Static probabilistic Worst Case Execution Time Estimation for architectures with Faulty Instruction Caches

Damien Hardy, Isabelle Puaut, University of Rennes 1/IRISA, France
{damien.hardy,isabelle.puaut}@irisa.fr

ABSTRACT

Semiconductor technology evolution suggests that permanent failure rates will increase dramatically with scaling, in particular for SRAM cells. While well known approaches such as error correcting codes exist to recover from failures and provide fault-free chips, they will not be affordable anymore in the future due to their non-scalable cost. Consequently, other approaches like fine grain disabling and reconfiguration of hardware elements (e.g. individual functional units or cache blocks) will become economically necessary. This fine-grain disabling will lead to degraded performance compared to a fault-free execution.

To the best of our knowledge, all static worst-case execution time (WCET) estimation methods assume fault-free architectures. Their result is not safe anymore when using fine grain disabling of hardware components, which degrades performance. In this paper we provide the first method that statically calculates a probabilistic WCET bound in the presence of permanent faults in instruction caches. The proposed method, from a given program, cache configuration and probability of cell failure, derives a probabilistic WCET bound. The proposed method, because it relies on static analysis, is guaranteed to identify the longest program path, its probabilistic nature only stemming from the presence of faults. The method is computationally tractable because it does not require an exhaustive enumeration of the possible locations of faulty cache blocks. Experimental results show that it provides WCET estimates very close to, but never below, the method that derives probabilistic WCETs by enumerating all possible locations of faulty cache blocks. The proposed method not only allows to quantify the impact of permanent faults on WCET estimates, but also can be used in architectural exploration frameworks to select the most appropriate fault management mechanisms.

1. INTRODUCTION

In safety-critical real-time systems it is crucial to prove that tasks meet their deadline in all execution situations,

RTNS 2013, October 16 - 18 2013, Sophia Antipolis, France
Copyright 2013 ACM 978-1-4503-2058-0/13/10 ...$15.00.
http://dx.doi.org/10.1145/2516821.2516842.

including the worst-case. This proof needs an estimation of the worst-case execution time (WCET) of any sequential task in the system. WCET estimation methods must offer both *safety* and *tightness* properties. *Safety* is the guarantee that the estimated WCET is greater than or equal to any possible execution time. *Tightness* means that the estimated WCET is as close as possible to the actual WCET.

WCET estimation methods can be split into two classes [22]: *measurement based* approaches and *static analysis* approaches. The former class of methods executes the task under study to derive a WCET estimation. It has the advantage to provide an estimation when the processor micro-architecture is not completely known. On the other hand it may underestimate the actual WCET if test-case generation does not cause the longest path to be measured, and thus may raise safety issues. The latter class of method, in contrast, statically analyzes the software and hardware under study. It provides a safe WCET upper bound but requires detailed knowledge of the processor micro-architecture.

Static WCET estimation methods are generally divided into two steps, commonly named *high-level* analysis and *low-level* analysis. The *high-level* analysis determines the longest execution path among all possible flows in a program. The *low-level* analysis is used to account for the processor micro-architecture. A number of static analysis methods have been designed in the last two decades at both levels (see [22] for an extensive survey of methods and tools). Concerning the low-level analysis, methods have been designed to predict WCETs in architectures equipped with caches [21, 16, 8, 11] or pipelines [7, 14].

A common implicit assumption in all static WCET estimation methods is that the hardware is not subject to faults. However, technology scaling, used to increase performance, has the negative consequence of providing less reliable silicon primitives due to static and dynamic variations [1, 2], resulting in an increase of the probability of failure (*pfail*) of circuits. The recently published resilience roadmap [17] underlines the magnitude of the problem we will very soon face. According to [17], the increase of *pfail* with scaling will be particularly significant for SRAM cells, for which the predicted *pfail* is 6.1e-13 at 45nm and will increase to 2.6e-04 at 12nm.

Existing space redundancy techniques defined to deal with process and latent defects and provide fault-free chips at current technology nodes, like column/row sparing and error correcting codes (ECC), will not be affordable anymore in the future due to their non-scalable cost when used extensively to recover from permanent faults. Furthermore, using

ECC to prevent the effect of permanent faults will reduce the transient errors recovery capabilities.

Consequently, other approaches like fine grain disabling and reconfiguration (e.g. individual functional units or cache blocks) will become economically necessary. We are going to enter in a new era: functionally correct chips with variable performance from the time they are shipped. A recent study [12] has analyzed the effect of fine grain disabling resulting from permanent faults on average performance. It reveals that caches, which take most of die real-estate in current processors and contain numerous SRAM cells, will be a non-negligible source of performance degradation in the near future. Therefore, ignoring the effect of such faults by assuming fault-free chips during their lifetime in WCET estimation methods may lead to an underestimation of the WCET, even for static analysis methods.

To the best of our knowledge, no static WCET estimation method so far has been proposed to estimate the impact of permanent faults on worst-case execution times. Our work provides the first results on this topic, through the proposal of a static analysis method that evaluates the impact of faulty SRAM cells in instruction caches on WCETs. The trivial solution consisting in assuming all cache blocks as faulty is obviously safe. However, the probability that such a scenario occurs is very low (although non null) and such a method would largely overestimate the calculated WCET. Instead, our method, based on the probability that a SRAM cell is faulty, evaluates probabilistically how many additional cache misses (*fault-induced misses*) may occur. The distribution of the timing penalties due to faults is then used to derive probabilistic worst-case execution times. These probabilistic execution times can finally be used to ensure that timing constraints are met under a targeted probability depending on the software criticality level (e.g. 10^{-15} per task activation for aerospace commercial industry [20]). An essential benefit of our approach is that its probabilistic nature stems *only* from the presence of faults; by construction, due to the use of a base WCET estimation method based on static analysis, the worst-case execution path cannot be missed. This allows the use of our method in safety critical real-time systems. In addition, as demonstrated by experimental results, the proposed method is computationally tractable, since it avoids the exhaustive enumeration of all possible fault locations. Experimental results also show that the proposed method accurately estimates WCETs in the presence of permanent faults compared to a method that explores all possible locations for faults. The proposed method not only allows to quantify the impact of permanent faults on WCET estimates, but also can be used in architectural exploration frameworks to select the most appropriate fault management mechanisms.

