Accelerating Fault-Tolerant Real-Time Classification with IDK Classifiers

Sanjoy Baruah[0000−0002−4541−3445] and Alan Burns[0000−0001−5621−8816]

[1] Washington University in St. Louis, USA (baruah@wustl.edu)
[2] The University of York, UK (burns@york.ac.uk)

Abstract. We consider the problem of rapid, fault-tolerant classification using *IDK classifiers*—components that return a class label or "I Don't Know" if confidence is insufficient. Multiple such classifiers may be available, each with different speed-accuracy trade-offs. Prior work has developed scheduling algorithms to minimize expected classification duration under classifier faults. We present improved schemes that achieve lower expected classification durations while maintaining fault tolerance.

Keywords: Classification · Hard-real-time· Fault Tolerance

1 Introduction

Classifiers are essential software components for categorizing inputs into predefined classes. In autonomous Cyber-Physical Systems (CPS), perception increasingly relies on Deep Learning-based classifiers [**?**], which must deliver accurate, real-time predictions on resource-constrained platforms. However, modern ML research often favors accuracy over runtime efficiency, leading to high-latency models with marginal accuracy gains—e.g., a tenfold increase in runtime yields only slight improvements on ImageNet [**?**,**?**].

IDK Classifiers. IDK classifiers improve runtime efficiency by using fast classifiers for easy inputs and deferring harder cases to slower, more accurate models. Multiple IDK classifiers with varying speed-accuracy trade-offs can be organized into an *IDK cascade* [**?**], where classifiers are applied sequentially until a real class is returned. A final deterministic classifier ensures classification completion. Recent work [**?**,**?**,**?**,**?**] provides algorithms to synthesize such cascades, optimizing expected classification time under optional deadline constraints.

Faulty Classifiers. Recent work [**?**] extends prior cascade synthesis algorithms by incorporating fault tolerance, enabling timely and accurate classification despite faulty IDK classifiers. Specifically, [**?**] presents an algorithm that minimizes expected classification time under fault-free conditions, while ensuring correct classification by a deadline under a defined fault model (see Section **??**).

Cascades Reconsidered. While the cascade approach proposed by Wang et al. [**?**] is elegant and effective for minimizing expected execution time in fault-free settings, we argue that it is overly restrictive when fault tolerance is required.

Predetermining a fixed classifier sequence limits flexibility; instead, better performance can be achieved by dynamically selecting the next classifier at runtime based on prior successes and failures, guided by precomputed decision logic.

This Research. Prior work on fault-tolerant classification using IDK classifiers has focused exclusively on cascades [**?**], as reflected in the very title of that paper. In this work, we explore a *dynamic, non-cascade-based* alternative aimed at minimizing expected classification duration under fault conditions. We demonstrate that our approach can outperform the cascade-based state-of-the-art [**?**], offering significant improvements. Our key **contributions** are:

- A *non-cascade-based strategy* for fault-tolerant use of IDK classifiers, shown via example to reduce expected classification time.
- *Algorithms* for both offline preprocessing and online decision-making, along with analysis.
- *Evaluation* of our method's scalability and its performance gains over prior work [**?**].

Organization. The remainder of the paper is organized as follows. Section **??** provides background: (i) the formal model for systems of IDK classifiers, (ii) the fault model from [**?**], and (iii) a review of relevant prior results. Section **??** presents our main conceptual contributions, beginning with a motivating example and followed by the development and analysis of an alternative to pre-synthesized cascades. Section **??** reports on experimental evaluation, assessing both performance and scalability using synthetic workloads and benchmarks from [**?**]. We conclude in Section **??** with a summary and directions for future work.

2 A Model for IDK Classifiers

We adopt the formal model for IDK classifiers commonly used in real-time systems literature [**?**,**?**,**?**,**?**,**?**]. We assume n IDK classifiers $K_0, K_1, \ldots, K_{n-1}$ and a deterministic classifier K_{det}, all targeting the same classification task. Classifier outputs are not assumed to be probabilistically independent; instead, their joint behavior is represented by 2^n regions in a Venn diagram (e.g., Figure **??** for $n = 3$), each corresponding to a possible combination of classifiers returning either a class or IDK. The probabilities of these regions can be estimated empirically using profiling methods [**?**,**?**]. The system is further characterized by worst-case execution times $C_0, \ldots, C_{n-1}, C_{\text{det}}$ for each classifier. Algorithms have been developed to synthesize IDK cascades minimizing expected classification time, with the most general known algorithm [**?**] having worst-case complexity $\mathcal{O}(4^n)$.

Incorporating Fault Tolerance

A *fault* occurs when an IDK classifier returns an incorrect class (i.e., not IDK) that differs from the ground truth. A *fault model* defines the types of such failures; we adopt the model from [**?**], which introduces *exclusivity sets* to capture

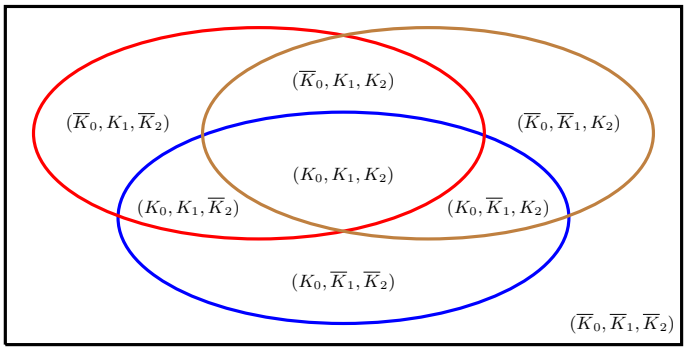

Fig. 1: (From [?]) Venn diagram illustrating the 2^n disjoint regions of the probability space for $n = 3$ IDK classifiers and one deterministic classifier. The blue, red, and brown ellipses represent the regions where classifiers K_0, K_1, and K_2 return real classes (i.e., not IDK). The surrounding rectangle indicates the deterministic classifier's success region (i.e., all inputs). Each of the $2^3 = \mathbf{8}$ regions is labeled by a 3-tuple, where K_i denotes success and $\overline{K}_i$ indicates IDK output by classifier K_i.

correlated failures among classifiers. Two classifiers form an *exclusivity pair* if they cannot validate each other's outputs (e.g., due to shared sensor input). The **exclusivity set** of classifier K_i, denoted $\mathcal{E}(K_i)$, contains all classifiers with which K_i forms exclusivity pairs. These sets are symmetric ($K_i \in \mathcal{E}(K_j) \Rightarrow K_j \in \mathcal{E}(K_i)$) and reflexive ($K_i \in \mathcal{E}(K_i)$), since repeating the same classifier offers no fault-detection benefit. However, exclusivity is not transitive: it is possible that $K_i \in \mathcal{E}(K_j)$ and $K_j \in \mathcal{E}(K_k)$, but $K_i \notin \mathcal{E}(K_k)$—e.g., if K_j uses two sensors, one shared with K_i and the other with K_k.

