 \cdot \overline{P}_1 = [0, 1, 0, 0, 0, 0, 0]$.
At step 2: $p_{1\to 2} = 1/2, p_{1\to 4} = 1/2$, $\overline{S}_3 = \overline{S}_2 \cdot \overline{P}_2 = [0, 0, 1/2, 0, 1/2, 0, 0]$.
At step 3: $p_{2\to 3} = 1/2, p_{2\to 6} = 1/2, p_{4\to 5} = 1/2, p_{4\to 6} = 1/2$, $\overline{S}_4 = \overline{S}_3 \cdot \overline{P}_3 = [0, 0, 0, 1/4, 0, 1/4, 1/2]$.
At step 4: $p_{3\to 1} = 1/2, p_{3\to 5} = 1/2, p_{5\to 5} = 1, p_{6\to 4} = 1/2, p_{6\to 5} = 1/2$, $\overline{S}_5 = \overline{S}_4 \cdot \overline{P}_4 = [0, 1/8, 0, 0, 1/4, 5/8, 0]$.
At step 5: $p_{1\to 2} = 1/2, p_{1\to 4} = 1/2, p_{4\to 4} = 1, p_{5\to 4} = 1/2, p_{5\to 6} = 1/2$, $\overline{S}_6 = \overline{S}_5 \cdot \overline{P}_5 = [0, 0, 1/16, 0, 5/8, 0, 5/16]$.

B. Timing Analysis

With Algorithm 1, Equation (4) can be used to describe the system state transitions. Nonetheless, the duration of a task execution is different from the step used in the Markov chain. At each step, one memory address is accessed and different number of cycles may be applied to the timing analysis according to the system state. Without loss of generality, we assume 1 cycle for a cache hit and 100 cycles for a cache miss (any timing behavior would work). With different number of cycles executing the program and their corresponding occurrence probabilities, we have timing distributions for programs. The resulting timing distributions are discrete-time distributions as each memory access takes $n \in \mathbb{N}$ number of cycles.

Two vectors are introduced for defining timing distributions and modeling timing behaviors with respect to probabilities. A cycle vector $\overline{C}$ can be used to denote the timing distribution in terms of number of cycles, and a probability vector $\overline{M}$ can represent the probability of occurrence for $\overline{C}$. Then we have $\overline{C} = [c_0, c_1, \cdots]$ and $\overline{M} = [m_0, m_1, \cdots]$, where $c_i \in \mathbb{N}$ represents the program duration in cycles and $m_i = Pr(c_i), m_i \in \mathbb{R}$ denotes the occurrence probability for c_i. A scalar addition of $\overline{C}$ and a scalar multiplication of $\overline{M}$ are defined as $\overline{C} + n = \{c + n | n \in \mathbb{N}, c \in \overline{C}\}$,
$\overline{M} \cdot p = \{m \times p | p \in \mathbb{R}, m \in \overline{M}\}$.

With the cycle vector $\overline{C}$ and its probability vector $\overline{M}$, we define the timing distribution for state s_i as $\mathcal{T}_i =< \overline{C}_i, \overline{M}_i >$. $\mathcal{T}_i$ collects the number of cycles to execute the program and corresponding probabilities for each cycle. The timing distribution changes during program execution; the initial values is $\mathcal{T}_i =< [0], [1] >$, and each memory access adds additional cycles and probabilities to the timing distribution.

Timing distributions $\mathcal{T}$ need to be combined during state transitions, since different states can transit to the same state after the memory access. Therefore the merge operation between timing distributions $\biguplus$ is defined such that the resulting distribution $\mathcal{T}_k$ is

$$\mathcal{T}_k = \mathcal{T}_i \biguplus \mathcal{T}_j =< \overline{C}_k, \overline{M}_k >, \tag{5}$$

where $\overline{C}_k = \overline{C}_i \cup \overline{C}_j$, and $\overline{M}_k = \{m_p + m_q | m_p \in \overline{M}_i, m_q \in \overline{M}_j, c_p \in \overline{C}_i, c_q \in \overline{C}_j, c_p = c_q\}$. The merge operation puts all number of cycles into one vector, and the probabilities with the same number of cycle are added together.

With the transition matrix $\overline{P}$, the timing distribution for state s_j is:

$$\mathcal{T}_j = \begin{cases} < \emptyset, \emptyset > & if \quad \forall i, p_{i \to j} = 0 \\ \biguplus_i < \overline{C}_i + n_a, \overline{M}_i \cdot p_{i \to j} > & otherwise, \end{cases} \tag{6}$$

where

$$n_a = \begin{cases} n_h & if \quad j = i \\ n_m & if \quad j \neq i, \end{cases} \tag{7}$$

n_h is the number of cycles for a cache hit, and n_m is the number of cycles for a cache miss, e.g. the previously chosen values of 1 and the 100.

By merging timing distribution vectors of all states, using Equation (6), we could compute the timing distribution of the whole program as $\mathcal{T} =< \overline{C}, \overline{M} >$.

Having the timing distribution, for each state and for the whole program, we can compute both the Cumulative Distribution Function and the inverse Cumulative Distribution Function (1-CDF) [13]. The inverse cumulative is an exceedance function showing the probability of exceeding a certain program duration in cycles. The 1-CDF probabilities are denoted as $\overline{Q} = [q_0, q_1, \cdots]$, with

$$q_i = \sum_{m_j \in \overline{M}, c_i, c_j \in \overline{C}, c_j > c_i} m_j. \tag{8}$$

The inverse timing distribution is defined as $\mathcal{I} =< \overline{C}, \overline{Q} >$ and can be related to the state s_i, i.e. $\mathcal{I}_i$, or the whole program, i.e. $\mathcal{I}$.