The remainder of the paper is organized as follows. Section 2 first surveys related work. The fault model and the base WCET estimation technique extended to cope with faults are presented in Section 3. Section 4 describes the proposed probabilistic WCET estimation method. Experimental results are presented in Section 5. Finally, we summarize the contributions of our research and give directions for future work in Section 6.

2. RELATED WORK

WCET estimation techniques may use *static analysis* or *measurements* to derive WCET bounds [22]. Our method falls into the first category, which by construction provides safe WCET bounds. Produced WCET estimates may be either *deterministic* (strict upper bound that is never exceeded at run-time) or *probabilistic* (execution times that have a given probability to be exceeded). The method we propose produces *probabilistic* WCET bounds to reflect the uncertainty of the location of faults in the architecture. It can be classified as *static probabilistic timing analysis* (SPTA) method according to the terminology introduced in the Proartis project [3].

Many static timing analysis methods for systems equipped with cache memories (instruction caches, data caches, cache hierarchies, with various cache structures and replacement policies) have been proposed in the last two decades [21, 16, 8, 11]. These methods are by construction safe, provided that the details of the cache architecture are known. They are accurate if the cache structure, in particular the cache replacement policy, is carefully selected [19]. To our best knowledge, all static cache analysis methods assume fault-free caches, and are not sound anymore if the hardware is subject to permanent faults; our work is the first to integrate the effect of permanent faults in a static cache analysis method.

Some scheduling approaches exist to cope with faults while keeping time predictability. Most of them support transient faults, that can be supported using mechanisms such as checkpointing and rollback, or active replication. Such approaches aim at integrating the cost of error detection and error recovery/masking in tasks schedulability analysis [9, 5, 18]. In the scope of real-time systems, comparatively less research has addressed the effect of permanent faults and disabling of resources on application performance.

The study presented in [12] addresses this issue for non real-time applications. They assume the same fault model as in our method, but concentrate on average-case performance, whereas our focus here is on real-time applications and worst-case performance.

The objective and assumptions in [20] are very close to ours. A very similar fault model is used, and the objective is to derive WCET estimates in the presence of permanent faults and disabling of hardware elements. The difference lies in the type of method used to obtain WCETs and in the processor architecture considered. Regarding WCET estimation, [20] uses a *measurement-based probabilistic timing analysis* method (MBPTA), that determines probabilistic WCETs based on measurements, whereas we use static probabilistic timing analysis (SPTA). By construction MBPTA, in contrast to SPTA, does not necessarily explore all application paths, which can cause the longest path to be missed and thus does not provide an absolute guarantee that the computed WCETs are safe. Regarding the processor architecture, [20] assumes a PTA-compliant architecture (e.g. the architecture described in [13] that implements random cache placement and replacement). This is not a requirement in our method. In summary, the fundamental difference between the two approaches is that in [20] the probabilistic nature of the estimated WCETs stems from several factors: uncertainty in the path exploration procedure, probabilistic hardware timing, probabilistic fault location and fault count. In contrast, in our method, the only source of uncertainty comes from the presence of faults.

3. BACKGROUND

3.1 Architecture and fault model

The proposed analysis is defined for set-associative instruction caches implementing the Least Recently Used (LRU) replacement policy. A cache configuration is defined by a number of sets s, a number of ways per set w, and a block size in bits k. The architecture is considered free from timing anomalies, meaning that a cache miss leads to the worst-case behavior. For the scope of the paper, a single level of cache is assumed.

Only *permanent* faults affecting the instruction cache are considered. The rest of the architecture is assumed to be fault-free. The interaction between faulty micro-architectural components is considered out of the scope of this paper and left for future work. The effects of transient faults (e.g. particle strikes) are not addressed and are assumed to be captured by other error detection/correction codes.

In the cache, a cache block with at least one bit affected by a permanent fault is considered as faulty and is disabled. Furthermore, LRU-stack bits are assumed to be fault-free by using hardening and/or redundancy techniques.

Faulty cache blocks are assumed to be detected using post-manufacturing and boot-time tests, ECC, and built-in self-tests. Cache block disabling has already been successfully used in commercial processors [15].

The term *faulty cache configuration* is used to denote a cache with faulty blocks. Such a configuration is represented by a vector with an entry per set, indicating its number of faulty blocks $\in [0, \ldots, w]$. The exact position of the faulty blocks in each set has no importance, thanks to the LRU replacement: in case of block failure, the LRU stack of a set is reduced by its number of faulty blocks.

Each SRAM cell (i.e. bit) is assumed to have an equal probability of failure ($pfail$). $pfail$ corresponds to the probability of a bit to be permanently faulty due to process defects, aging, low-voltage operation... Moreover, permanently faulty SRAM cells locations are considered as random, to capture some major causes of uncorrelated faults [4].

Based on the above assumptions, the probability of a cache block failure p_{bf} can be determined from the probability of bit failure $pfail$ as follows:

$$p_{bf} = 1 - (1 - pfail)^k \tag{1}$$

And the probability $p_{wf}(i)$ to have i faulty ways in a set can be derived by using the binomial probability law:

$$p_{wf}(i) = \binom{w}{i} (p_{bf})^i (1 - p_{bf})^{w-i} \tag{2}$$

3.2 Base static WCET estimation technique

The proposed analysis extends the standard static WCET estimation technique briefly described hereafter, originally designed for fault-free architectures. Since our focus is on the impact of permanent faults on instruction caches, the low-level analysis for the scope of this paper only includes static instruction cache analysis. The base static WCET estimation technique is representative of the current state-of-the-art and is implemented in several tools [22]. The reader is referred to [22] and [21] for further details on the static analysis techniques extended in our work.