Tolerating Faults. We assume the deterministic classifier K_{det} is either fault-free or adequately handles its own faults via recovery mechanisms. To tolerate up to $F \geq 0$ faulty IDK classifiers, any classification must satisfy one of the following: (i) agreement among $F+1$ IDK classifiers that are mutually exclusive (i.e., not in each other's exclusivity sets), or (ii) a classification by K_{det}. Our *performance objective* is to minimize the expected execution duration under fault-free behavior, reflecting the assumption that faults are rare. However, if a hard deadline is specified, it must be met regardless of whether faults occur.

Run-time Algorithm. The run-time algorithm that we will develop executes classifiers one at a time until either: (i) $F+1$ classifiers that are not in each other's exclusivity sets have returned real (i.e., non-IDK) classes; or (ii) the deterministic classifier K_{det} is executed. If K_{det} is executed, then we return the class that it outputs. Otherwise, once $F + 1$ classifiers not in each other's exclusivity sets return real classes, we check whether or not these classifiers have returned the same class. If so, we return this class and are done, otherwise *a fault has been detected* and we immediately call the deterministic classifier K_{det} and return the class that is output by K_{det}. (As stated above, if a hard deadline is specified

within which classification must complete, this entire process may take no more than the specified deadline during both fault-free and faulty behaviors.)

3 A Fault-Tolerant Classification Algorithm

This section presents our main technical contribution: a proposed dynamic alternative to the state-of-the-art cascade-based approach of [**?**] for achieving fault-tolerant classification via IDK classifiers, that has smaller expected duration to successful classification. We first illustrate with an example, in Section **??**, that dynamically determining the next classifier to execute at run-time is superior to statically pre-determining (as a cascade does) the order in which the classifiers are to be called. The remainder of the section is devoted to deriving, evaluating, and explaining our proposed algorithm for such dynamic choosing of classifiers. We start out in Section **??** with a high-level overview, before presenting the detailed pre-processing and run-time algorithms in Sections **??** and **??** respectively. This is followed by a theoretical analysis in Section **??**, in which interesting and relevant properties of the algorithm (including its runtime complexity) are established.

Summarizing the Workload Model. Throughout this section we will use the workload model discussed in Section **??** that is commonly used in the real-time computing literature for describing instances of IDK classifiers, and assume that we have a collection $\mathcal{K}$ of IDK classifiers $K_0, K_1, K_2, \ldots$ and one deterministic classifier K_{det}, for the same classification problem. Hence the instance is completely specified as described in Section **??**, by $2^{|\mathcal{K}|}$ probabilities; the $|\mathcal{K}| + 1$ WCETs $C_o, C_1, C_2, \ldots$ and C_{det}; the number F of faults that need to be tolerated; an exclusivity set $\mathcal{E}_i$ for each IDK classifier K_i; and (optionally) a hard deadline within which classifications must always be completed.

3.1 A Motivating Example

We will now step through a contrived example that illustrates the benefits of choosing classifiers for execution dynamically, rather than pre-determining the sequence of classifiers beforehand prior to runtime. In the example we have four IDK classifiers K_0, K_1, K_2, and K_3, as well as a deterministic classifier K_{det}, for the same classification problem, with WCETs as follows:

classifier	K_0	K_1	K_2	K_3	K_{det}
WCET	1	48	25	25	C_{det} (value given later)

and probabilities of successful classification specified in Venn-diagram form in Figure **??**. From the Venn diagram in Figure **??** we learn that of the $2^4 = 16$ possible combinations of outcomes (individual classifiers returning a real class or IDK), only three happen with non-zero probability on any individual input:

1. All four IDK classifiers will return IDK (this happens with probability ϵ, where ϵ is a small positive real number: $\epsilon \approx 0.0$).

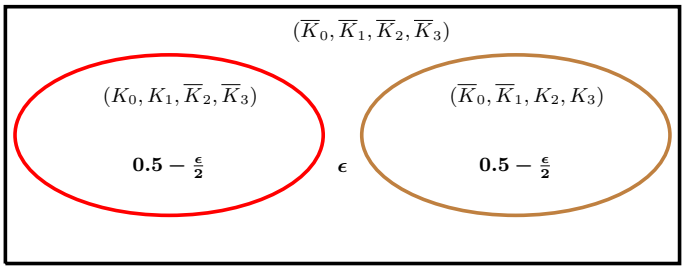

Fig. 2: Venn diagram depiction of probabilities of non-IDK classification for the example of Section **??**. Probability values are in **bold**, and only regions with non-zero probability are depicted.

2. Classifiers K_0 and K_1 will both return non-IDK classes while K_2 and K_3 will both return IDK; this happens with probability $\left(0.5 - \frac{\epsilon}{2}\right)$.
3. Classifiers K_2 and K_3 both return non-IDK classes while K_0 and K_1 will both return IDK; this, too, happens with probability $\left(0.5 - \frac{\epsilon}{2}\right)$.

(All other possible outcomes are assumed to have, in this contrived example, a zero probability of occurrence.)

Suppose no exclusivity pairs exist beyond reflexivity (i.e., $\mathcal{E}(K_i) = \{K_i\}$ for all i), and we aim to tolerate a single fault ($F = 1$). Then, in fault-free cases, two IDK classifiers must return non-IDK results. From Figure **??**, we observe that only either (K_0, K_1) or (K_2, K_3) can do so on any given input. Hence, any static cascade omitting one of these pairs will invoke K_{det} in about half of fault-free cases. Assuming $C_{\text{det}} = 500$ and $\epsilon \approx 0$, the optimal static cascade $\langle K_0, K_1, K_2, K_3, K_{\text{det}} \rangle$ has an expected duration of

$$0.5 \times (1 + 48) + 0.5 \times (1 + 48 + 25 + 25) = \mathbf{74}. \tag{1}$$

By comparison consider a dynamic strategy (depicted in Figure **??**) that begins with K_0:

- If K_0 succeeds, it runs K_1 for confirmation (leaf **A** in Figure **??**).
- If K_0 fails, it tries K_2 and then K_3 (leaf **C**), falling back to K_{det} only if K_2 also fails (leaf **E**).

This strategy yields an expected duration of

$$0.5 \times (1 + 48) + 0.5 \times (1 + 25 + 25) = \mathbf{50}, \tag{2}$$

an $\approx 33\%$ improvement over the optimal static cascade.