Example 4.2: In this example we demonstrate how to do timing analysis for the parameters listed in Example 3.1.

At each step, we use Equation (6) to compute $\mathcal{T}_i$. Let $n_h = 1$, $n_m = 100$

At step 1: $\mathcal{T}_1 =< [100], [1] >$.
At step 2: $\mathcal{T}_2 =< [200], [1/2] >$, $\mathcal{T}_4 =< [200], [1/2] >$.
At step 3: $\mathcal{T}_3 =< [300], [1/4] >$, $\mathcal{T}_5 =< [300], [1/4] >$, $\mathcal{T}_6 =< [300], [1/2]$.
At step 4: $\mathcal{T}_1 =< [400], [1/8] >$, $\mathcal{T}_4 =< [400], [1/4]$, $\mathcal{T}_5 =< [301, 400], [1/4, 3/8] >$.
At step 5: $\mathcal{T}_2 =< [500], [1/16] >$, $\mathcal{T}_4 =< [401, 500], [3/8, 1/4]$, $\mathcal{T}_5 =< [401, 500], [1/8, 3/16] >$.

By Equation (6), we have $\mathcal{T} =< [401, 500], [1/2, 1/2] >$.

From Equation (8), the inverse timing distribution is $\mathcal{I} =< [401, 500], [1/2, 0] >$, i.e. there is the probability of 1/2 to exceed 401 cycles, and the probability to exceed 500 cycles is 0, since the maximum execution time is 500 cycles.

It is worth noting that for a set associative cache, the Markov chain model applies to each cache set CS_k. As a result, there are both the timing distribution $\mathcal{T}_{CS_k}$ and the inverse timing distribution $\mathcal{I}_{CS_k}$ specific of the cache set. Assuming a deterministic placement policy (e.g. modulo placement) is applied, let a_m^k be an address assigned to cache set CS_k. This address can assigned to only one cache set. The timing distribution $\mathcal{T}_{CS_k}$ is a function of the addresses assigned to it, i.e. $\mathcal{T}_{CS_k} = f(a_0^k, a_1^k, ...)$. For another cache set timing distribution, we have $\mathcal{T}_{CS_l} = f(a_0^l, a_1^l, ...)$ and $a_m^k \neq a_n^l : k \neq l$. We can see that cache set timing distributions are functions of different addresses and are thus statistically independent of each other, i.e. $\mathcal{T}_{CS_k} \perp \mathcal{T}_{CS_l} : k \neq l$.

To obtain the timing distribution $\mathcal{T}$ of different cache sets, we apply the convolution operator $\otimes$ between different cache set timing distributions:

$$\mathcal{T}_{CS} = \mathcal{T}_{CS_k} \otimes \mathcal{T}_{CS_l} =< \overline{C}, \overline{M} >,$$

where $\overline{C} = \{c_p + c_q | c_p \in C_{CS_k}, c_q \in C_{CS_l}\}$, and $m_i \in \overline{M}$ calculated as $m_i = \sum_{m_p \in \overline{M}_{CS_k}, m_q \in \overline{M}_{CS_l}, c_p + c_q = c_i} m_p m_q$.

C. Adaptive Method

The result of our Markov model is an accurate timing analysis, because it takes all states into account and computes how they change over time. The Markov model overcomes the pessimism introduced by formulae in [13]. The resulting timing distribution $\mathcal{T}$ is the pWCET obtained accounting for all the cache configurations while the program executes.

However, from Equation (2) we can see that the number of states increases polynomially with a high exponent value as more memory addresses are accessed. The method proposed could become intractable. We use then an adaptive method to limit the number of states and to produce a result with reasonable accuracy, which scales with the size of memory accesses.

Suppose there are n different memory addresses, in order to reduce computational complexity, we would like to use only m ($m < n$) memory addresses for the states so that the number of states is limited and we can enumerate all states. This is realized with two parts: the i) *state modification* and the ii) *state and timing distribution merge*.

i) *State modification*: for the first m different addresses $a_0, a_1, ...a_{m-1}$, where $a_i \neq a_j$ for $i \neq j$. We construct the state space $\{s_0, s_1, ...\}$ using the proposed Markov chain method. We have $\forall A \subseteq \{a_0, a_1, ...a_{m-1}\}$, $\exists i : A \subseteq s_i$. The number of states is from Equation (2). When another new memory address a_m comes, we modify states in the state space, instead of increasing the number of states.

In order to modify states, we find a memory address $a \in \{a_0, a_1, ...a_{m-1}\}$. The state s_i containing a is changed to state s_j in which a_m replaces a. There are different heuristics to select a. We assume that a least recently used address a has a low probability to be used again in recent memory accesses, and a is to be replaced as follows: $\forall i : a \in s_i, s_j = s_i \setminus \{a\} \cup \{a_m\}$. The probability and the timing distribution for s_j are respectively $Pr(s_j) = 0$ and $\mathcal{T}_j =< \emptyset, \emptyset >$. This way, the number of states remains the same, but different addresses can be used in the state space. The method works in an adaptive way by modifying the states with the least recently used addresses. Note that the least recently used address is used to change states, and it is not a cache replacement policy.

ii) *State and timing distribution fusion*: When states are changed, we need to take timing analysis into account as well, because each state is assigned different timing distributions. To obtain the safe bound to the pWCET, we use a conservative method dealing with the $\overline{S}$ and $\mathcal{T}$ variables for the timing analysis. $\overline{S}$ is the state occurrence vector and $\mathcal{T}$ is the timing distribution.