3.2.1 Low-level analysis

The contribution of instruction caches to the WCET is determined by associating a Cache Hit/Miss Classification (CHMC) to every memory reference. The CHMC we use, defined in [21], represents the worst-case behavior of each reference with respect to the instruction cache:

- *always-hit* (AH): the reference will always result in a cache hit,
- *first-miss* (FM): the reference could neither be classified as hit nor as miss the first time it occurs but will result in cache hits afterwards,
- *always-miss* (AM): the reference will always result in a cache miss,
- *not-classified* (NC): the reference could neither be classified as hit nor as miss.

CHMCs are obtained by applying the static analysis techniques described in [21], based on abstract interpretation. When using this technique, three analyses that operate on the program control flow graph are defined:

- the *Must* analysis determines if a memory block is *always* present in the cache at a given program point: if so, the reference CHMC is *always-hit*,
- the *Persistence* analysis determines if a memory block will not be evicted after it has been loaded; the CHMC of such references is *first-miss*,
- the *May* analysis determines if a memory block may be in the cache at a given point: if not, the reference CHMC is *always-miss*. Otherwise, if present neither in the Must analysis nor in the Persistence analysis the reference CHMC is *not classified*.

In order to avoid enumerating all possible (concrete) cache contents, each of these three analyses compute an *Abstract Cache State* (ACS) at every program point until a fixpoint is reached. The semantics of abstract cache states depends on the considered analysis. For instance, a block in the *Must* ACS at a given point is guaranteed to be in the cache at that point. Furthermore, for the *Must* and *Persistence* analyses, the age of a block in an ACS is the maximum possible age that block could have in the cache at run-time. The classifications of references as AH/FM are based on the maximal age of the reference in the corresponding ACS (e.g. a reference is classified AH if its age is lower or equal to w).

This concept of age in abstract cache states is at the core of the method. An in-depth understanding of static cache analysis methods is not mandatory to understand the method developed in this paper; further details can be found in [21].

3.2.2 High-level analysis

WCET calculation in this study uses the most prevalent technique, named IPET for *Implicit Path Enumeration Technique*. IPET is based on an Integer Linear Programming (ILP) formulation of the WCET calculation problem [22]. It reflects the program structure and the possible execution flows using a set of linear constraints. An upper bound of the program's WCET is obtained by maximizing the following objective function:

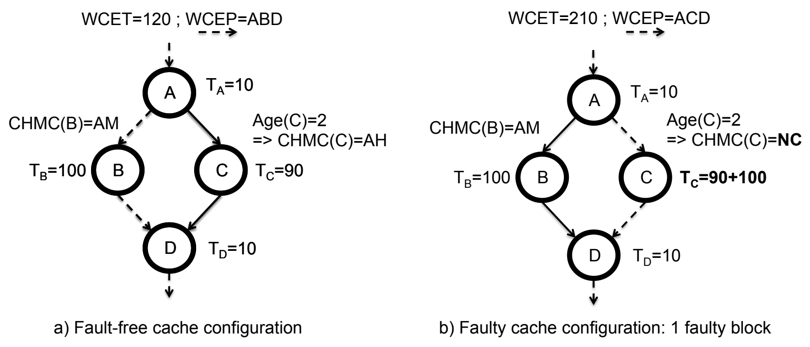


Figure 1: WCET and WCEP estimation for an architecture with a fully-associative 2-way cache (1 cycle cache/100 cycles memory latency). A reference is performed by each basic block of the conditional structure. A WCEP variation occurs between the fault-free configuration (a) and the faulty cache configuration (b).

$$\sum_{i \in BasicBlocks} T_i * x_i \tag{3}$$

T_i (constant in the ILP problem) is the timing information of basic block i. T_i integrates the effects of caches, and is computed using the CHCM of its references and the cache and memory latencies. x_i (variables in the ILP system, to be instantiated by the ILP solver) corresponds to the number of times basic block i is executed. The values of variables x_i, once set by the ILP solver to maximize the objective function, identify a path in the program leading to the estimated WCET. This path is termed Worst-Case Execution Path (WCEP).

4. PROBABILISTIC STATIC WCET ESTIMATION FOR FAULTY CACHES

4.1 Design rationale

Cache blocks with permanent faults are disabled, which reduces the cache capacity and leads to additional cache misses, named all along the paper *fault-induced misses*; a reference that was known to hit the cache for a fault-free cache may result in a cache miss in the presence of faults.

Consequently, when statically analyzing faulty caches, some references that were classified as hits (i.e. AH or FM) when ignoring faults may have to be changed to misses (i.e. NC) in the presence of faults. For instance, let us assume a 2-way set associative cache. For a given reference, if the cache line containing that reference has a maximal age determined by the *Must* analysis equal to 2, the fault-free analysis will classify the reference as AH. However, if there is one faulty block in the set, the reference has to be classified as NC. The estimated WCET is thus affected by faults due to the possible modifications of the timing information of individual basic blocks.

The WCEP may change as well as compared to the fault-free WCEP. This behavior is illustrated on Figure 1 on a toy example. In a fault-free execution, basic block C is not on the WCEP because the memory reference in C, always hits the cache. With the faulty-cache configuration, this access is classified as a miss and changes the timing information of C, which now becomes part of the WCEP.

Due to the possible WCEP variation in the presence of faults, a naive brute-force solution to compute the WCET probability distribution in the presence of faults would consist in estimating for each possible faulty-cache configurations the WCET and the corresponding probability of that configuration to happen. The probability of a faulty-cache configuration to occur can be computed as follows:

$$\prod_{i=1}^{s} p_{wf}(nbfaulty(i)) \tag{4}$$

where the function $nbfaulty(i)$ returns the number of faulty blocks of set i in the configuration.