The Presence Of Hard Deadlines. Thus far in this example, we have assumed that either no hard deadline is specified, or it is large enough that meeting it is not an issue. The gap in performance between prior static cascade-based and the proposed dynamic approaches may become even more pronounced if hard

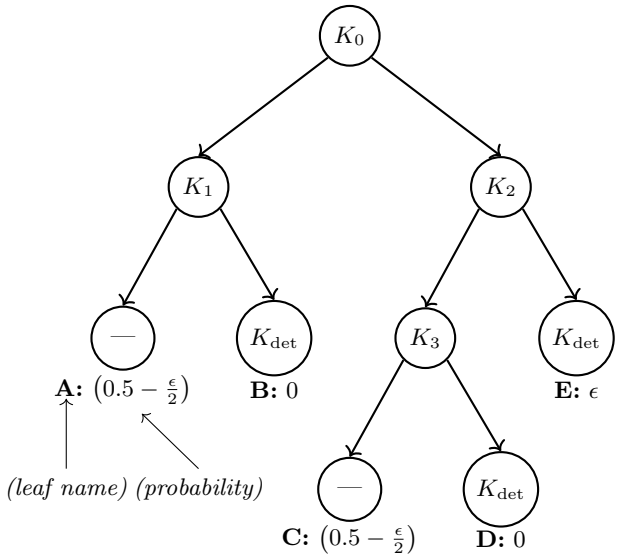

Fig. 3: Decision tree for fault-free behaviors (left branch: real class, right branch: IDK). Each leaf is labeled with a name and the probability of its becoming the terminating vertices in fault-free behavior.

deadlines must be considered. Suppose that some hard deadline $D \geq C_{\text{det}}$ were specified such that each classification must complete within a duration of D time units (note that if $D < C_{\text{det}}$ then this is impossible to guarantee). Since $C_0 + C_1 + C_2 + C_3 = 99$ for our example, then if $(D - C_{\text{det}}) < 99$ the static cascade cannot include all four classifiers. Now, the optimal cascade becomes $\langle K_0, K_1, K_{\text{det}} \rangle$ (rather than $\langle K_2, K_3, K_{\text{det}} \rangle$ since $C_0 + C_1 < C_2 + C_3$), with expected execution duration approximately:

$$(1 + 48) + 0.5 \times C_{\text{det}} \tag{3}$$

while the dynamic strategy discussed above remains valid for $D \geq 51 + C_{\text{det}}$, i.e., $D - C_{\text{det}} \geq 51$.[3] Hence for D satisfying:

$$51 + C_{\text{det}} \leq D < 99 + C_{\text{det}}.$$

[3] This follows from the observation that the three root-to-leaf paths in the decision tree of Figure **??** ending in K_{det} have execution durations

$$\begin{aligned}(C_0 + C_1 + C_{\text{det}}) &= 49 + C_{\text{det}} \\ (C_0 + C_2 + C_{\text{det}}) &= 26 + C_{\text{det}} \\ (C_0 + C_2 + C_3 + C_{\text{det}}) &= 51 + C_{\text{det}}\end{aligned}$$

respectively. Therefore, the strategy is "safe" – will not cause a deadline miss – provided $D \geq \max(49 + C_{\text{det}}, 26 + C_{\text{det}}, 51 + C_{\text{det}})$, i.e., $D \geq 51 + C_{\text{det}}$.

the expected duration under the dynamic approach is 50 whereas the expected duration under the static approach will increase with increasing value of C_{det}; hence the performance improvement of the dynamic approach as compared to the static one becomes more marked as C_{det} becomes larger.

3.2 An Overview of the Dynamic Strategy

Our classification strategy is to dynamically select the next classifier to execute at run-time, based upon what has happened thus far during the current classification attempt. Accordingly, at any time during runtime let us define the current **state** to be an ordered pair (S, T), where S is a set of classifiers ($S \subseteq \mathcal{K}$) and T a subset of S ($T \subseteq S$), denoting that the classifiers in S are those that have been executed thus far and of these, each classifier in the set T has returned a class other than IDK (and hence each classifier in $(S \setminus T)$ has returned IDK). We emphasize that S and T are *sets* – the order in which these classifiers were executed in arriving at the current state is not relevant. It has been previously observed [**?**, p. 369] that during classification using IDK classifiers the current state is defined "only [by] the set of classifiers, and not on their order," and that this fact reduces the number of states to be considered from a factorial function of the number of classifiers to merely an exponential one. For states defined as above – i.e., the ordered pair (S, T) – Theorem **??** (see Section **??**) shows that exactly $\left(3^{|\mathcal{K}|}\right)$ distinct states are possible. Prior to run-time, our preprocessing algorithm (Section **??**) will compute a lookup table

row	**state**	**next**	**cost**	$\boldsymbol{r_Y}$	$\boldsymbol{r_N}$
0	$(\emptyset, \emptyset)$	K_i	c	n_1	n_2
1	...	...	...	...	...
2	...	...	...	...	...
⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮

in which the number of rows equals the number of distinct states. This lookup table is interpreted as follows.

- Each row corresponds to a different state, which is specified in the column labeled **state**.
- When in a particular state during runtime, the next classifier one should execute in order to minimize the expected remaining duration to completion is the one specified in the column labeled **next** in the row corresponding to that particular state. The expected remaining duration to complete successful classification upon so doing is given in the column labeled **cost**.
- Upon executing this classifier, the row corresponding to the resulting state if the execution is successful (i.e., does not return the class IDK) is specified in the column labeled $\boldsymbol{r_Y}$, whereas if the execution is unsuccessful (returns IDK), the row corresponding to the resulting state is specified in the column labeled $\boldsymbol{r_N}$.

row	**state**	**next**	**cost**	r_Y	r_N
0	$(\emptyset, \emptyset)$	K_0	50	2	1
1	$(\{K_0\}, \emptyset)$	K_2	50	13	11
2	$(\{K_0\}, \{K_0\})$	K_1	48	8	6
3	$(\{K_1\}, \emptyset)$	K_2	50	17	15
4	$(\{K_1\}, \{K_1\})$	K_0	1	8	7
⋮					
6	$(\{K_0, K_1\}, \{K_0\})$	K_{det}	100	-	-
⋮					
8	$(\{K_0, K_1\}, \{K_0, K_1\})$	$\perp$	0	-	-
9	$(\{K_2\}, \emptyset)$	K_0	49	12	11
10	$(\{K_2\}, \{K_2\})$	K_3	25	48	46
⋮					
11	$(\{K_0, K_2\}, \emptyset)$	K_{det}	100	-	-
⋮					
13	$(\{K_0, K_2\}, \{K_2\})$	K_3	25	55	51
⋮					
27	$(\{K_3\}, \emptyset)$	K_0	49	30	29
28	$(\{K_3\}, \{K_3\})$	K_2	25	48	47
⋮					
51	$(\{K_0, K_2, K_3\}, \{K_2\})$	K_{det}	100	-	-
⋮					
55	$(\{K_0, K_2, K_3\}, \{K_2, K_3\})$	$\perp$	0	-	-
⋮					

Fig. 4: Selected rows of the lookup table for our example

– If the classifiers in T include $F + 1$ classifiers not in each others' exclusivity sets, then no further execution is needed; this is reflected by the column **next** containing a $\perp$ and the column **cost** a 0.