Suppose $s_i : a \in s_i$ is the state before state modification, and s_p is the state containing all memory addresses in s_i except the address a that is to be replaced, i.e. $s_p = s_i \setminus \{a\}$. In state modification, we have seen that whenever a new address is accessed, we may change the state s_i. Therefore the state vector which represents its occurrence probability must be modified accordingly. In the new state vector $\overline{S}$, we use

$$Pr(s_p) = Pr(s_i) + Pr(s_p) \tag{9}$$

The occurrence probability $Pr(s_p)$ can be accumulated to account for the occurrence probability $Pr(s_i)$, because if $s_p \subseteq s_i$, for any new memory address, state s_i has the same or a higher cache hit probability compared to that from state s_p. Equation (9) adds pessimism to timing analysis, but it provides a safety bound. In addition to state modification, we need to merge the timing distribution $\mathcal{T}_i$ to $\mathcal{T}_p$ using Equation (5).

After the *state modification* and *state and timing distribution fusion*, the Markov chain model uses the same methodology developed in Section IV for new memory accesses. By using the state space constructed by m addresses, the adaptive method is tractable. To trade off for tractability, the accuracy is compromised, because only some of addresses are used to build the state space. Therefore timing behaviors from the addresses that are discarded are not considered. Nevertheless, the pWCET estimate from this method is safe and it becomes tighter as more addresses are applied.

V. Fault Impacts

In this section, both transient and permanent fault models are introduced to the system that is equipped with an online fault detection mechanism. We consider faults that only occur in the storage elements of the cache and apply probabilistic models for faults. Faults in combinational circuits are not considered in this paper. We use the fault occurrence probability of each memory access step for analysis. Since a cache miss takes longer than a hit, to simplify the analysis and obtain a safe bound, we assume that each memory access step is a cache miss, i.e. it takes n_m cycles and this value is used for fault rate calculation in following sections.

For set associative caches, different cache sets may be accessed. Let n_i and n_{i+1} be consecutive steps to access the same cache set, and n_s be the step difference. We have

$$n_s = n_{i+1} - n_i. \tag{10}$$

For fully associative caches, $n_s = 1$. We assume a constant fault rate f is applied to both transient and permanent faults. The probability to have a fault for one cache block is

$$1 - (1 - f)^{n_s}, \tag{11}$$

and the probability without a fault is $(1 - f)^{n_s}$.

A. Transient Fault Impact

A Single Event Upset (SEU) is a change of state caused by a high-energy particle. We regard an SEU as a transient fault, since it does not cause permanent damage and the system can be recovered. SEUs are known to be independent, and we assume they are uniformly distributed in space and time, i.e. each cache block has the same fault rate throughout the program execution. After the fault occurs, the cache block remains faulty unless this fault is detected. This is because the transient fault happens to the storage element. Once the storage state changes, it can not recover automatically. As a result, transient fault effect lasts. After the fault is detected, this block is seen as invalid, but it does not affect following data storage in it.

To deal with transient fault, many techniques have been proposed. For example, Reed-Solomon codes have been used extensively for space applications to detect and correct transient fault errors. In this paper, we employ a simple parity check fault detection mechanism – where parity bits are are added to the data – to first level (L1) cache, which has less area and speed penalties for L1 caches compared to commonly used single error correction-double error detection (SEC-DED) techniques [21].

When a cache block is accessed, the parity bits stored are examined to see if any transient fault has occurred. Due to low probability of fault, we assume that all faults can be detected. If any fault is detected, the corresponding data is regarded as invalid and will be fetched from the main memory again.

We note that with parity check mechanism, the impact of a transient fault is equivalent to a cache eviction, i.e. once the transient fault occurs, the data is not valid any more and it is a cache miss. Let f_t be the transient fault probability at each step and s_i be the state before transient fault detection. After fault detection, s_i becomes s_p. With Equation (10) and (11), we have $s_p \subseteq s_i$ and

$$Pr(s_p) = Pr(s_i)((1-f_t)^{n_s})^{|s_p|}(1-(1-f_t)^{n_s})^{|s_i|-|s_p|}. \quad (12)$$

Transient fault detection produces new states, which may be the same as existing ones. If $\exists s_m : s_m = s_p$, we change timing distributions by multiplying the state change probability, and then merge state probabilities and timing distributions using Equation (9) and Equation (5). Example 5.1 demonstrates how a state changes because of transient fault after one step.

Example 5.1: Suppose we have a state $s = \{a, b\}$, $Pr(s) = 0.5$ and the transient fault probability is $f_t = 0.1$ at each step. After one step, from Equation (12) we know that new states are produced and we have

$$\begin{aligned} Pr(s = \{a,b\}) &= 0.5 \times (1-0.1)^2 = 0.405 \\ Pr(s = \{a\}) &= 0.5 \times 0.1 \times (1-0.1) = 0.045 \\ Pr(s = \{b\}) &= 0.5 \times 0.1 \times (1-0.1) = 0.045 \\ Pr(s = \emptyset) &= 0.5 \times 0.1^2 = 0.005 \end{aligned}$$

B. Permanent Fault Impact

1) Permanent Fault Model: Permanent faults are faults whose effects are assumed to last from the moment they appear to the end of the program execution. When a permanent fault occurs to a cache block, it cannot be used any more.