However, this exhaustive computation is unusable in practice, due to the prohibitive number of faulty-cache configurations for which a WCET estimation has to be performed: $(w+1)^s$. The method we propose to deal with faulty caches provides probabilistic WCET estimates very close to the one of the exhaustive method, but is computationally tractable. The brute force method will only be used as a baseline in the performance evaluation in Section 5.

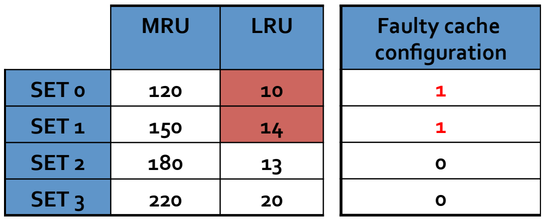
4.2 Method description

The proposed approach computes, for a given program, cache configuration and probability of cell failure, a WCET probability distribution. In order to avoid an exhaustive computation of WCETs for all combinations of fault locations, the proposed approach operates in two steps. It first computes the fault-free WCET and then determines an upper bound of the probability distribution of the time penalties caused by fault-induced misses. The WCET probability distribution is then obtained by adding these two components. The fault-free WCET estimation is performed using the static analysis described in section 3.2. Thus, at the core of the method is the computation of the penalty probability distribution described hereafter.

4.2.1 *Computation of fault-induced miss penalty probability distribution*

To progressively go into the details of our method, we first explain how the penalty is computed from a known faulty-cache configuration. Based on the independence of sets, the fault-induced miss penalty corresponds to the number of additional cycles resulting from fault-induced cache misses in each set.

For single-path code, the number of fault-induced misses can be easily derived from the results of the fault-free WCET analysis, since the number of cache hits for each set on the



	MRU	LRU	Faulty cache configuration
SET 0	120	10	1
SET 1	150	14	1
SET 2	180	13	0
SET 3	220	20	0

Figure 2: Number of references detected as hits per set and way, determined with the fault-free WCET computation for single-path code and a 2-way 4-set LRU cache. The number of fault-induced misses for the considered faulty cache configuration is equal to 10+14.

(unique) execution path is known. This is depicted in Figure 2. The figure shows for every set and every way, from the Most Recently Used (MRU) to the Least Recently Used (LRU) position in the LRU stack, how many accesses are detected as hits in that position. The position in the LRU stack is given by the maximal age of each reference provided by the cache analysis (*Must* and *Persistence* analyses). For instance, number 10 in the LRU position in set 0 means that 10 references are detected as hits in that position when set 0 is fault-free. Thus, there are $10 + 14$ fault-induced misses resulting from two faulty blocks, located in sets 0 and 1. which are disabled.

However, due to the WCEP variation effect presented in Figure 1, generalizing the method to multiple-path programs is not straightforward. Using the number of hits occurring along the fault-free WCEP may indeed lead to an underestimation of the number of references that cause misses and thus may lead to an underestimation of the fault-induced miss penalty.

To overcome this issue and ensure that the number of fault-induced misses is never underestimated, we compute for each set an *upper bound* of the number of references detected as hits instead of computing the exact value. This is performed separately for each set of the cache and for each possible number of faulty cache blocks $f \in [1, \ldots, w]$ in the set. The computation is made insensitive to path variations by considering the memory references on *all* paths, and not only the ones on the WCEP. More precisely, for a given set s_j and a given number of faulty blocks f, the upper bound of the number of fault-induced misses is obtained by an ILP formulation where the objective function to maximize is as follows:

$$\sum_{i \in BasicBlocks} R_i * x_i \tag{5}$$

in which R_i stands for the number of references of basic block i which map onto set s_j and determined by the cache analysis to have a maximal age in the f latest LRU position. x_i is the number of times basic block i is executed. This ILP formulation is subject to the same set of flow constraints as the IPET formulation used to compute the WCET (§ 3.2.2).

By solving this ILP system for all sets and all possible numbers of faulty blocks per set ($s * w$ different resolutions), we obtain a Fault Miss Map, noted FMM in the following. This map gives an upper bound of the number of fault-induced misses per set and per number of faulty blocks. In the map a row corresponds to a set and each column corresponds to a number of faulty blocks. Figure 3.a gives the FMM derived from the example of Figure 2.

Given a faulty-cache configuration and the FMM, the corresponding fault-induced latency is then:

$$Lat_{mem} * \sum_{i=0}^{s-1} \begin{cases} FMM[i][nbfaulty(i)] & \text{if } nbfaulty(i) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

with Lat_{mem} representing the memory latency.

In order to avoid exploring all possible faulty-cache configurations to obtain the penalty probability distribution, our method takes benefit of the *independence of sets*, as illustrated in Figure 3.b. The penalty distribution is first computed for each cache set independently by studying the respective impact of $f \in [1, \ldots, w]$ faults. For each set, the distribution is composed of at most $w + 1$ different penalties that correspond to 0 up to w faulty blocks with the probability for each case given by p_{wf} (equation 2). Since sets are independent, all these discrete distributions can be combined using convolutions to obtain an upper bound of the penalty distribution.

4.2.2 Tightness improvement

While the method as described so far never underestimates the number of memory references that may be affected by faults, it may be pessimistic. Indeed, mutually exclusive accesses (i.e. accesses performed along the two branches of an if-then-else construct) mapped to different sets are *all* considered as causing fault-induced misses. This phenomenon is particularly important when entire sets are faulty, because references target the MRU position more significantly than the other positions. To improve the tightness of the distribution of penalties at a reasonable computation cost, we focus on such situations.