Some of the $3^4 = 81$ rows of the lookup table that is constructed for the example instance of Sec. **??** are depicted in Figure **??** (for $C_{\text{det}} \leftarrow 100$); the reader may verify that the decision tree of Fig. **??** is indeed implicitly embedded[4] in this table. This lookup table is used during runtime (after some additional preprocessing) to determine which classifier to execute next; the precise manner in which this is done is described in Section **??**.

3.3 The Preprocessing Algorithm

We now describe the algorithm that constructs the lookup table prior to runtime. This algorithm first initializes the table to have $3^{|\mathcal{K}|}$ rows with one row corresponding to each of the $3^{|\mathcal{K}|}$ possible states (see Theorem **??**). The first

[4] Since the initial state corresponds to Row 0 of the table in Figure **??**, the first classifier executed is K_0, with K_1 executed next if K_0 returns a real class (row 2) or K_2 if it returns IDK (row 1), and so on.

row corresponds to the initial state $(\emptyset, \emptyset)$, and the remaining rows correspond to different states in some canonical order, such that the row corresponding to any particular state can be identified in $\mathcal{O}(|\mathcal{K}|)$ time.[5] The column labeled 'cost' is initially set equal to the dummy sentinel value ∞ in all rows. As will be detailed later, the function EXPAND(S, T) (listed as Algorithm **??**) implements a top-down dynamic program that fills in the row in the lookup table corresponding to the state (S, T). Hence calling EXPAND$(\emptyset, \emptyset)$ fills in the first row of the lookup table; as we will soon see, the resulting recursive calls also fill in the other rows of the lookup table. Before providing a detailed description of this top-level function EXPAND(S, T) and explaining how a single call to EXPAND$(\emptyset, \emptyset)$ fills in the entire table, we first briefly discuss two helper functions that are used by EXPAND(S, T).

Helper Function FTCOUNT$(\boldsymbol{T}, \boldsymbol{F})$**.** For a set T of IDK classifiers and an $F \in \mathbb{N}_{\geq 0}$, this function *returns* TRUE *if there is a cardinality-*$(F+1)$ *subset of* T *for which the classifiers are all not in each others' exclusivity sets*, and FALSE otherwise.

As we will see in Theorem **??**, this function encapsulates an NP-hard problem. We solve it by exhaustive enumeration: if $K_{\text{det}} \notin T$, then each of the $\binom{|T|}{F+1}$ subsets of T of cardinality $(F+1)$ is examined separately to determine whether there is one in which no two classifiers comprise an exclusivity pair. Hence a call to this function has runtime complexity $\mathcal{O}\left(|\mathcal{K}|^{F+1} \cdot (F+1)^2\right)$.

Helper Function CONDPR$(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{K_j})$**.** For a given state (S, T) and IDK classifier $K_j \in (\mathcal{K} \setminus S)$, this function *computes the probability that* K_j *will return a class other than IDK when executed from state* (S, T).

When at state (S, T) during runtime, the IDK classifiers in S have been executed with those in T returning a real class and those in $(S \setminus T)$ returning IDK. Hence the probability space for the outcome if K_j were to be called next comprises all those regions in the Venn diagram for which the $|\mathcal{K}|$-tuple labeling the region contains a K_i for each $K_i \in T$ (since K_i returned a real class) and $\overline{K}_i$ for each $K_i \in (S \setminus T)$ (since K_i returned IDK), while containing either K_i or $\overline{K}_i$ for each $K_i \notin S$. Let $\mathcal{R}$ denote the collection of all these regions. For each region $r \in \mathcal{R}$, let $p(r)$ denote the probability associated with the region r in the specification of $\mathcal{K}$ (i.e., the probability that a randomly-drawn input will receive non-IDK classifications from exactly the classifiers that appear in non-negated form in the $|\mathcal{K}|$-tuple labeling region r). Let $\mathcal{R}' \subset \mathcal{R}$ denote the collection of all those regions in $\mathcal{R}$ for which the $|\mathcal{K}|$-tuple labeling the region contains K_j (rather than $\overline{K}_j$). The probability that K_j will return a class other than IDK

[5] Many schemes are possible, including binary search of the table, which takes

$$\mathcal{O}(\log(3^{|\mathcal{K}|})) = \mathcal{O}(|\mathcal{K}| \times \log 3) = \boldsymbol{\mathcal{O}(|\mathcal{K}|)} \text{ time.}$$

```
   // Update next and cost for the state (S,T), and return the
      updated cost
 1 if (( ∑_{K_ℓ ∈ S} C_ℓ ) + C_det > D) then                    //not a safe state
 2     return ∞
 3 if (FTCOUNT(T,F) = TRUE) then       //no need to execute any further
 4     cost ← 0; next ← ⊥; return 0
 5 if (cost ≠ ∞) then                 //This state has already been explored
 6     return cost
   // For each remaining classifier, investigate what happens if it is
      executed next
 7 for each K_j ∈ (K \ S) do                        //If K_j were executed next...
 8     Prob ← CONDPR(S,T,K_j)                        //Probability K_j succeeds
 9     cost_Y ← EXPAND(S ∪ {K_j}, T ∪ {K_j})         //cost if K_j succeeds...
10     cost_N ← EXPAND(S ∪ {K_j}, T)                 //...cost if it fails;
11     tmpCost ← C_j + Prob · cost_Y + (1 − Prob) · cost_N   //expected cost of
        executing K_j
12     if (cost > tmpCost) then                      //cheaper to execute K_j?
13         cost ← tmpCost
14         next ← K_j
15 if (cost > C_det) then
16     cost ← C_det
17     next ← K_det
18 return cost
```

Algorithm 1: EXPAND(S,T) – A Dynamic Program (using memoization)

when executed from state (S,T) is then the ratio

$$\left(\sum_{r\in\mathcal{R}'} p(r)\right) \div \left(\sum_{r\in\mathcal{R}} p(r)\right) \tag{4}$$

and hence this is what is returned by CONDPR(S,T,K_j). It is straightforward to determine this by examining all $2^{|\mathcal{K}|}$ regions, determining for each in $\mathcal{O}(|\mathcal{K}|)$ time whether it belongs to $\mathcal{R}$ and if so, to $\mathcal{R}'$ as well; hence a call to this function has runtime complexity $\mathcal{O}\left(2^{|\mathcal{K}|}\cdot|\mathcal{K}|\right)$.