To model permanent faults we start by defining the probability $f_p(t, T)$ of a permanent fault occurring. It is the probability of fault in a system component by time t, $failure \leq t$, given that the component was still functional at the end of the previous interval $t - T$, $failure > t - T$. T is the scrubbing period, i.e. the time interval between two consecutive fault detection to avoid error accumulation. In this paper it is the time for one memory access. This probability can be computed using the Kolmogorov definition from the formula in [22], as:

$$\begin{aligned} f_p(t,T) &= Pr(failure \leq t | failure > t - T) \\ &= \frac{Pr(failure \leq t \wedge failure > t - T)}{Pr(failure > t - T)} \\ &= \frac{\mathsf{cdf}_{failure}(t) - \mathsf{cdf}_{failure}(t-T)}{1 - \mathsf{cdf}_{failure}(t-T)}, \quad (13) \end{aligned}$$

with $\mathsf{cdf}_{failure}$ the cumulative density function of the random variable failure describing the time at which the failure happens.

In literature, several probability distributions are used to model failure times [22]. One of the most frequently used is the exponential distribution; however, the exponential distribution representation is somewhat imprecise because it lacks the ability to capture the increasing failure probability due to accumulated wear in the component. A common alternative used to overcome this limitation is a log-normal failure distribution:

$$f_p(t,T) = \frac{\mathsf{cdf}_{norm}\frac{ln(t)-\mu}{\sigma} - \mathsf{cdf}_{norm}(\frac{ln(t-T)-\mu}{\sigma})}{1 - \mathsf{cdf}_{norm}(\frac{ln(t-T)-\mu}{\sigma})} (14)$$

where cdf_{norm} the cumulative density function of the normal distribution. The mean and standard deviation parameters of such distribution can be computed from the Mean Time To Failure (MTTF) such that

$$\begin{aligned} \mu &= ln(\frac{MTTF^2}{\sqrt{var_{MTTF} + MTTF^2}}) \\ \sigma &= \sqrt{ln(1 + \frac{var_{MTTF}}{MTTF^2})}. \quad (15) \end{aligned}$$

Note that in Equation 14, $f_p(t, T)$ depends on the actual time t and the scrubbing period T. The non-memoryless distribution function describe the occurrence of a recent failure with larger probability than a memory-less distribution like the exponential one.

Figure 1a summarizes the comparison of the log-normally distributed failure times with different MTTFs. The plot is discretized in years. We can see that as MTTF increases, the permanent fault rate f_p for each memory access decreases, because a smaller fault rate can lead to a longer lifetime. In addition, f_p is an increasing function of time. As system operation time increases, f_p increases continuously. In this paper, however, we assume that f_p is constant, because execution times of our benchmarks are short and f_p rises extremely slowly.

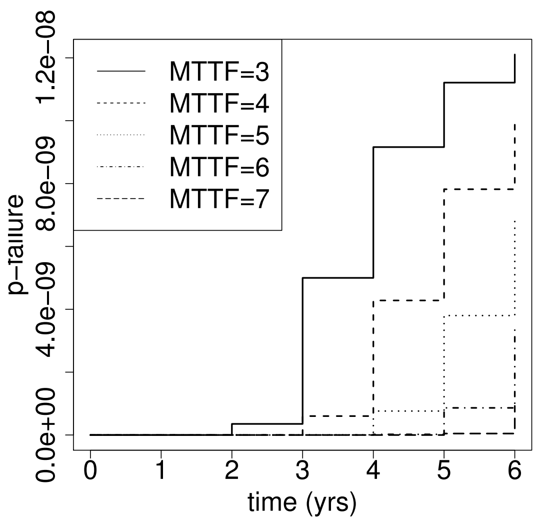
Figure 1b depicts the comparison of the log-normally distributed failure times with different operating frequencies. The desired MTTF is arbitrarily set at 5 years. At the beginning, the fault rate is extremely low, e.g. with KHz frequency, the permanent rate is 4.4×10^{-18} at year 2. The MHz and GHz fault rates are 10^{-3} and 10^{-6} smaller respectively.

2) Permanent Fault Detection: To deal with permanent fault in the SPTA, we establish different Markov chain models with different numbers of faults. For N-way set associative caches, we implement $N + 1$ Markov chain models, where the ith model contains $i - 1$ permanent faults and there are $N - (i - 1)$ available cache blocks. The model with N faults is the worst case where all cache blocks are faulty. Its timing analysis is easy to calculate since there are always cache misses.

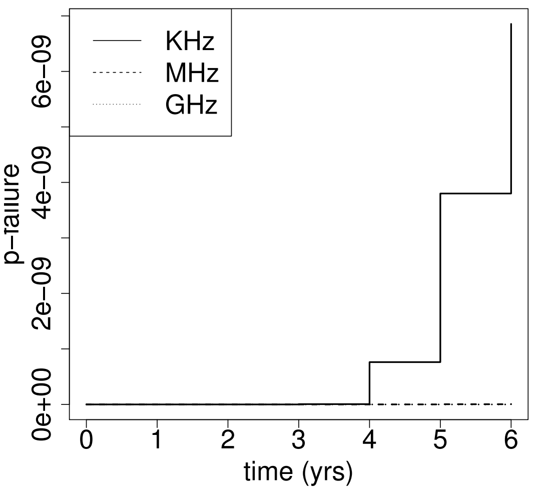
When the memory is accessed, fault detection is applied using parity check. To identify if the fault is permanent, we adopt the method proposed by [23]. It is simple to implement and tracks the number of fault occurrences. If a threshold value is exceeded, the fault is classified as permanent. To improve following timing behaviors, we assume that each cache block can be controlled separately. Once a cache block is classified to have a permanent fault, it will be disabled and will not be used any more.