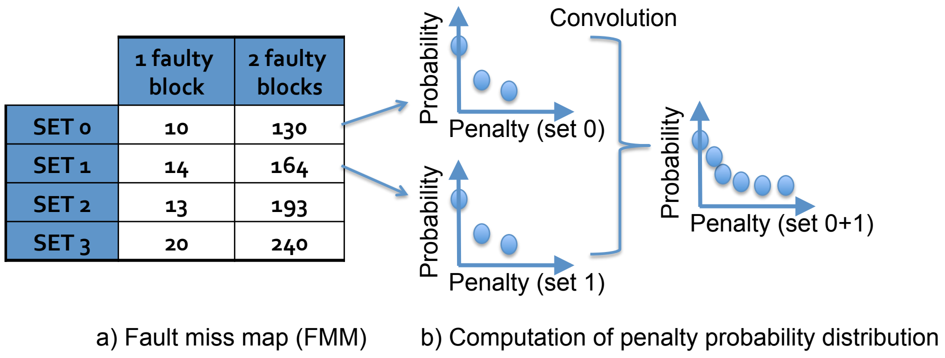
We estimate the worst-case penalty when sf sets ($sf \in [1, \ldots, s]$) are entirely faulty and $s - sf$ sets have all their blocks faulty except the MRU block. Let us denote by $penalty_{sf}$ the computed worst-case penalty. After having computed the distribution of penalties as described before (§ 4.2.1), the distribution is corrected to eliminate overestimated penalties. For each value of the distribution corresponding to sf entirely faulty sets, none of the corresponding penalty values can exceed $penalty_{sf}$ meaning that the values above that penalty can safely be reduced to $penalty_{sf}$.

To estimate the worst-case penalty for sf entirely faulty sets, the IPET formulation that was presented in section 3.2.2, is extended. The key idea of that extension is to compute the WCET in the presence of sf entirely faulty sets. The worst-case penalty is then obtained by subtracting the fault-free WCET already computed.

In the ILP formulation, the presence of an entirely faulty set j is represented by a binary variable sf^j set to 1 when all blocks in the set are faulty. The following constraint limits to sf the number of sets that can be entirely faulty:

$$\sum_{j=0}^{s-1} sf^j \leq sf \tag{7}$$

The penalty can be separated in two distinct parts: (i) the penalty resulting from all accesses that hit exclusively in the MRU position (noted P_i^j for the accesses of basic block i that hit in the MRU position of set j) and, (ii) the penalty resulting from all accesses that hit but not in



	1 faulty block	2 faulty blocks
SET 0	10	130
SET 1	14	164
SET 2	13	193
SET 3	20	240

Figure 3: Fault miss map (FMM) derived from the hits per sets and ways of Figure 2 (a). Example of computation of the distribution of fault-induced miss penalties for the first two sets (b).

the MRU position (noted P'_i for the accesses of basic block i). Thus, The IPET objective function can be extended as follows:

$$(\sum_{i \in BasicBlocks} (T_i + P'_i + \sum_{j=0}^{s-1} P_i^j * sf^j) * x_i) \quad (8)$$

In this formula, the penalty due to accesses in the MRU position is represented by the sum of the penalties coming from the MRU position of each set j times sf^j.

However, equation 8 is in a quadratic form, because sf^j and x_i are two variables of the ILP system. To obtain a linear formulation, we introduce the variable ff_i^j which is equal to x_i if sf^j is set to 1 and equal to 0 otherwise, which is modeled by the following constraints:

$$ff_i^j \leq x_i \quad \text{and} \quad ff_i^j \leq M * sf^j \quad (9)$$

where M[1] is a constant higher than any possible x_i.

The objective function can thus be rewritten as follows:

$$(\sum_{i \in BasicBlocks} (T_i + P'_i) * x_i + \sum_{j=0}^{s-1} P_i^j * ff_i^j) \quad (10)$$

Finally, to reduce the explored solution space and thus to improve the solver's resolution time, an extra constraint on the objective function is added, indicating an upper bound of that function. This upper bound corresponds to the sum of all the fault-induced misses times the memory latency plus the fault-free WCET. Formally:

$$(10) \leq \sum_{i=0}^{s-1} FMM[i][w] * Lat_{mem} + WCET_{fault-free} \quad (11)$$

s distinct resolution of the above ILP formulation are needed to determine the values of $penalty_{sf}$.

4.3 Complexity considerations

The base method that was presented in this section requires one run of the WCET estimation tool to estimate the fault-free WCET. Then, $s * w$ resolutions of ILP systems, whose complexities are similar to the ones of the WCET calculation step, are required to obtain the fault miss map (FMM). Finally, to obtain the distribution of fault-induced miss penalties, convolutions of the distributions of the s sets have to be performed (complexity $O(log(s))$). This complexity has to be compared to the $(w+1)^s$ runs of the WCET estimation tool that would be required if all faulty configurations were exhaustively examined. The method we have proposed to improve the tightness of the base method requires additional s ILP system resolutions. Measured analysis times will be given in Section 5.

[1]In our experiments M is fixed to MAX_INT.

Name	Description
SetAssoc	2-way set associative cache, block size of 64 bytes, 8 sets, pfail of 10^{-4}, cache latency of 1 cycle, memory latency of 100 cycles
DirectMapped	direct-mapped cache, block size of 64 bytes, 16 sets, pfail of 10^{-4}, cache latency of 1 cycle, memory latency of 100 cycles

Table 2: Cache configurations

5. EXPERIMENTAL RESULTS

5.1 Experimental setup

Analyzed codes.

The experiments were conducted on 25 benchmarks from the Mälardalen WCET benchmark suite[2]. Table 1 summarizes the applications' characteristics. The analyzed programs include both single path programs (*fdct, jfdctint, mincer, prime, ud, matmult*) and multi-path programs. They include both small loop intensive programs and control programs, containing less loops and more complex flows of control.

Cache configurations.