Top-Level Function EXPAND(S,T). This function, presented in high-level pseudo-code form in Algorithm **??**, is essentially a memoization-based implementation of a dynamic program that *computes the minimum expected remaining duration to successful classification in fault-free behaviors* that is achievable from the input state (S,T). The lookup table is generated as an auxiliary data structure during the execution of this dynamic program (see, e.g., the subsection titled "Reconstructing the optimal solution" in [**?**, p. 387] for a textbook review of how such auxiliary data structures are an essential component of dynamic

programming). The call EXPAND(S,T) first checks in Line **??** whether the state (S,T) is an unsafe one because executing all the classifiers in S would exceed the deadline (if one is specified), in which case the expected duration that is returned is ∞. Then in Line **??** it checks whether no further execution is needed because the successful classifiers —the ones in T— satisfy the fault-tolerance requirement; if so, then a value of zero is returned.

Since the lookup table was initialized to have all entries in the *cost* column equal to ∞, a non-∞ value here implies that this state has already been visited in some prior recursive call, and the computed value cached in the *cost* column of the corresponding row – this is checked in Line **??**. Else (i.e., $cost = \infty$), this state is being explored for the first time.

The for-loop at Lines **??**–**??** explores, for each IDK classifier K_j that has not yet been executed, the option of executing it.

- In Line **??**, the helper function CONDPR(S,T,K_j) is called to determine the probability that classifier K_j would be successful if called from the current state (S,T). The next two lines (Lines **??** and **??**) make recursive calls to determine the expected remaining duration to successful classification when K_j returns a real class or IDK respectively. Line **??** then uses the results of the computations in the three prior lines to determine the expected remaining duration to successful classification if K_j is executed.
- The if statement beginning at Line **??** determines whether executing K_j results in a lower expected duration to successful classification than has been discovered thus far; if so, the *cost* and *next* fields in the corresponding row are updated accordingly.

And finally, the if statement beginning at Line **??** determines whether executing K_{det} results in a lower expected duration to successful classification than has been discovered thus far; if so, the *cost* and *next* fields in the corresponding row are updated accordingly.

The following example illustrates the workings of EXPAND(S,T). In this example we walk through a high-level trace of a call to EXPAND$(\emptyset,\emptyset)$ for the collection of classifiers $\mathcal{K}$ we had considered in Section **??**, thereby explaining how row 0 of the lookup table of Figure **??** is obtained.

Example 1. Let us suppose that EXPAND$(\emptyset,\emptyset)$ is called upon the example of Section **??** (with no deadline specified). It may be verified that the conditional tests of Line **??**, **??**, and **??** all fail and execution continues through to the for-loop beginning at Line **??**. This for-loop (Lines **??**–**??**) executes four times with K_j taking on each of the values K_0, K_1, K_2, and K_3 in succession.

The recursive call for $K_j \leftarrow K_0$ in Line **??** fills in Row 2, and the one in Line **??** fills in Row 1. While we will not trace these calls explicitly, it may be verified by re-reading Section **??** that the *cost* and *next* entries in these rows are indeed what a correct algorithm would conclude. (Consider, for instance, Row 1, corresponding to the state when K_0 has been executed and has returned IDK.

Our decision tree in Figure **??** tells us that the right thing to do here is execute K_2 next, and that doing so will have us execute K_3 next for a total duration of $C_2 + C_3 = \mathbf{50}$. Row 2 may be verified in a similar manner.)

The recursive calls for $K_j \leftarrow K_1$, $K_j \leftarrow K_2$, and $K_j \leftarrow K_3$ respectively fill in Rows 3–4, Rows 9–10, and Rows 27–28 of the table respectively; these, too, may be verified by referring back to the decision tree of Figure **??**.

With Rows 1–4, 9–10, and 27-28 so filled, let us see how the columns of Row 0 get assigned values. It may be verified from the Venn diagram of Figure **??** that *Prob* is assigned a value $(0.5 - \epsilon)$ in Line **??** for each of the four cases $K_j \leftarrow K_0, K_j \leftarrow K_1$, $K_j \leftarrow K_2$, and $K_j \leftarrow K_3$. With $\epsilon \approx 0$, *tmpCost* for these four cases evaluates as follows:

$$\begin{array}{ll}
\text{for } K_j \leftarrow K_0\text{:} & 1 + 0.5 \times 48 + (1 - 0.5) \times 50 \;\; = 50 \\
\text{for } K_j \leftarrow K_1\text{:} & 48 + 0.5 \times 1 + (1 - 0.5) \times 50 \;\; = 73.5 \\
\text{for } K_j \leftarrow K_2\text{:} & 25 + 0.5 \times 25 + (1 - 0.5) \times 49 = 62 \\
\text{for } K_j \leftarrow K_3\text{:} & 25 + 0.5 \times 25 + (1 - 0.5) \times 49 = 62
\end{array}$$

from which it is evident that *tmpCost* takes on its minimum value of 50 for $K_j \leftarrow K_0$, as reported in the columns labeled *cost* and *next* of Row 0. Since K_0 is the next classifier executed, r_Y points to Row 2 (representing the state when K_0 is executed and returns a real class) and r_N to Row 1 (representing the state when K_0 is executed and returns IDK).

This completes our explanation of how the call to EXPAND$(\emptyset, \emptyset)$ upon the example of Section **??** fills in row 0 of the table of Figure **??**. □

Compressing the Lookup Table. The lookup table computed by the call to EXPAND$(\emptyset, \emptyset)$ is compressed next, by removing the rows that are not needed for run-time use. Specifically, only the rows that are *reachable* from Row 0, i.e., those that are referenced directly or recursively in the r_Y and r_N columns starting from Row 0, need to be retained. (On the table of Figure **??**, for instance, only the **9** rows numbered 0, 1, 2, 6, 8, 11, 13, 51, and 55 are reachable from Row 0, and are hence the only ones of the $3^4 = \mathbf{81}$ rows that need be retained for runtime use.)