Figure 2 shows how to apply different Markov chain models to N-way caches, with each node representing a Markov chain model state space. There are $N + 1$ rows for N-way caches, i.e. $N + 1$ Markov chain models. Memory addresses are accessed at step $1, 2, 3, ...$ The analysis is completed in two phases to account for fault events.

Phase 1: Fault detection. This is denoted by dotted lines. Node $\overline{S}_n^m$ denotes the state occurrence probability vector of the system with m faults at step n. Let f_p be the probability of permanent fault at each step. With Equation (10) and (11), for each node, the state $s_i^m \in \overline{S}_n^m$ is changed as follows.



(a) Varying the MTTF. Increasing the MTTF the f_p in years (yrs) decreases consistently. f_p computed from a KHz frequency



(b) Varying the platform frequency. The f_p in years (yrs) for MHz and GHz are respectively 10^{-3} and 10^{-6} smaller than those for KHz

Fig. 1: Failure probability f_p the MTTF and the platform frequency

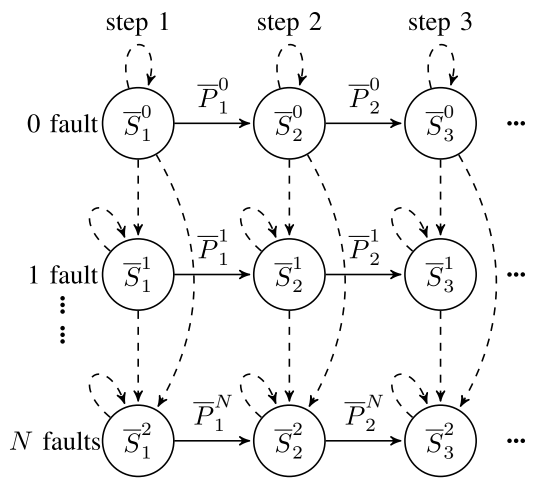


Fig. 2: Different Markov chain models taking account of permanent faults for N-way caches. Dotted lines indicate state changes due to fault detection, and solid lines denote state transitions due to memory accesses.

- No permanent faults occur.

$$Pr(s_i^m) = Pr(s_i^m)((1 - f_p)^{n_s})^{N-m}. \tag{16}$$

The probability of this state changes due to potential permanent fault occurrences, and timing distribution are changed by multiplying the state change probability.

- Permanent faults occur. Let l be the number of added permanent faults on current model and $s_p^{m+l} \in \overline{S}_n^{m+l}$ be the state after permanent fault detection. We assume that permanent faults can be detected immediately after they appear. Then we have $s_p^{m+l} \subseteq s_i^m$ and

$$Pr(s_p^{m+l}) = Pr(s_i^m)((1-f_p)^{n_s})^{N-m-l}(1-(1-f_p)^{n_s})^l. \tag{17}$$

For state $s_m^{m+l} : s_m^{m+l} = s_p^{m+l}$, we change timing distributions by multiplying the state change probability, and then merge state probabilities and timing distributions using Equation (9) and Equation (5).

Phase 2: State transition. This is denoted by solid lines. After the fault detection, states in different Markov chain models are updated, since new addresses are accessed. Together with the transition matrix $\overline{P}_n^m$ (the transition matrix with m faults at step n), the timing analysis methodology from Section IV is applied to each model.

VI. Experimental Results

A. Experimental Setup

In our experiments, we used the SoCLib open platform[1] to generate memory traces for benchmarks. The platform is equipped with one MIPS 32-bit processor with L1 instruction cache. In order to evaluate our approach, we adopt Mälardalen benchmarks [24], a popular benchmark suite used for WCET evaluation and analysis. Due to limited space, we only present results of two benchmarks ($fdct$ and crc). By injecting faults into instruction cache with different fault rates, we investigate their impacts on the system. The cache size is 512 bytes, with 2-way associativity and 4-byte cache block. We assume that for each cache miss, the duration is 100 cycles; for each cache hit, the duration is 1 cycle. To account for fault detection delays, we add additional 10 cycles. A cache with bigger size can also be adopted, in which more cache sets are used and thus there are fewer addresses for each set. The timing distribution calculations for each set perform faster due to address reductions.