The instruction cache size is fixed to 1KB in all experiments. To estimate the accuracy of our method, two config-

[2]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Name	Description	Code size (Bytes)
adpcm	Adaptive pulse code modulation algorithm	7644
bs	Binary search for the array of 15 integer elements	328
bsort100	Bubblesort program	588
cnt	Counts non-negative numbers in a matrix	816
compress	Data compression program (adopted from SPEC95 for WCET-calculation)	2724
crc	Cyclic redundancy check computation on 40 bytes of data	1368
expint	Series expansion for computing an exponential integral function	940
fdct	Fast Discrete Cosine Transform	3368
fft	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm	6244
fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample	4460
insertsort	Insertion sort on a reversed array of size 10	472
jfdctint	Discrete-cosine transformation on a 8x8 pixel block	3048
lcdnum	Read ten values, output half to LCD	608
ludcmp	LU decomposition algorithm	3092
matmult	Matrix multiplication of two 20x20 matrices	848
minver	Inversion of floating point matrix	5312
ndes	Complex embedded code	3820
ns	Search in a multi-dimensional array	600
nsichneu	simulate an extended Petri Net	44028
prime	Calculates whether numbers are prime	512
qurt	Root computation of quadratic equations	2412
select	A function to select the Nth largest number in a floating point array	1892
sqrt	Square root function implemented by Taylor series	780
statemate	Automatically generated code generated by the STAtechart Real-time-Code generator STARC	8768
ud	Calculation of matrixes	2348

Table 1: Analyzed benchmarks

urations, listed in Table 2, are selected. These two configurations have a sufficiently small number of different faulty cache configurations ($6561 = 3^8$ for *SetAssoc* and $65536 = 2^{16}$ for *DirectMapped*) to allow an exhaustive computation of the WCET distribution for all possible fault locations.

A cache block is composed of the data bytes, the tag bits and their corresponding ECC bits assumed to be SEC-DED codes [10]. As an example, for the *SetAssoc* configuration, for each 64 bytes cache block, there are 23 bits for the tag and 6 bits for the tag ECC, as well as 11 ECC bits to protect the data.

WCET analysis.

The experiments were conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 with no optimization and the default linker memory layout. The WCETs of tasks are computed by the Heptane static WCET estimation tool [6], that implements the cache analysis and WCET computation steps as presented in Section 3. The experimental results presented hereafter only account for the contribution of instruction caches to the WCET. The effects of other architectural features are not considered. Cplex 12.5 is used to solve the different ILP systems.

5.2 Experimental results

In this section, the accuracy of the proposed methods is evaluated. Measurements of the analysis computation time are provided. Finally, some first results on the impact of architectural parameters are given.

Accuracy of probabilistic WCET estimates.

We evaluate the accuracy of the proposed method by comparing it to a baseline method, named *Exhaustive* hereafter, that enumerates all faulty cache configurations to obtain the WCET probability distribution. The results of the two versions of the proposed method, named *Base* (§ 4.2.1) and *Improved* (§ 4.2.2), are given and compared to the baseline by using the inverse cumulative distribution function. This curve indicates for a given targeted probability p (e.g. 10^{-15} per task activation for aerospace commercial industry [20]) the value at which the random variable WCET will be equal to, or below, with probability $1-p$. What is meant by *accuracy* in the following is the difference between the WCETs computed by the proposed methods and the ones produced by the baseline.

The probability of failure *pfail* is fixed to 10^{-4} which is representative of the highest assumed probability of cell failure in related work [17, 20]. Other smaller *pfails* have been used during our experiments (not shown here for space constraints). We found that the worst accuracy occurred when the highest *pfail* is used.

An analysis of the results for all benchmarks reveals that the benchmarks can be classified into three different categories according to the accuracy of our method:

- *perfect* (identical to the *Exhaustive* method): *fdct, jfdctint, minver, prime, ud, matmult, nsichneu, ns, insertsort, bsort100*
- *very accurate* (very close to the *Exhaustive* method): *adpcm, fir, qurt, sqrt, fft, ludcmp*
- *accurate* (close to the *Exhaustive* method): *bs, cnt, compress, crc, expint, lcdnum, ndes, select, statemate*

For space considerations, only a subset of the results of the analyzed 25 benchmarks are given in Figure 4 for the two considered cache configurations (*DirectMapped* on the left, *SetAssoc* on the right). Presented results are selected to have at least one benchmark in one of the three categories distinguished above. For the last two categories, the benchmarks are chosen to show the worst observed accuracy. Each graph depicts the inverse cumulative distribution function of the *Exhaustive*, *Base* and *Improved* methods. The x-axis corresponds to the WCET (in cycles) and the y-axis, in log scale, corresponds to the probability. By construction,

