Minimizing Execution Duration in the Presence of Learning-Enabled Components

Kunal Agrawal
kunal@wustl.edu

Sanjoy Baruah
baruah@wustl.edu

Alan Burns
alan.burns@york.ac.uk

Abhishek Singh
abhishek.s@wustl.edu

***Abstract*—Autonomous systems are increasingly using components that incorporate machine learning and other AI-based techniques in order to achieve improved performance. We address the problem of assuring correctness in safety-critical systems that use such components. We investigate an approach which formulates the problem as one in which performance is an objective function to be optimized while safety is a hard constraint that must be satisfied. We then apply heuristics and algorithmic techniques from optimization theory in order to solve the resulting constrained optimization problem.**

***Index Terms*—Learning-enabled components (LECs); Safety-critical systems; Performance optimization; Run-time monitoring; Typical analysis.**

I. Introduction

Many autonomous cyber-physical systems (CPS's), including unmanned aerial vehicles, self-driving cars, and unmanned underwater vehicles, are safety-critical. The safety of the run-time behavior of such safety-critical systems must be assured before they can be considered for deployment. However it is challenging to directly apply traditional approaches towards safety assurance to modern autonomous CPS's due to multiple reasons, including the presence of complex and adaptive functionalities depending upon machine learning techniques that are not well understood in the way that components traditionally used in safety-critical systems are. The importance and the immense complexity of the problem of obtaining assurance for autonomous CPS's that incorporate machine learning has been widely recognized, and approaches for solving this problem are being actively sought. For example, the *Assured Autonomy* Program [1] of the United States Defense Advanced Research Projects Agency (DARPA) has a goal of creating technology for establishing assurance of CPS's that contain "***Learning-Enabled Components"*** (**LECs**), which are an abstraction defined in [1] that generalizes a wide variety of popular machine learning approaches. In a similar vein, the *Assuring Autonomy International Programme* [2] is an initiative funded by the international insurance company Lloyd's of London at the University of York (UK), in response to a 2016 study by Lloyd's that identified assurance and regulation as the biggest obstacles to gaining the benefits of robotics and autonomy. Yet another important example is the *Bounded Behavior Assurance* initiative [3]spearheaded by the major US defense contractor Northrop Grumman Corporation, which seeks to define processes for establishing assurance (and eventually, obtaining certification) that the behavior of unmanned aerial vehicles that use machine learning to make safety-critical and mission-critical decisions will always remain within pre-specified bounds.

It has been observed [4] that *predictability* of run-time behavior is key to assuring safety in safety-critical systems. Although most non-trivial safety-critical systems inevitably encounter some unpredictability in run-time behavior, safety-critical systems designers have developed techniques for dealing with inherent run-time unpredictability with regards to extra-functional properties such as timing (the duration required to complete execution) or energy consumption. However, safety-critical systems that make use of LECs tend to additionally not be predictable from the functional perspective: the precise "worth" or value of a computation performed by an LEC that incorporates deep learning or similar AI-based techniques is often not easily predicted beforehand. How should one deal with such functional non-predictability in safety-critical systems? In this paper, we propose a possible approach towards doing so for a particular form of computation involving LECs, that possess the following characteristics:

- The overall computation can be looked upon as a multi-stage one, in which a series of functional blocks are to be executed in a specified sequence. For an execution of the computation to be considered *correct* (and hence safe), a specified minimum level of service must be obtained over all the stages; we assume that this minimum level of service is quantified as a numerical target value.
- We have a choice of different alternative implementations for each stage of the computation, some or all of which may involve the use of LECs. Each implementation takes some *duration* to complete execution, and achieves an associated *value* – a quantitative measure of the quality of the computation that was achieved by executing that implementation.[1] We perform the complete end-to-end computation by selecting and executing exactly one of the implementation choices for each stage, in sequence. The total value obtained by the end-to-end computation is defined to be the sum of the values associated with the implementations that were

This research presented was supported, in part, by the National Science Foundation (USA) under Grant Numbers CNS-1814739, CPS-1932530, and CNS-1911460, and the EPSRC (UK) funded project Strata.