[1]http://www.soclib.fr

In our adaptive Markov chain model, we adopt 4 memory addresses for adaptive state modification. Since execution times of benchmarks are short, we create synthetic benchmarks which repeat the same benchmark 10 times. This way, we can study as execution time increases, how the system is affected by the faults and we can see if repeated benchmarks produce similar pWCET estimates to the original benchmarks. We perform 1,000 simulations for each benchmark as the base line and this can be used to verify the accuracy of the method at around exceedance probability of 10^{-3}. A brief comparison can be done using such an exceedance probability between simulations and our approach. If a lower exceedance probability is required, more simulations can be performed. [2]

Figure 3 – Figure 6 show timing analyses of benchmarks $fdct$ and crc respectively. $fdct$ is fast discrete cosine transform using a lot of calculations based on integer arrays and crc is cyclic redundancy check computation using complex loops with lots of decision. On each figure, the x-axis shows the number of cycles and y-axis represents the exceedance probability (i.e. 1-CDF) for corresponding cycles. The exceedance probability is set as 10^{-15}, for the failure rate requirement at the highest level for commercial airborne is translated into the region of around 10^{-13}. To show simulation results in detail, a zoomed figure of each benchmark which limits the exceedance probability to 10^{-3} is displayed. Different fault scenarios are applied. The transient fault rate that we applied is at each step, there is a probability of 10^{-20} for fault occurrence. Different permanent fault rates are applied to show account for wear-out effects at different times. Since for a MHz with 5-year MTTF, the permanent fault probability is around 10^{-20}, we applied permanent fault probabilities of $10^{-20}, 10^{-15}, 10^{-10}, 10^{-7}, 10^{-6}, 10^{-5}$ respectively to study how the system is affected when permanent fault rate increases.

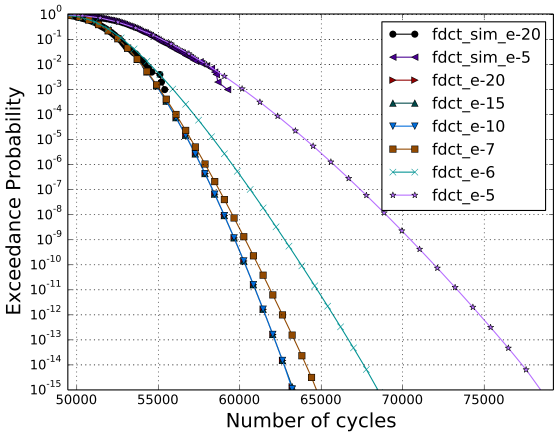
B. Discussion

To verify the accuracy of our SPTA approach, simulations are performed with transient fault rate at 10^{-20} per memory access, and two permanent fault rates at 10^{-20} and 10^{-5} per memory access are applied. From zoomed figures (Figure 3b, 5b, 6b), we can see that for all fault scenarios, at any exceedance probability the simulation execution time matches our result, which means that our approach provides tight pWCET estimates. Note that simulations results may show an error at low probabilities as the number of datapoints are insufficient to accurately estimate the exceedance function.

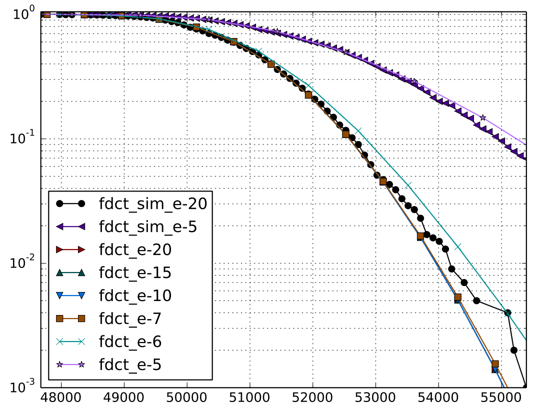
In Figure 4, we can see that there is a slight difference between simulations and the result from our approach, because our adaptive method uses only some of states for analysis. As a consequence, the accuracy may be compromised. Although the synthetic benchmark is a repetition of an original benchmark, its timing behavior may be different. At an exceedance probability, the number of cycles for the synthetic benchmark is not the original value multiplied by the number of repetitions, because after the first completion of the benchmark, some code may exist in the cache, which can help reduce execution time for following executions.

For transient fault, its fault rate is extremely low. In addition, when the cache block with transient fault is accessed, the transient fault will be detected if it has occurred. The new data to be put into this cache block will not be affected, since transient fault does not accumulate. As a result, the impact of transient fault is not significant.

[2]The replication package of our method script is available on demand.



(a) original fdct

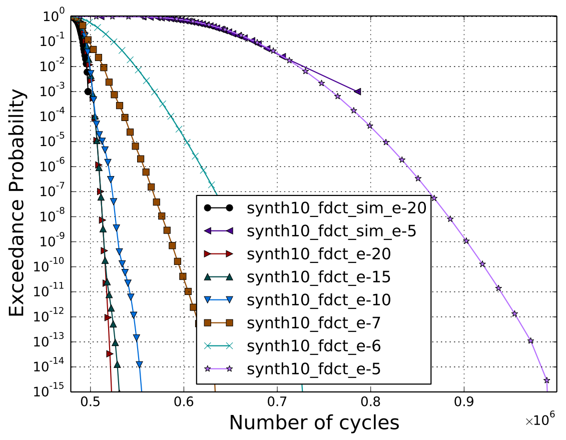


(b) zoomed original fdct

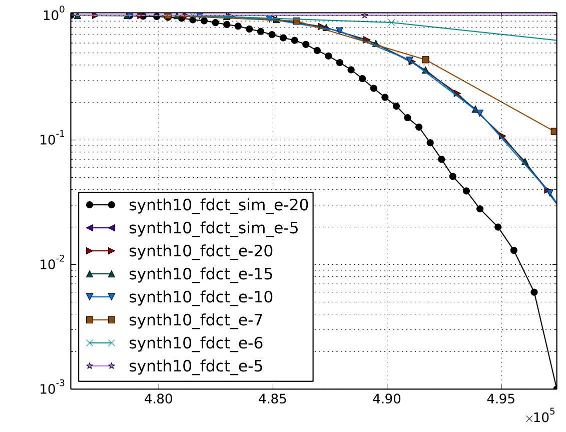
Fig. 3: fdct and the zoomed figure

The fact that permanent faults can accumulate has a significant impact on the behavior of the system, especially since device aging can increase their rate. In our experiments, we have applied different permanent fault rates to the system. We can see that when the permanent fault rate is extremely low, the system is not affected during benchmark execution. However, as fault rate increases, the system takes more time to finish the benchmark.