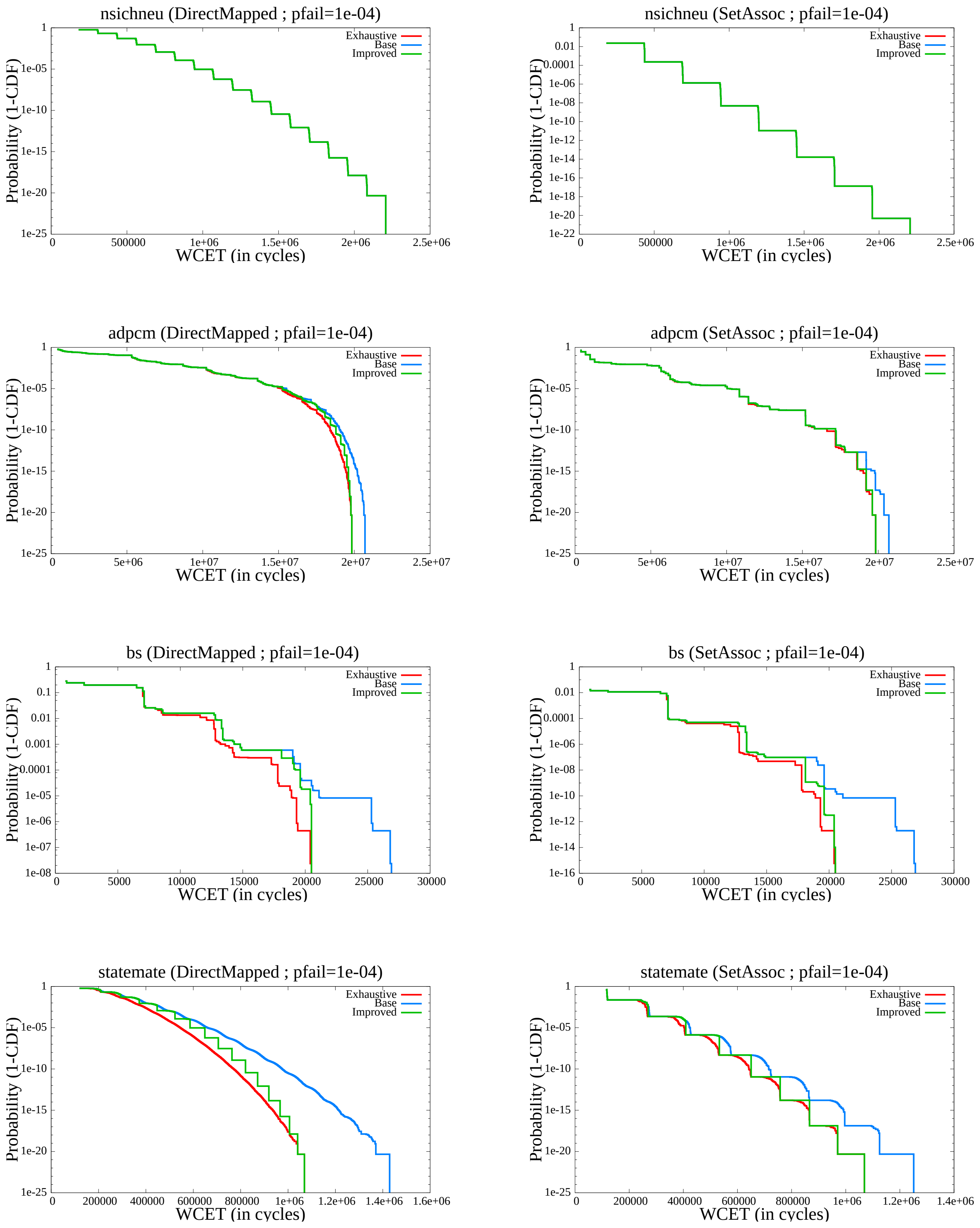


Figure 4: Inverse cumulative distribution function for nsichneu, adpcm, bs and statemate

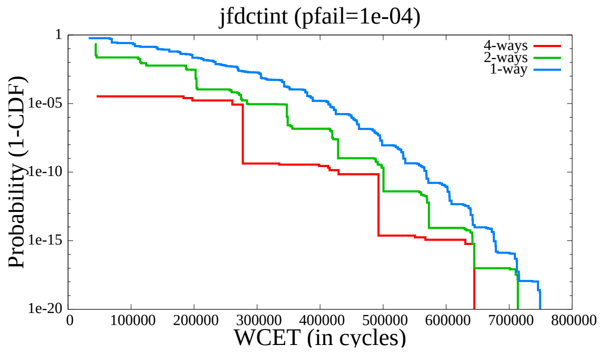


Figure 5: Impact of the cache associativity on the WCET distribution

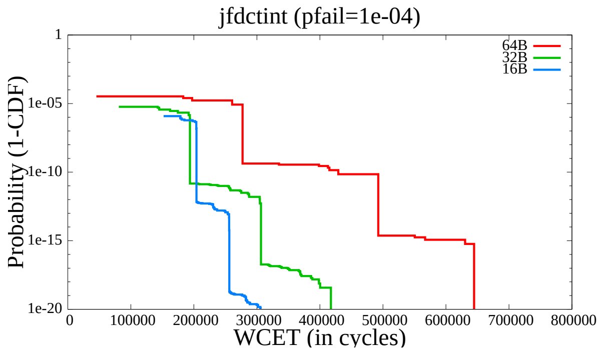


Figure 6: Impact of the cache block size on the WCET distribution

the *Base* and *Improved* curves are two upper bounds of the *Exhaustive* distribution meaning that they are equal to or above the *Exhaustive* curve.

The general observation that can be made on all graphs is that the presence of permanent faults significantly increases WCET estimates.

Regarding accuracy, for the *perfect* category, represented by *nsichneu* in Figure 4, the *Base* and *Improved* curves are identical to the *Exhaustive* distribution. This category is composed of two groups of benchmarks: single-path and multi-path applications. For the former group, this behavior is expected since the memory accesses are known and are the same whatever the execution conditions. For the latter group, we found that these benchmarks (*nsichneu, ns, insertsort, bsort100*) do not have *else* blocks in the conditional constructs of their code. This means that even if they are not single path, there is no overestimation for these codes of the number of memory references when computing the impact of faults.

The second category, illustrated by *adpcm* in Figure 4, reveals the presence of a small overestimation of the *Base* curve as compared to the *Exhaustive* curve, and this when nearly all the cache blocks are faulty (right part of the distribution). This overhead is due to *if-then-else* constructs in their code which induces a few additional memory references in the *Base* method as compared to the *Exhaustive* method. The *Improved* method eliminates most of this overestimation and the resulting distribution is very accurate.

The last category, depicted in Figure 4 with *bs* and *statemate*, shows that a significant overestimation can be observed (up to nearly 40% for *statemate*) when comparing the *Base* method to the *Exhaustive* method. For this set of benchmarks, we found that their code contains *if-then-else* or *switch* constructs inside loops, which results in a significant overestimation of the memory references in the *Base* method. The *Improved* method reduces significantly this overestimation, and even in that case, the *Improved* method produces accurate distributions.

Finally, in all the experiments, even if the shapes of the curves vary across the cache configurations, the accuracy of our method is only marginally affected by the cache configuration. The code structure is the main factor impacting the method accuracy.