[1]It may be convenient to think of this value as a measure of the progress that will be made towards achieving the overall objective for the computation, if this implementation were selected for this stage of the computation.

selected for the individual stages.
- We can monitor the computation — determine certain aspects of system state — after each stage during run-time.

For computations possessing these properties we consider different approaches for scheduling the computation that can **guarantee safety** — i.e., guarantee that the computation will *achieve the specified target value of quality of service* — and **optimize for performance** – specifically, *reduce the overall duration* of the computation. We provide a precise formulation of the scheduling problem that needs to be solved as a constrained optimization problem (Section II); explore a number of algorithms, ranging from simple heuristics that are efficiently implementable to provably optimal ones, for solving this problem (Section III); and compare these different algorithms via simulation experiments on randomly-generated synthetic workloads (Section V).

The model of computation that we are considering in this paper is rather restrictive: several reasonable generalizations (a few of which are discussed in Section IV and others are enumerated in Section VI) could be thought of. We plan to explore these generalizations in future work – we look upon the results in the current paper as an initial step towards developing a body of results that will prove capable of dealing with a far more general model of LEC-based computations than the simple one we consider here.

II. Model and Problem Statement

As discussed above, we consider multi-stage computations in which a series of functional blocks are to be executed in a specified sequence, and we have a choice of several different implementations for each stage. Let n denote the number of stages, and m the maximum number of available alternative implementations for any stage. (An example multi-stage computation with n and m both equal to 2 is depicted in Figure 1.). Let $\mathcal{V} \in \mathbb{N}$ denote a target value that must be obtained cumulatively across all stages of the computation. We will use the notation $\mathcal{I}_{i,j}$ to denote the j'th implementation choice for the i'th stage, $0 \leq i < n$ and $0 \leq j < m$. Let $V_{i,j} \in \mathbb{N}$ denote the value that is obtained by executing the implementation $\mathcal{I}_{i,j}$, and let $C_{i,j} \in \mathbb{N}$ denote the duration of this execution — we do not assume that the numerical value of these parameters are known prior to executing $\mathcal{I}_{i,j}$ (and indeed allow for the possibility that they may be different on different executions of $\mathcal{I}_{i,j}$). Consider some execution of the end-to-end computation, and let $\phi(i)$ denote the implementation of the i'th stage that is chosen (i.e., $\mathcal{I}_{i,\phi(i)}$ is the executed implementation) for each i, $0 \leq i < n$. (Note that the function $\phi(\cdot)$ thus specifies the schedule for the computation.) It is required that this function $\phi(\cdot)$ satisfy the constraint that $\sum_i V_{i,\phi(i)} \geq \mathcal{V}$; from amongst all such ϕ, we seek the one that minimizes $\sum_i C_{i,\phi(i)}$. That is, our correctness constraint is that the sum of the values returned across all n stages should equal (or exceed) the specified threshold value $\mathcal{V}$, while the performance objective is that the cumulative duration of the computation be minimized.

As stated above, the C_{ij}, V_{ij} values are unknown prior to actually executing $\mathcal{I}_{i,j}$, and will in general take on different values each time $\mathcal{I}_{i,j}$ is executed. In order to be able to do pre-run-time verification, it is necessary that *worst-case bounds* be known on the values that these quantities may take. Let $c_{i,j}$ and $v_{i,j}$ denote safe worst-case bounds on the values of $C_{i,j}$ and $V_{i,j}$ respectively, that can be determined beforehand; by "safe," we mean that it is guaranteed that $C_{i,j} \leq c_{i,j}$ and $V_{i,j} \geq v_{i,j}$ for all executions of $\mathcal{I}_{i,j}$.

- The value of $c_{i,j}$ is what is commonly referred to in the real-time computing literature as the *worst-case execution time* (WCET) of the implementation $\mathcal{I}_{i,j}$, and may be determined using the wide range of tools, techniques, and methodologies for WCET-determination [5] that have been developed within the real-time computing community.
- We require that similar tools, techniques, and methodologies be developed that enable us to determine lower bounds on the value of the computation that is performed by an LEC. While we recognize that this is a major "ask" that will require a large concerted effort on the part of the safety-critical systems community, we believe it is unavoidable — we don't really see any other path to enabling the safe and effective use of LECs in safety-critical systems.