The size of benchmarks has different impacts on transient and permanent faults. For transient faults, since they can be recovered, the benchmark size does not have a big influence. However, since permanent faults accumulate, as benchmark size increases, the execution times may become longer. For example, at the exceedance probability of 10^{-15}, for original $fdct$ with permanent fault rate of 10^{-5}, it takes around 80,000 cycles. Its synthetic benchmark takes around 1000,000 cycles. Even if it may contain some existing code for repetitions, the synthetic benchmark takes more than 10 times in terms of

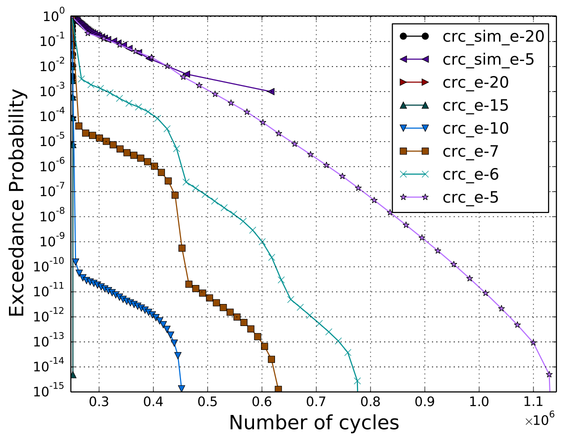


(a) synthetic (repeated) fdct

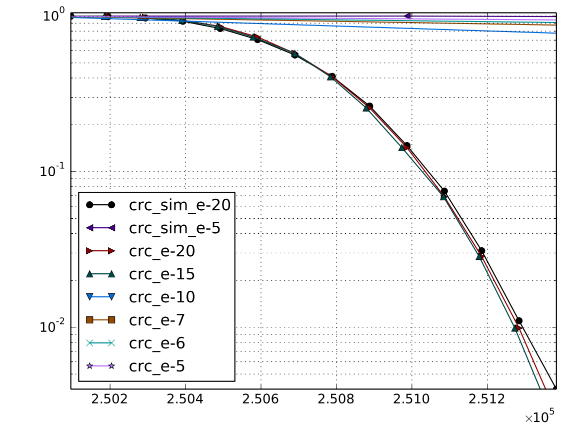


(b) zoomed synthetic (repeated) fdct

Fig. 4: synthetic fdct and the zoomed figure



(a) original crc



(b) zoomed crc

Fig. 5: crc and the zoomed figure

cycles due to permanent fault effects.

We note that for some permanent fault rates, there may be a sudden drop in exceedance probability, especially in Figure 5, where the exceedance decrease drops at similar execution time. This is because permanent faults may have significant impacts on some cache sets depending on benchmark characteristics. In Figure 7, two cache set exceedance probabilities are convolved, where x-axis is execution time and y-axis indicates exceedance probability. It shows that one cache set exceedance probability changes gradually while the other one is affected badly by permanent faults. As a result, the convolution result may have drastic drop around some execution time.

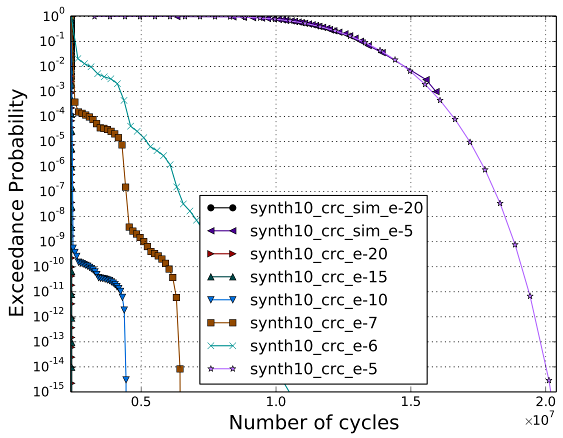
From the experiments, we can see that our approach can work with different fault rates. Depending on characteristics of benchmarks, their pWCET estimates are affected in different ways. For example, the $fdct$ benchmark pWCET varies gradually as higher-level permanent fault rates are applied, while crc benchmark exhibits pWCET by a dramatic change at some exceedance probabilities. One potential use of our approach is to estimate fault impacts on program timing behaviors with random replacement caches, so that we can make sure that safety requirement is met under different fault scenarios.

VII. Conclusion

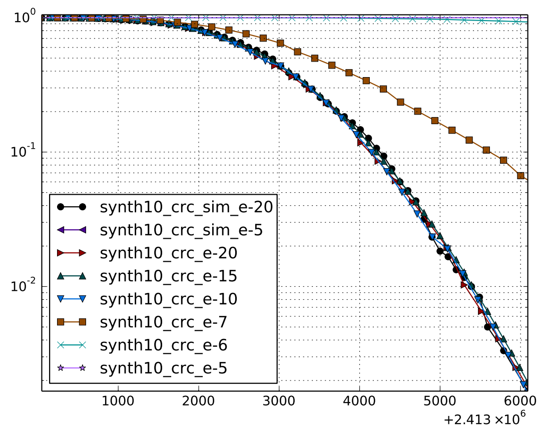
In this paper, we have demonstrated an adaptive Markov chain based Static Probabilistic Timing Analysis (SPTA) methodology in presence of faults; our methodology is based on a non-homogeneous Markov chain model. Both transient and permanent faults are then introduced into the system. The states are modified accordingly for including fault impacts and the pWCET obtained embeds faults effects. The experimental results show how faults affect execution time.