Should run-time memory be a concern, further compression is possible by also removing the *state* and *cost* columns from the rows that remain – these columns were only used for constructing the table, and will not be needed for runtime decision making.

How large can these tables be after compression? While one can come up with pathological examples where most of the $3^{|\mathcal{K}|}$ rows must be retained, our experience on the evaluation experiments (reported in Section **??**) has been that the number of rows in the compressed table is typically very small, rarely exceeding a number that is a low-degree polynomial of the number of classifiers. And in those cases where the table size is unacceptably large, one can adapt standard table-compression techniques to develop *approximation schemes* that trade off table-size for optimality.

```
1  // "T[r].colName" denotes the entry in the column that is labeled colName of
   row r of the lookup table (denoted T)
2  row ← 0
3  while TRUE do
4      if T[row].next = ⊥ then
5          if All non-IDK classifications obtained are the same then
6              return this classification
7          else                                              // fault detected
8              Call K_det and return its classification
9      else
10         Call the classifier specified in T[row].next
11         if this classifier returns a non-IDK class then
12             record this classification
13             row ← T[row].r_Y
14         else
15             row ← T[row].r_N
```

Algorithm 2: Runtime Algorithm

3.4 The Runtime Algorithm

The runtime algorithm that is executed for selecting the classifiers to execute when a new input needs to be classified is depicted in high-level pseudocode form as Algorithm **??**. In this pseudo-code, the variable *row* denotes the row corresponding to the current state of the system; it is initialized to 0 since Row 0 is the row corresponding to the initial system state, $(\emptyset, \emptyset)$, in the lookup table. The **while** loop beginning at Line **??** is repeatedly executed to choose the next classifier to execute — $T[row].next$ denotes the column *next* of the row *row* in the lookup table, denoted T in the pseudocode. If this equals $\perp$ (i.e., the **if** condition of Line **??** evaluates to TRUE), the current state satisfies the fault-tolerance condition. When this happens, the non-IDK classification decisions that have been made thus far are compared; if they are all equal, this classification is returned, else, a fault has been detected and so the deterministic classifier K_{det} is called and its classification decision returned.

If on the other hand the **if** condition of Line **??** evaluates to FALSE, then fault tolerance has not yet been achieved and additional calls to classifiers are needed. Lines **??**–**??** are executed: the classifier that should be executed from the current state is called, and the variable *row* updated appropriately depending upon whether this classifier returns a real class or IDK; if a real class, then this class is recorded (for comparison with other non-IDK classes that are returned by other classifiers, for purposes of fault detection in Line **??**).

3.5 Some Properties

We now prove the result, claimed in Section **??** and subsequently used in Section **??** (to dimension the lookup table that is constructed there), of the upper bound of $3^{|\mathcal{K}|}$ on the number of distinct states:

Theorem 1. *There are $3^{|\mathcal{K}|}$ distinct ordered pairs (S,T) where $S \subseteq \mathcal{K}$ and $T \subseteq S$.*

Proof. This is shown by a simple counting argument. For each i, $0 \leq i \leq |\mathcal{K}|$, there are $\binom{|\mathcal{K}|}{i}$ subsets of $\mathcal{K}$ of exactly i elements; hence, the number of distinct sets S of exactly i classifiers equals $\binom{|\mathcal{K}|}{i}$. Each such S has exactly 2^i distinct subsets; hence for each such S, T can take on exactly 2^i distinct values. The total number of (S,T) pairs is therefore equal to

$$\begin{aligned}
&\sum_{i=0}^{|\mathcal{K}|} \left(\binom{|\mathcal{K}|}{i} \cdot 2^i \right) \\
&= \sum_{i=0}^{|\mathcal{K}|} \left(\binom{|\mathcal{K}|}{i} \cdot 2^i \cdot 1^{|\mathcal{K}|-i} \right) \quad \text{//Note that } 1^{|\mathcal{K}|-i} = 1 \\
&= (2+1)^{|\mathcal{K}|} = 3^{|\mathcal{K}|} \quad \text{//Applying the Binomial Theorem}
\end{aligned}$$

and the theorem is proved.

We have described, in Section **??**, how the helper function FTCOUNT(T,F) is implemented using an approach of exhaustive enumeration, i.e., by considering all subsets of T of cardinality $(F+1)$ to determine whether there is one with no exclusivity pairs. We show below that one is not likely to be able to obtain a more efficient implementation than this, by proving that it is NP-hard to determine whether a set of IDK classifiers T contains at least $F+1$ mutually exclusive ones.

Theorem 2. *Given a set $T \subseteq \mathcal{K}$ of IDK classifiers and a positive integer F, determining whether T contains $(F+1)$ mutually exclusive classifiers is an NP-complete problem.*

Proof. The problem is clearly *in* NP: we can simply guess which classifiers in T are the $F+1$ mutually exclusive ones, and verify that none of them is in each other's exclusivity sets, all in polynomial time.

To show that it is NP-*hard*, we will reduce from the INDEPENDENT SET problem, which is defined [**?**, p. 194] in the following manner:
INSTANCE: Graph $G = (V,E)$, positive integer $K \leq |V|$.
QUESTION: Does G contain an independent set of size K or more, i.e., a subset $V' \subseteq V$ such that $|V'| \geq K$ and such that no two vertices in V' are joined by an edge in E?

Given an instance $\langle G = (V,E), K \rangle$ of INDEPENDENT SET, we can reduce to an instance of determining whether a set T of IDK classifiers contains $F+1$

mutually exclusive ones as follows. We will have one IDK classifier I_v for each $v \in V$, and

$$\mathcal{E}(I_v) = \bigcup_{(v,w)\in E} \{I_w\}$$

It is evident that the set of $|V|$ IDK classifiers so obtained will have $K+1$ mutually exclusive ones if and only if the instance $\langle G = (V, E), K\rangle \in$ INDEPENDENT SET.

Runtime Complexity (of the Preprocessing Phase). The preprocessing phase essentially consists of initializing the lookup table with $3^{|\mathcal{K}|}$ rows and then calling EXPAND$(\emptyset, \emptyset)$. The call to EXPAND$(\emptyset, \emptyset)$ will make recursive calls to EXPAND(S, T) for different values of (S, T); however, memoization as implemented in Algorithm **??** (see Line **??**) ensures that each such recursive call is expanded at most once per state. Hence there are at most $3^{|\mathcal{K}|}$ calls to EXPAND( ). For each such call, the only non straight-line (i.e., constant-time) steps are:

- Line **??**: FTCOUNT(T, F). We saw in Section **??** that each such call has runtime complexity $\mathcal{O}\left(|\mathcal{K}|^{F+1} \cdot (F+1)^2\right)$
- Line **??**: CONDPR(S, T, K_j). We saw in Section **??** that each such call has runtime complexity $\mathcal{O}\left(2^{|\mathcal{K}|} \cdot |\mathcal{K}|\right)$.
- The for-loop at Lines **??**–**??**, which is executed $|\,(\mathcal{K} \setminus S)\,| = \mathcal{O}(|\mathcal{K}|)$ times.