If we are to be able to verify correctness of a given computation prior to run-time, it is evident that there should exist some implementation of each stage such that the worst-case value bounds of these implementations sum to at least the target value – this correctness requirement is formalized in Section III as a *feasibility test*, and computations passing the feasibility test are said to be *feasible*. If a computation is deemed feasible, our approach, as briefly described in Section I, will generate a schedule prior to run-time that can be verified for correctness, and shown to have an acceptably small duration. What properties must such a schedule satisfy? Recall that the function $\phi : [0, \ldots, n-1] \rightarrow [0, \ldots, m-1]$ specifies the schedule for the computation – i.e., which implementation of each stage is selected for execution. One possibility for the initial schedule would be to choose $\phi(\cdot)$ to minimize the quantity $\left(\sum_{i=0}^{n-1} c_{i,\phi(i)}\right)$, subject to the constraint that $\left(\sum_{i=0}^{n-1} v_{i,\phi(i)} \geq \mathcal{V}\right)$. Such a schedule guarantees to have the optimal (i.e., smallest) duration bound in the worst case; in the absence of additional information about run-time behavior, this is a reasonable initial schedule to choose to work with. However, it may be the case that additional information regarding run-time behavior is available prior to run-time (in addition to the worst-case bounds $c_{i,j}$ and $v_{i,j}$); if so, it may be possible to use such additional information in order to further optimize the initial schedule *provided* we are able to do so without compromising the correctness guarantee whatsoever. An example of such additional information that may be available, that we believe may be particularly interesting and useful, is suggested by Quinton et al. [6] via the concept of **typical analysis**. The idea behind typical analysis is that while a worst-case characterization of a system must encompass all possible behaviors of the system, a "typical" characterization excludes

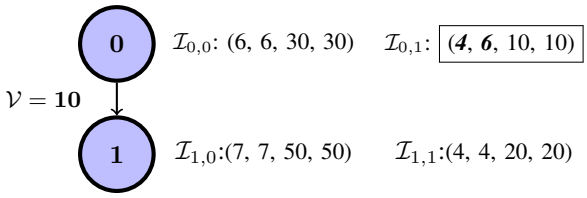

Fig. 1: An example instance: a 2-stage computation ($n = 2$), with two possible implementations per stage ($m = 2$), that must achieve a value of at least 10 ($\mathcal{V} = 10$). The 4-tuples represent the $(v_{i,j}, v^T_{i,j}, c_{i,j}, c^T_{i,j})$ parameters of the implementations.

pathological behaviors that are extremely unlikely to occur in practice.[2] Let us suppose that our multi-stage computation is subjected to such typical-case analysis, and let parameters $c^T_{i,j}$ and $v^T_{i,j}$ denote the typical execution duration of, and the typical value obtained by, the implementation $\mathcal{I}_{i,j}$; the interpretation of these parameters being that implementation $\mathcal{I}_{i,j}$ will complete execution within a duration no greater than $c^T_{i,j}$ ($C_{i,j} \leq c^T_{i,j}$) and obtain a value no smaller than $v^T_{i,j}$ ($V_{i,j} \geq v^T_{i,j}$) in all non-pathological executions of the computation.

Problem statement. We now summarize our workload model, and the problem we seek to solve. A problem *instance* is specified by specifying values for

- the number of stages n of the multi-stage computation;
- the maximum number of alternative implementations m for each stage;
- the target value $\mathcal{V}$ that is needed for correctness; and
- the worst-case and typical values $c_{i,j}$ and $c^T_{i,j}$ for the execution-duration and $v_{i,j}$ and $v^T_{i,j}$ for the value-obtained parameters, for each implementation $\mathcal{I}_{i,j}$, $0 \leq i < n, 0 \leq j < m$.

Given an instance specified in this manner, we will consider how to schedule it in order to guarantee correctness under all circumstances (assuming this is possible – i.e., the instance is *feasible*) while optimizing for duration. We will consider two different objectives with regards to this optimization criterion

1) minimize duration under all circumstances; and
2) minimize duration under all non-pathological conditions only (while continuing to guaranteeing correctness under all circumstances, including atypical, pathological ones).