Computation time.

The measurement provided below are performed using an Intel Core i7 (2.9GHz) under OSX 10.8.2 with 8GB of DDR3 (1600MHz). Cplex 12.5 is set to run up to 4 threads in parallel to solve ILP systems. In terms of computation time, the longest benchmark to analyze is *nsichneu* which is the biggest benchmark used in our experiments (code size of 43KB). For that benchmark, the computation time needed to get the fault-free WCET is 0.26 seconds and the time to compute the FMM is around 5 seconds for both cache configurations. The computation of the different worst-case penalties for that benchmark takes respectively 9 seconds for the *SetAssoc* configuration and 270 seconds for the *DirectMapped* configuration. When looking at the other benchmarks, the computation takes at most 10 seconds, all steps included. To put it in perspective, for the configurations that allow an exhaustive computation, more than 1600 seconds (*SetAssoc*) and 16000 seconds (*DirectMapped*) are needed for *nsichneu*.

Exploration of architectural parameters.

As seen during the experiments evaluating the accuracy of our method, the impact of permanent faults on the WCET is significant. There is thus a need to mitigate the effect of such faults to minimize their resulting penalty. The proposed method facilitates the exploration of design tradeoffs to address this challenge.

In this section, we explore the cache parameters as a first step in that direction. All along this section, the *Improved* method will be used. We first present the impact of the cache associativity on the WCET distribution. The WCET distributions obtained by our method for three associativity degrees are presented in Figure 5 for *jfdctint*. The three depicted cache configurations only differ by their associativity degree (configurations *DirectMapped* and *SetAssoc* from Table 2 and a third configuration with an associativity degree of 4). The results show that the higher the associativity, the lower the impact on the WCET. A similar behavior is observed for all benchmarks. In other words, selecting a set-associative cache instead of a direct-mapped can significantly reduce the WCET that has to be considered for a given targeted probability. This is explained by the fact that the probability to have all the blocks faulty in a set for set-associative caches is lower than the probability to have one block faulty in a direct-mapped cache.

The second studied cache parameter is the size of the cache blocks. Based on the previous observation, we focus on 4-way set associative cache with respectively 16B, 32B and 64B cache block size. The WCET distributions are depicted in Figure 6 for *jfdctint*. The results show that even if the fault-free WCET is higher when smaller cache blocks are used, the obtained distribution highlights the fact that smaller blocks are profitable when considering the impact of permanent faults. This conclusion can be made for all the benchmarks and is explained by the fact that the probability of a block failure depends on the size of blocks, the lower the number of bits in a block, the lower is the probability that the block is faulty.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a method to calculate a probabilistic WCET bound in the presence of permanent faults in instruction caches. The method is based on static analysis, and as such is guaranteed to always detect the longest execution path. Its probabilistic nature only comes from the presence of faulty blocks in the architecture. The method is shown to be computationally tractable, and does not require an exploration of all the possible locations of faults. Experimental results show that the method is accurate, in the sense that its provides probabilistic WCETs close to (but never under) the method that exhaustively explores all possible locations for faults.

So far, our method has focused on faults in instruction caches. Generalizing it to other micro-architecture components with SRAM cells (e.g. branch predictors, data caches, cache hierarchies) and studying the interactions between architectural components is a first direction for future work. Comparison with similar methods based on measurements [20] is another direction for forthcoming work. A more general direction would be to integrate our method in an architectural exploration framework, to decide of the most appropriate fault management mechanism for all architectural elements.

Acknowledgments

The authors would like to thank Benjamin Lesage, Bastien Pasdeloup and Jaume Abella for their fruitful comments on earlier versions of this paper.

7. REFERENCES

[1] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *DAC40*, pages 338–342, June 2003.

[2] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar. Circuit techniques for dynamic variation tolerance. In *DAC46*, pages 4–7, New York, NY, USA, 2009. ACM.

[3] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinellei, L. Kosmidis, C. Lo, and D. Maxim. Proartis: Probabilistically analysable real-time systems. *ACM Trans. Embed. Comput. Syst.*, 2013.

[4] L. Cheng, P. Gupta, C. J. Spanos, K. Qian, and L. He. Physically justifiable die-level modeling of spatial variation in view of systematic across wafer variability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(3):388–401, 2011.

[5] P. Chevochot and I. Puaut. Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies. In *6th International Conference on Real-Time Computing and Applications Symposium*, pages 356–363, 1999.

[6] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 37–44, Delft, The Netherlands, June 2001.

[7] J. Engblom. *Processor pipelines and static worst-case execution time analysis*. PhD thesis, Uppsala University, 2002.

[8] C. Ferdinand and R. Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 16–30, 1998.

[9] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(3):272–284, 1997.

[10] R. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[11] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th Real-Time Systems Symposium*, pages 456–466, Dec. 2008.

[12] D. Hardy, I. Sideris, N. Ladas, and Y. Sazeides. The performance vulnerability of architectural and non-architectural arrays to permanent faults. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'12, pages 48–59, 2012.

[13] 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*, DATE '13, pages 513–518, 2013.

[14] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet estimation. *Real Time Systems Journal*, 34(3), Nov. 2006.

[15] C. McNairy and J. Mayfield. Montecito error protection and mitigation. In *HPCRI '05: 1st Workshop on High Performance Computing Reliability Issues*, 2005.

[16] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3):217–247, 2000.

[17] S. R. Nassif, N. Mehta, and Y. Cao. A resilience roadmap. In *DATE*, pages 1011–1016, 2010.

[18] S. Punnekkat, A. Burns, and R. Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems*, 20(1):83–102, 2001.

[19] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[20] M. Slijepcevic, L. Kosmidis, J. Abella, E. Q. nones, and F. J. Cazorla. DTM: Degraded test mode for fault-aware probabilistic timing analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2013. To appear.

[21] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, 2000.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.