Hence the total running time is

$$\begin{aligned}
&\mathcal{O}\left(3^{|\mathcal{K}|} \times \left(|\mathcal{K}|^{F+1} \cdot (F+1)^2 + 2^{|\mathcal{K}|} + |\mathcal{K}|\right)\right) \\
&= \mathcal{O}\left(3^{|\mathcal{K}|} \cdot |\mathcal{K}|^{F+1} \cdot (F+1)^2 + 6^{|\mathcal{K}|} + 3^{|\mathcal{K}|} \cdot |\mathcal{K}|\right) \\
&\approx \mathcal{O}\left(6^{|\mathcal{K}|}\right)
\end{aligned}$$

where the last step makes the reasonable assumption that $|\mathcal{K}|$ is significantly larger than F.

We contrast this to the state-of-the-art algorithms that are used for synthesizing static cascades, that are known [**?**, p. 369] to have a runtime bound of $\mathcal{O}\left(4^{|\mathcal{K}|}\right)$.

4 Evaluation

In this section, we present experiments evaluating two key aspects of our dynamic alternative to cascade-based fault-tolerant classification using IDK classifiers: *effectiveness* and *scalability*. By effectiveness, we refer to the reduction in expected execution duration compared to the state-of-the-art static approach from [**?**]. As discussed earlier, our preprocessing algorithm has a worst-case runtime complexity of $\mathcal{O}(6^{|\mathcal{K}|})$, compared to $\mathcal{O}(4^{|\mathcal{K}|})$ for the approach in [**?**]. Our scalability evaluation assesses the practical implications of this higher asymptotic cost by comparing how preprocessing times scale with increasing $|\mathcal{K}|$.

Our experiments used two kinds of data. First, we evaluated the two workloads considered in [?]—a hand-crafted illustrative instance and a real-world case study—reported in Section ??. Second, we developed a synthetic workload generator to explore a broader range of parameter settings, with results reported in Section ??.

All experiments were conducted on a 2022 MacBook Air equipped with an Apple M2 processor and 8 GB of RAM, running macOS Sequoia 15.0.1. The algorithms and synthetic workload generator were implemented in Python 3.11.5. Code was written with an emphasis on clarity and ease of development, rather than runtime efficiency.

4.1 Evaluating the Workloads From [?]

Workload I. The simple illustrative example introduced in [?, Sec. VI-B], consisting of four IDK classifiers and a deterministic classifier, was used to compare our dynamic approach with the cascade-synthesis algorithm from [?]. Execution times for the preprocessing phase were 36.7 ms for our method and 14.5 ms for the cascade synthesis—both negligible. We then computed the expected classification durations under fault-free conditions for various values of C_{det}; results are shown in Figure ??. As the table indicates, our dynamic approach consistently

C_{det}	Static Duration	Dynamic Duration	Ratio
500	212	209	0.99
1000	387	362	0.94
1500	562	512	0.91
2000	735	662	0.90
2500	885	812	0.92
3000	1035	962	0.93

Fig. 5: Expected execution durations for the example of [?, Sec. VI-B] (see Sec. ??).

outperforms the static cascade, with reductions in expected duration ranging from 1% to 10% depending on C_{det}.

Workload II. The applicability of the approach advocated in [?] was demonstrated upon a real-world, multi-modal case study in [?, Sec. VII].[6] We ran both our dynamic preprocessing algorithm and our implementation of the cascade-synthesis algorithm of [?] on this case study; the measured running times were 38.2 ms and 12.0 ms respectively – once again, both are fast enough for preprocessing time to not be a concern. We then computed the expected duration to

[6] As was explained in [?], this case study concerns the autonomous detection of potentially hostile enemy vehicles in a battlefield environment. Three different kinds of sensors were involved: acoustic (a microphone array), seismic (a vertical-axis geophone), and vision (a camera) – hence the term 'multi-modal.'

successful classification in fault-free scenarios for both algorithms; these turned out to be 7340.05 for the static cascade and 7261.07 for the dynamic approach: a savings of $(7340.05 - 7261.07)/7340.05$ or $\approx 1.1\%$. Hence for this case study the improvement in performance, while present, is not particularly large.

4.2 Synthetically Generated Workloads

We developed a synthetic workload generator, not with the aim of modeling real-world scenarios faithfully but rather to facilitate exploration of the space of possible system configurations by systematically varying several parameters. Our generator accepts the number $|\mathcal{K}|$ of IDK classifiers, the desired fault-tolerance level F, and several additional parameters:

- Two positive integers, *wcet_ max* and *wcet_ det_ ratio*, determine the WCET values. Each IDK classifier's WCET is drawn uniformly at random from $[1, wcet\_\ max]$, while the deterministic classifier's WCET is set to $C_{\text{det}} = wcet\_\ max \times wcet\_\ det\_\ ratio$.
- A real-valued parameter *excl_ prob* in $[0, 1]$ governs the generation of exclusivity pairs. Several generation modes are supported, including random (each pair of classifiers forms an exclusivity pair independently with probability *excl_ prob*, subject to symmetry) and bipartite (classifiers are partitioned into two sets, with exclusivity pairs spanning the partition).
- Probabilities over the 2^n Venn regions are generated using a Dirichlet distribution, with several options: uniform (equal concentration parameters), biased (higher concentration parameters for certain regions), and partitioned (different classifier subsets dominate under different input conditions).
- An optional deadline D may be specified; if omitted, D defaults to ∞.

We conducted an extensive set of experiments using workloads generated by this tool. Below, we present findings from our scalability study and summarize selected parameter exploration results aimed at evaluating the effectiveness of our proposed dynamic algorithm. (Due to space constraints, we omit a full report of the parameter space exploration.) **Scalability.** We measured the running times of our preprocessing algorithm across a wide range of randomly generated workloads. These running times were observed to depend primarily on the value of $|\mathcal{K}|$ (the number of IDK classifiers). The only other parameter that was found to significantly influence runtime is the deadline D: when D is very small, the preprocessing algorithm typically completes quickly, returning a strategy that either executes only the deterministic classifier K_{det} (if $D \geq C_{\text{det}}$ by a small margin) or reports infeasibility otherwise.