We illustrate some of these concepts (in particular, the difference between the two optimization criteria) via an example:

Example 1: Consider a 2-stage computation ($n = 2$) with a choice of 2 implementations per stage ($m = 2$), for which correctness requires that a cumulative value of at least 10 be obtained ($\mathcal{V} = 10$) — see Figure 1. This example has been constructed to be particularly simple in order to highlight the difference between the two optimization criteria: the typical delay estimate for each of the four implementations is equal to its worst-case delay estimates, and the second implementation of the first stage, $\mathcal{I}_{0,1}$, is the only one of the four implementations for which its typical value estimate is different from its worst-case estimate (the specification of this implementation is enclosed within a box in the figure).

This instance is clearly *feasible*; e.g., executing the implementations $\mathcal{I}_{0,0}$ and $\mathcal{I}_{1,0}$ yields a value $\geq v_{0,0} + v_{1,0} = (6 + 7) = 13$, which is $\geq$ the target value $\mathcal{V}$ of 10.

If we seek to *optimize for the worst case*, then the initial schedule would choose the implementations $\mathcal{I}_{0,0}$ and $\mathcal{I}_{1,1}$ (equivalently, $\phi(0) \leftarrow 0$ and $\phi(1) \leftarrow 1$), for a cumulative value at least $v_{0,0} + v_{1,1} = (6 + 4) = 10$ (thereby assuring correctness), and a corresponding duration bound equal to $c_{0,0} + c_{1,1} = (30 + 20)$ or $\mathbf{50}$.

If, however, we seek to *optimize for the typical case*, then the initial schedule would choose the implementations $\mathcal{I}_{0,1}$ and $\mathcal{I}_{1,0}$ (equivalently, $\phi(0) \leftarrow 1$ and $\phi(1) \leftarrow 0$). This guarantees a cumulative value at least $v_{0,1} + v_{1,0} = (4+7) = 11$, thereby assuring correctness.

- During a typical execution of $\mathcal{I}_{0,1}$, we would expect to obtain a value $\geq v^T_{0,1}$ or 6. This will be determined by the run-time monitor, which will thus know that the remaining value that needs to be obtained is at most $(\mathcal{V} - 6)$, or 4. But this value can also be guaranteed by implementation $\mathcal{I}_{1,1}$ (because $v_{1,1} = 4$); since $\mathcal{I}_{1,1}$ has a smaller execution duration than $\mathcal{I}_{1,0}$, the run-time monitor will modify the remainder of the schedule by changing $\phi(1)$ to equal 1. Implementation $\mathcal{I}_{1,1}$ is therefore executed next, for a duration bound equal to $(10 + 20)$ or $\mathbf{30}$. Hence under typical circumstances the duration bound of 30 (rather than the 50 that was obtained by optimizing for worst-case behavior).
- Even during non-typical (pathological) executions, $\mathcal{I}_{0,1}$ guarantees a value of at least 4. If the value obtained is determined to be smaller than 6 by the run-time monitor, then implementation $\mathcal{I}_{1,0}$ is executed next as per the initial schedule (that was constructed prior to run-time); the resulting duration bound is then equal to $(10 + 50)$ or $\mathbf{60}$.

■

As illustrated in the example above, optimizing for typical-case behavior may yield very different schedules than optimizing for worst-case behavior. Which is more appropriate to use? We argue that there is no single good answer here: different applications and different use-cases may favor one over the other. Optimizing for typical-case behavior results in better average performance (assuming that the typical cases are accurately characterized), which optimizing for worst-case behavior ensures smaller durations under all circumstances. We therefore consider below, a number of algorithms for scheduling under both optimization metrics; we reiterate that the algorithms for both metrics guarantee correctness under

[2] E.g., worst-case characterization of the value obtained by an implementation may be obtained by performing static analysis of the implementation, making worst-case (or pessimistic) assumptions and rigorously proving the value that will be obtained under these assumptions. In contrast, a typical characterization of this value may be obtained via extensive experimentation and measurement, executing the implementation under a wide range of "typical" conditions and using the smallest measured value that is obtained.

all circumstances – even if duration is being optimized for the typical case, correctness guarantees remain worst-case ones.