In order to reduce computational complexity, the state space can be limited to the specified level. The state space is modified in an adaptive way, such that existing addresses can be replaced by new incoming addresses in the state space. This guarantees good accuracy and scalability of our SPTA analysis.

As future work, we intend to address aspects such as benchmark and platform fault tolerance. Furthermore, we will enhance our SPTA Markov chain methodology to apply preemptions and multi-processor embedded system platforms.



(a) synthetic (repeated) crc



(b) zoomed synthetic (repeated) crc

Fig. 6: synthetic crc and the zoomed figure

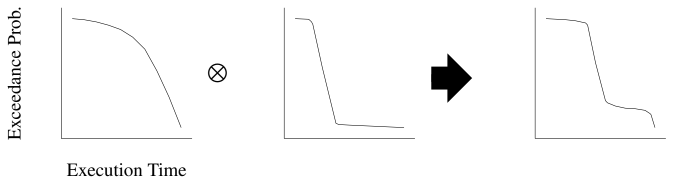


Fig. 7: Convolution of two different exceedance probabilities. One exceedance probability decreases gradually, and the other decreases dramatically due to fault impacts.

Acknowledgment

The authors would like to thank Regroupement Stratégique en Microsystèmes du Québec (ReSMiQ) for their support.

References

[1] G. Bernat, A. Colin, and S. Petters, "Wcet analysis of probabilistic hard real-time systems," in *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, 2002, pp. 279–288.

[2] E. Quinones, E. Berger, G. Bernat, and F. Cazorla, "Using randomized caches in probabilistic real-time systems," in *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, July 2009, pp. 129–138.

[3] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, Jul. 2003.

[4] E. Normand, "Single-event effects in avionics," *Nuclear Science, IEEE Transactions on*, vol. 43, no. 2, pp. 461–474, Apr 1996.

[5] S. Guertin and M. White, "Cmos reliability challenges the future of commercial digital electronics and nasa," in *NEPP Electronic Technology Workshop*, 2010.

[6] R. Serfozo, *Basics of applied stochastic processes*. Springer, 2009.

[7] S. Zhou, "An efficient simulation algorithm for cache of random replacement policy," in *Network and Parallel Computing*. Springer, 2010, pp. 144–154.

[8] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "Proartis: Probabilistically analyzable real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 94:1–94:26, May 2013.

[9] S. Altmeyer and R. Davis, "On the correctness, optimality and precision of static probabilistic timing analysis," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, pp. 1–6.

[10] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "A cache design for probabilistically analysable real-time systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 513–518.

[11] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, July 2012, pp. 91–101.

[12] R. Davis, "Improvements to static probabilistic timing analysis for systems with random cache replacement policies," *RTSOPS 2013*, pp. 22–24, 2013.

[13] R. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean, "Analysis of probabilistic cache related pre-emption delays," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013, pp. 168–179.

[14] S. Altmeyer, L. Cucu-Grosjean, and R. Davis, "Static probabilistic timing analysis for real-time systems using random replacement caches," *Real-Time Systems*, vol. 51, no. 1, pp. 77–123, 2015.

[15] D. Griffin, B. Lesage, A. Burns, and R. I. Davis, "Static probabilistic timing analysis of random replacement caches using lossy compression," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014, pp. 289:289–289:298.

[16] B. Lesage, D. Griffin, S. Altmeyer, and R. Davis, "Static probabilistic timing analysis for multi-path programs," in *Real-Time Systems Symposium, 2015 IEEE*, Dec 2015, pp. 361–372.

[17] J. Abella, D. Hardy, I. Puaut, E. Quinones, and F. Cazorla, "On the comparison of deterministic and probabilistic wcet estimation techniques," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 266–275.

[18] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quinones, and F. Cazorla, "Dtm: Degraded test mode for fault-aware probabilistic timing analysis," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, July 2013, pp. 237–248.

[19] ——, "Timing verification of fault-tolerant chips for safety-critical applications in harsh environments," *Micro, IEEE*, vol. 34, no. 6, pp. 8–19, Nov 2014.

[20] D. Hardy and I. Puaut, "Static probabilistic worst case execution time estimation for architectures with faulty instruction caches," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, ser. RTNS '13. New York, NY, USA: ACM, 2013, pp. 35–44.

[21] K. Mohr and L. Clark, "Delay and area efficient first-level cache soft error detection and correction," in *Computer Design, 2006. ICCD 2006. International Conference on*, Oct 2006, pp. 88–92.

[22] J. Panerati, S. Abdi, and G. Beltrame, "Balancing system availability and lifetime with dynamic hidden markov models," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, July 2014, pp. 240–247.

[23] J. Abella, P. Chaparro, X. Vera, J. Carretero, and A. Gonzalez, "On-line failure detection and confinement in caches," in *On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International*, July 2008, pp. 3–9.

[24] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 15, 2010, pp. 136–146.