Figure **??** reports the running times of our preprocessing algorithm for various values of $|\mathcal{K}|$ with $D \equiv \infty$. Abdelzaher et al. observe [**?**, p. 356] that values of $|\mathcal{K}|$ "that are much greater than about 12 are unlikely to be commonly encountered in practice". In light of this, we note that the largest running time we observed—approximately 4598 seconds (or about 75 minutes) for $|\mathcal{K}| = 12$—is quite acceptable. This suggests that our prepossessing algorithm scales well for problem sizes likely to arise in practical applications.

$\|\mathcal{K}\|$	4	5	6	7	8	9	10	11	12
static	0.0056	0.0368	0.2349	0.7992	5.0043	33.6420	493.1335	707.9548	1506.1612
dynamic	0.0210	0.1248	0.9224	2.5149	17.4509	107.0504	2335.3348	2817.1647	4597.7350
ratio	3.7750	3.3895	3.9267	3.1466	3.4872	3.1820	4.7357	3.9793	3.0526

(a) Times (in seconds), and ratio

(b) Running times

(c) *Logarithm* of running times

Fig. 6: **Scalability** experiments: measured running times of the static and dynamic prepossessing algorithms for increasing values of $|\mathcal{K}|$.

Effect of F. We examined the impact of the desired degree of fault tolerance, F, on the expected duration to successful classification under fault-free behavior. As expected, this duration generally increases with larger values of F. What is more interesting is how the performance advantage of the dynamic strategy over the static one evolves with F.

For $F = 0$ (i.e., when no faults must be tolerated), the two strategies exhibit identical expected durations—this is anticipated, and indeed static cascades are most suitable when fault tolerance is not a requirement. On the other hand, for large values of F, the expected durations of the two strategies converge. Intuitively, the need to tolerate many faults makes it more efficient to execute the deterministic classifier K_{det} directly, bypassing the overhead of coordinating among IDK classifiers.

Figure **??** plots the ratio of the expected execution duration of the optimal static cascade to that of the dynamic strategy, as a function of F, using a set of example instances with $|\mathcal{K}| = 7$.[7] These instances were generated with uniformly distributed WCETs for the IDK classifiers, a *wcet_ det_ ratio* of 50, uniform probability distributions, and randomly generated exclusivity pairs with *excl_ prob* set to 0.05.

(a) The number F of faults that must be tolerated is changed.

(b) The probability of pairs forming exclusivity sets is changed.

(c) The hard deadline D is changed.

Fig. 7: **Effectiveness** experiments: variation of the ratio of expected execution durations of the optimal static cascade and our dynamic strategy as different workload parameters are varied. (All evaluations were on example instances with $|\mathcal{K}| = 7$; other parameters are varied as discussed in Section **??**).

[7] This experiment was conducted to explore how increasing F affects classification duration; it is not intended to model realistic scenarios—note that even in safety-critical systems, F is rarely set to a value greater than 2.

Effect of Exclusivity Sets. We also examined the impact that changing the likelihood of pairs of classifiers forming exclusivity pairs has on the expected duration to successful classification in fault-free behaviors – this was achieved by changing the value of the parameter *excl_prob* provided to our workload generator. The ratio of expected execution duration of the optimal static cascade to the dynamic strategy is plotted as a function of *excl_prob* in Figure **??**, for example instances with $|\mathcal{K}| = 7$ that were generated to have uniformly-distributed WCETs for the IDK classifiers, a *wcet_det_ratio* of 50, uniform probability values, and no deadline specified. As can be seen from Figure **??**, the performance improvement of the dynamic strategy over the static one becomes more marked with increasing likelihood of pairs of classifiers not being able to validate each others' classifications.

Effect of D. Towards the end of Section **??**, we briefly discuss how the improved performance of our dynamic strategy over static cascades may be further magnified if a hard deadline is specified within which classification must be completed. We have investigated this issue more systematically and methodically upon synthetically generated workloads; our observations are remarkably similar to the ones made with respect to the degree of fault tolerance. Specifically, the ratio of the expected execution duration of the optimal static cascade to the dynamic strategy

- is the same as that for the case when deadlines are not specified (and hence defaults to the case $D \equiv \infty$) for large values of $(D - C_{\text{det}})$;
- increases as $(D - C_{\text{det}})$ is decreased – this shows that the dynamic strategy's superiority increases; and
- decreases again as $(D - C_{\text{det}})$ becomes very small – this is a consequence of the fact that as $(D - C_{\text{det}}) \to 0$, the only fault-tolerant strategy is often to directly execute the deterministic classifier K_{det} (i.e., the static cascade $\langle K_{\text{det}} \rangle$ is also the optimal dynamic strategy).

An illustrative outcome is plotted in Figure **??**, which depicts the ratio of expected execution duration of the optimal static cascade to the dynamic strategy is plotted as a function of $(D - C_{\text{det}})$. (This plot is for example instances with $|\mathcal{K}| = 7$ that was generated by our workload generator to have uniformly-distributed WCETs for the IDK classifiers, a *wcet_det_ratio* of 50, and uniform probability values.)

5 Conclusions

We have shown in this paper that when fault tolerance is desired, restricting oneself to only using pre-synthesized cascades has an adverse effect on the expected duration to successful classification. Hence we propose a dynamic alternative that adaptively decides at runtime which classifier to execute next based on the outcome of prior classifier executions. We derive pre-processing and runtime algorithms for implementing this adaptive approach, and demonstrate that these algorithms have acceptable running times that are comparable to those of the

corresponding algorithms for static cascades, and achieve smaller expected duration to successful classification than the prior static approaches were able to achieve.

The ideas of exclusivity pairs and exclusivity sets, introduced in [**?**] and used here to specify the requirement that multiple classifiers return the same class subject to a constraint over which classifiers to use, can also be applied to other classification problems. For example, the requirement for a collection of diverse classifiers to return different classes when applied to the same input can be addressed in a similar manner. Here diversity would replace exclusivity. And hence sampling via a diverse set of classifiers will deliver a broader spectrum of samples (in a shorter period of time). This application will be investigated further as part of future work.

In future work we also plan to investigate adaptive strategies for exploiting the differences between classifiers' WCETs (the C_i parameters) and their actual execution times, and extend our pre-processing and runtime algorithms to exploit these differences in order to achieve lower expected duration to successful classification in both fault-free and fault-tolerant settings.

Acknowledgments. Supported in part by the US National Science Foundation – NSF CNS-2141256, CNS-2229290 (CPS), and CNS-2502855 (CPS). This research was also funded in part by Innovate UK SCHEME project (10065634). EPSRC Research Data Management: No new primary data was created during this study.