III. Algorithms for Analysis and Scheduling

We now propose several different approaches for solving the scheduling optimization problem that was formalized in Section II above; all involve deciding at each stage which of the available implementations to choose. These approaches may be characterized along four orthogonal axes:

1) LOCAL/ GLOBAL: is the choice at a stage made based only on information available at that stage, or are the static attributes of future stages taken into account?
2) STATIC/ DYNAMIC: is the schedule that must be synthesized prior to run-time and verified for safety, subject to modification based on information obtained via run-time monitoring?
3) OPTIMAL/ HEURISTIC: We will both consider approaches for which precise notions of optimality can be proved, and ones based on what appear to be intuitively reasonable heuristics.
4) WORST-CASE/ TYPICAL: as discussed above, the problem framework is open to a worst-case or typical-case focus.

Recall that our correctness requirement mandates that we obtain a cumulative value $\geq \mathcal{V}$ across all the stages of our multi-stage computation. Hence a *feasibility test* would check that the largest worst-case value obtainable at each stage, summed across all the stages, is $\geq \mathcal{V}$: $\sum_{0 \leq i < n} \max_{0 \leq j < m} \{v_{i,j}\} \geq \mathcal{V}$. As the computation progresses, in order to retain feasibility it is necessary that the actual cumulative value obtained thus far, plus the largest values obtainable from the remaining stages, remains $\geq \mathcal{V}$.

From this consideration of feasibility and the characteristics outlined above it is possible to define a number of potential scheduling approaches; Figure 2 enumerates a (non-exhaustive) list of potential candidates. The first three are based upon reasonable heuristics that are very efficiently implementable; the last two are obtained by applying standard dynamic programming techniques that translate into pseudo-polynomial time algorithms,[3] to obtain optimal solutions.

IV. Adding State and Modes

As stated in Section I, the model of computation considered in this paper is rather restrictive; analysis of this simple model should be considered a (necessary) first step towards enabling the safe use of LECs in safety-critical systems. We now describe two generalizations that we believe would enhance its applicability; we have some preliminary ideas as to how to extend our analysis techniques to deal with these extensions, that we propose to study as future research.

State. Since we are modeling multi-stage computations, it is likely that some *state* is generated by a stage and communicated to subsequent stages, in the sense that (some of) this state will influence the behavior of the available implementations of subsequent stages. For instance, the typical value that is obtained by an implementation may depend upon the progress achieved by the implementations chosen. at previous stages. Consider for example a stage of an image progressing algorithm tasked with determining how many people there are in an image. The next stage may consist of classifiers, some of which are sensitive to this number. Knowing the value achieved at the previous stage is one method of capturing influence, but in general it is likely that further state information will be required.

The introduction of value-influencing state does not effect the framework developed in this paper. We retain the notions of worst-case value and duration, and hence retain the same definition of feasibility. However, the optimization problem becomes more difficult if there is a significant quantity of state with this influencing role; there may be more typical values to accommodate.

Modes. Some implementations may have more than one mode of operation: they offer a number of "*(value, computation-time)*" profiles, that are mutually incomparable. If the number of such modes is small then this is essentially equivalent to having more actual implementations (that happen to share the same worst-case behavior). However if the number of modes is high, or any one of a continuum of profiles is possible (as is the case with some anytime algorithms), then it is not immediately evident whether our proposed algorithms would scale appropriately with the number of modes that need to be considered.

As future work we will attempt to classify the problem space into domains that are amenable to optimal solutions and those that will need to fall back on the use of heuristics. (Note that in the evaluation section that follows the systems under evaluation simple ones without state or modes.)

V. Experimental Evaluation

We conducted our experimental evaluation upon two synthetically-generated sets of 100 multistage computation instances each – the two generation methods that we used are detailed below. Each instance was separately scheduled over

1) NAÏVE: At each stage, execute the implementation with the largest worst-case value ($v_{i,j}$). This strategy ensures a correct solution for any feasible instance, and will hence constitute the baseline strategy for our experimental evaluation.
2) LARGEST (v^T/c^T): At each stage execute the implementation with the largest typical "value density" from amongst those implementations that retain feasibility.
3) SMALLEST c^T: At each stage execute the implementation with the smallest typical execution duration from amongst those implementations that retain feasibility.
4) WORST-CASE: The initially-generated schedule seeks to optimize for worst-case behavior; modifications made to this schedule in response to run-time monitoring make use of worst-case parameter estimates for future stages.
5) TYPICAL: The initially-generated schedule, as well as modifications resulting from run-time monitoring, seek to optimize for typical-case behavior; the notion of typical-case optimality is analogous to that described in [7], [8].

Fig. 2: The scheduling strategies considered.

[3]Due to lack of space we will not describe these algorithms here, instead explaining what each seeks to optimize. They are compared with the simple heuristics in our experimental evaluation (Section V). We will detail these algorithms in an extended version of this paper currently under preparation.

500 simulated runs according to each of the five scheduling approaches described in Figure 2. For each instance and each strategy, we measured the cumulative computation duration over all 500 simulations. We quantified the normalized performance of each strategy S vis-à-vis NAÏVE as follows:

$$\texttt{perf}_S \stackrel{\text{def}}{=} \frac{\texttt{total duration of NAÏVE}}{\texttt{total duration of } S}$$

We now briefly[4] describe our two workload-generation methods. All generated instances have ten stages ($n = 10$) and a choice of five implementations per stage ($m = 5$). For every one of the fifty implementations $\mathcal{I}_{i,j}$ in each generated instance, we randomly select a sub-interval (v_{ij}^L, v_{ij}^H) of $(1, 100)$, and a sub-interval (c_{ij}^L, c_{ij}^H) of $(1, 1000)$ – the manner in which we do so for each of our two generation methods is discussed below. The interpretation is that the duration of each execution of $\mathcal{I}_{ij}$ will be $\geq c_{ij}^L$ and $\leq c_{ij}^H$, and the value obtained, $\geq v_{ij}^L$ and $\leq v_{ij}^H$; in simulation runs during our experiments we choose values from these intervals according to some probability distribution (as discussed below). We assign the worst-case parameters accordingly: $v_{ij} \leftarrow v_{ij}^L$, and $c_{ij} \leftarrow c_{ij}^H$. For our experiments, we assign the typical parameters v_{ij}^T and c_{ij}^T the expected value of the parameters (as discussed below).

Once all the parameters of all implementations of an instance have been assigned values, we need to assign a value to $\mathcal{V}$, the target value parameter. Here we introduce a configuration parameter $p \in \{10, 20, \ldots, 90\}$ denoting the percentage of the largest value that can be guaranteed, and set $\mathcal{V}$ as follows:

$$\mathcal{V} = (p/100) \times \Big(\sum_{0 \leq i < n} \max_{0 \leq j < m} \{v_{i,j}\} \Big)$$

Generation Method 1. For each implementation $\mathcal{I}_{ij}$, c_{ij}^L and c_{ij}^H are chosen uniformly over $(1, 1000)$, and v_{ij}^L and v_{ij}^H uniformly over $(1, 100)$. The actual durations taken and values obtained by executing $\mathcal{I}_{ij}$ during simulation runs are drawn from one of two distributions (in different experiments): one a symmetric distribution derived from the Normal distribution and the other, a skewed distribution derived from the Gamma distribution with shape parameter 2. Since we have a choice of two forms of distributions for the computation duration and for the value obtained, we have a total of 2×2 or four sub-configurations; we report on results for all four.

Generation Method 2. Instances are generated to have a positive correlation between the c_{ij}^T and v_{ij}^T values for each implementation. To do so, a value is assigned to v_{ij}^T according to the uniform distribution, and c_{ij}^T set equal to v_{ij}^T (appropriately scaled) plus some random noise. Next, values are assigned to v_{ij}^L and v_{ij}^H such that the $v_{ij}^T = (v_{ij}^L + v_{ij}^H)/2.0$, thereby ensuring that v_{ij}^T corresponds to the mean of our symmetric distribution (derived from the normal distribution). Values for c_{ij}^L and c_{ij}^H are assigned analogously. The actual durations taken and values obtained by executing $\mathcal{I}_{ij}$ during simulation runs are drawn from the symmetric distribution only; we do not use the skewed distribution in Generation Method 2.

Results. A subset of our results are graphically depicted in Figures 3 and 4. As previously stated, performance of each strategy is normalized against the performance of the baseline NAÏVE strategy; the scale of the vertical axes in these graphs represent this performance improvement. Each colored vertical box extends from the lower to the upper quartile of the data being depicted, with a horizontal black line at the median. The "whiskers" show the range of the data (with extreme outliers discarded – in plotting these graphs, the `whis` parameter[5] retained its default value of 1.5).

In each of the graphs in Figure 3, each group represents one of the four sub-configurations of Generation Method 1 that are obtained by choosing symmetric or skewed distributions for each of the two parameters, duration and value; individual columns within these groups represent the four individual strategies numbered (2)-(5) in Figure 2.

There is only one configuration for Generation Method 2, with both value and duration drawn from the symmetric distribution. Figure 4 depicts the performance improvement over NAÏVE for the different strategies, with each group representing a different choice for parameter p (and hence $\mathcal{V}$).

Discussion. In addition to providing strong evidence that all the strategies considered appear to provide significant improvement (up to a factor of 10) over the baseline NAÏVE strategy, some obvious conclusions leap out from our experimental observations. First, we note that in all our experiments the simple local heuristic of choosing at each stage the implementation with smallest WCET that retains feasibility proves remarkably efficient: its performance closely tracks that of the optimal strategy TYPICAL, which requires pseudo-polynomial pre-processing and run-time monitoring. Next, we observe that skewing the distributions from which actual execution durations and obtained values are drawn results in all the algorithms showing improved performance vis-à-vis NAÏVE. This is consistent with expectations: the more skewed the distributions, the more benefit we would expect to obtain by adapting our choice of implementations based upon run-time monitoring observations. Third, we note that the performance improvements obtained when Generation Method II is used are significantly better than when Generation Method I is used – the y-axis in Figure 4 is labeled over the range $[0, 12]$ in contrast to the range $[0, 5]$ of Figure 3. This, too, is intuitively appealing: correlated values of WCET and value obtained imply that choosing an implementation that returns greater value takes a greater duration, and hence making the right choice should yield greater benefit. LARGEST (v^T/c^T) is the only under-performer because it is uniquely ill-suited for multi-stage computations generated under the correlation hypothesis, since v^T/c^T is close to constant.

[4]The 6-page limit prevents us from providing further detail here; we will make fully annotated versions of our workload-generation code available online subsequent to anonymous peer-review.

[5]See https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.boxplot.html

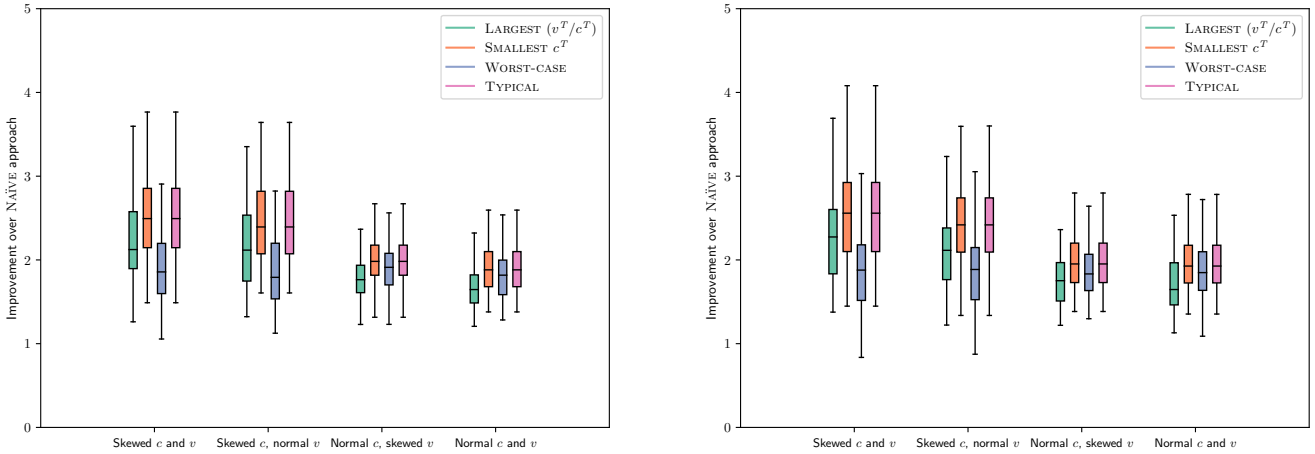

Fig. 3: Generation Method I: Comparing strategies at $p = 20$ (left figure) and $p = 80$ (right figure).

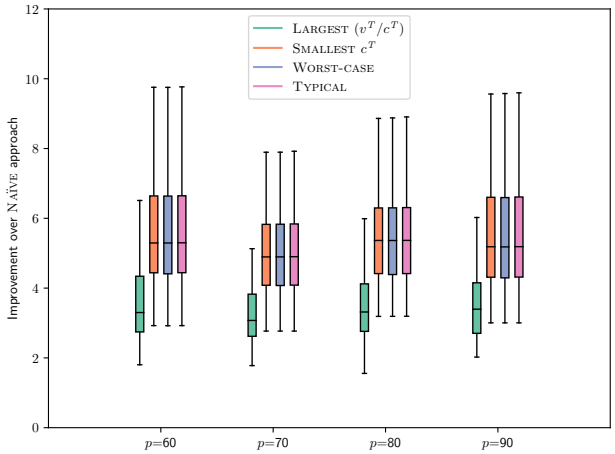

Fig. 4: Generation Method II: Comparing strategies at $p \in \{60, 70, 80, 90\}$.

VI. Conclusions

It appears inevitable that Learning-Enabled Components (LECs) based upon deep learning and similar AI-based principles will play an increasingly major role in safety-critical autonomous CPS's; it is therefore incumbent on the safety-critical systems research community to devise techniques for the analysis of such systems. This paper reports on our initial efforts in this direction. We have proposed a formal model for representing the behavior as well as the timing properties of some kinds of LECs. We have formulated the problem of synthesizing computations that can be modeled as chains of functional blocks using LECs and that need to achieve a minimum cumulative value to assure safety, and for which performance is quantified by the total duration of the computation, as an optimization problem. We have proposed several strategies, some heuristic and others provably optimal, for solving this optimization problem, and have compared these different strategies via simulation experiments upon synthetically generated workloads. As ongoing and future work we are evaluating, and will continue to evaluate, specific LECs (such as ones based on deep learning) to determine whether they are amenable to representation using our model and if not, how our model may be generalized to accommodate them (see, e.g., the discussion in Section IV that has come out of our efforts in this direction). We will further develop our workload generator and simulation platform, incorporating insights from our study of specific LECs, in order to be able to conduct more extensive experiments of the kind reported in this paper. We plan to further explore the simple heuristic "SMALLEST c^T" that seems to have performed very well in our experimental evaluation, to further study its properties and thereby understand the conditions and circumstances where its use is particularly warranted. From an algorithmic perspective, we plan to formally characterize notions of optimality for computations involving the use of LECs, and to develop algorithms for generating optimal schedules for these computations.

References

[1] Sandeep Neema. Assurance for Autonomous Systems is Hard. https://www.darpa.mil/attachments/AssuredAutonomyProposersDay_ProgramBrief.pdf. Accessed: 2019-03-07.

[2] Assuring autonomy international programme. https://www.york.ac.uk/assuring-autonomy/ Accessed: 2019-09-20.

[3] J. Lee, A. Prajogi, E. Rafalovsky, and P. Sarathy. Assuring behavior of autonomous UxV systems. In *S5: The Air Force Research Laboratory (AFRL) Safe and Secure Systems and Software Symposium*, July 2016.

[4] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Syst.*, 2(4):247–254, October 1990.

[5] R. Wilhelm et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

[6] Sophie Quinton, Matthias Hanke, and Rolf Ernst. Formal analysis of sporadic overload in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 515–520, San Jose, CA, USA, 2012. EDA Consortium.

[7] Sanjoy Baruah. Rapid routing with guaranteed delay bounds. In *Real-Time Systems Symposium (RTSS), 2018 IEEE*, Dec 2018.

[8] Kunal Agrawal and Sanjoy Baruah. Rapid routing in polynomial time. In *Real-Time Systems Symposium (RTSS), 2019 IEEE*, Dec 2019.