The Safe and Effective Use of Learning-Enabled Components in Safety-Critical Systems

Kunal Agrawal
Washington University in Saint Louis, MO, USA
kunal@wustl.edu

Sanjoy Baruah
Washington University in Saint Louis, MO, USA
baruah@wustl.edu

Alan Burns
The University of York, UK
alan.burns@york.ac.uk

Abstract

Autonomous systems increasingly use components that incorporate machine learning and other AI-based techniques in order to achieve improved performance. The problem of assuring correctness in safety-critical systems that use such components is considered. A model is proposed in which components are characterized according to both their worst-case and their typical behaviors; it is argued that while safety must be assured under all circumstances, it is reasonable to be concerned with providing a high degree of performance for typical behaviors only. The problem of assuring safety while providing such improved performance is formulated as an optimization problem in which performance under typical circumstances is the objective function to be optimized while safety is a hard constraint that must be satisfied. Algorithmic techniques are applied to derive an optimal solution to this optimization problem. This optimal solution is compared with an alternative approach that optimizes for performance under worst-case conditions, as well as some common-sense heuristics, via simulation experiments on synthetically-generated workloads.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Computing methodologies → Machine learning; Software and its engineering → Real-time schedulability

Keywords and phrases Learning-enabled components (LECs), Safety-critical systems, Typical analysis, Performance optimization, Run-time monitoring

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2020.7

Funding *Kunal Agrawal*: National Science Foundation Grants CCF-1733873 and CCF-1439062.
Sanjoy Baruah: National Science Foundation Grants CNS-1814739 and CPS-1932530.
Alan Burns: EPSRC (UK) grants EP/K011626/1, EP/P003664/1, and EP/N023641/1.

1 Introduction

Many autonomous cyber-physical systems (CPS's) such as self-driving cars are safety-critical, and must have their safety properties verified before they can be considered for deployment. However approaches that have traditionally been used for the purposes of performing safety assurance in safety-critical systems do not seem directly applicable to modern autonomous CPS's due to multiple reasons, including the presence of complex and adaptive functionalities that are based upon the incorporation of machine learning techniques that are not well understood in the way that components traditionally used in safety-critical systems are. The importance, as well as the enormous 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 – consider the following example initiatives:

- The *Assured Autonomy* Program [8] 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 [8] that generalizes a wide variety of popular machine learning approaches.
- In a similar vein, the *Assuring Autonomy International Programme* [1] is an initiative funded by the international insurance company Lloyd's of London at the University of York (UK), motivated in part by a 2016 study by Lloyd's identifying assurance and regulation as being amongst the biggest obstacles to realising the benefits of robotics and autonomy.
- Yet another important example project in this space is the *Bounded Behavior Assurance* initiative [6], 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 is widely accepted that *predictability* of run-time behavior [10] is very important for the purposes of 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 a range of advanced and sophisticated 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. This aspect of run-time unpredictability has not been as widely studied in the safety-critical systems community: How should one deal with such functional uncertainty in safety-critical systems? In this paper, we continue our investigations, first reported in [3], of one possible approach for doing so for a particular form of computation involving LECs, that possess the following characteristics:

- The overall computation can be considered to be 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 cumulatively 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 selected for the individual stages.
- We can monitor the computation – determine certain aspects of system state – after each stage during run-time.

[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.

For computations that can be characterized in this manner. we consider different approaches for scheduling the computation in ways that **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 2); explore a number of algorithms, ranging from provably optimal ones (Section 3) to simple heuristics that are efficiently implementable (Section 4), for solving this problem; and compare these different algorithms via simulation experiments on randomly-generated synthetic workloads (Section 4).

Organization. The reminder of this paper is organized in the following manner. In Section 2 we formally specify the problem that is studied in this paper, and present arguments making the case that this is indeed a relevant problem that merits further exploration by the real-time computing community. In Section 3 we present algorithms for solving this problem, and derive properties that demonstrate the correctness and characterize the efficacy of our algorithms. In Section 4 we explore some heuristic solution techniques: although these heuristics are easily shown to be non-optimal, some simulation-based experiments on synthetically generated workloads indicates that carefully chosen heuristics may be adequate for solving some specific classes of the problem in a near-optimal manner. We discuss some directions in which the problem we are studying here could be generalized in Section 5, and conclude in Section 6 by placing this work within a larger persepctive on the design and analysis of highly complex safety-critical cyber-physical systems.

2 Model and Problem Statement

In this section we motivate and formally define the model of LEC-enabled computations that we are considering in this paper. We highlight, via illustrative examples, some of the challenges that arise in the pre-runtime safety analysis of such systems that we seek to solve, and formally define the problem that is studied in the remaining sections of this paper.

As discussed in Section 1 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 $n = 2$ and $m = 3$ 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 values 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 – we will henceforth often refer to it as *the scheduling function*, or simply *the schedule*, for the computation. It is required that the scheduling function $\phi(\cdot)$ satisfy the constraint that $\sum_i V_{i,\phi(i)} \geq \mathcal{V}$; from amongst all the functions ϕ that do so, we seek 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.

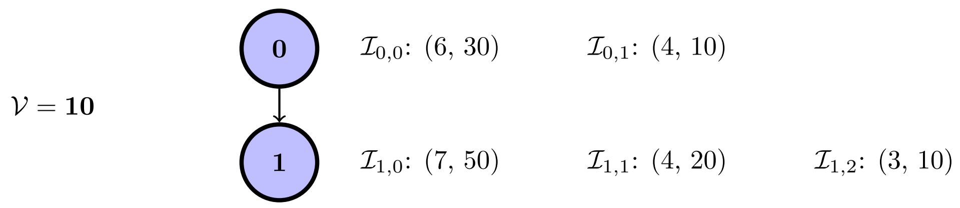
$\mathcal{V} = \mathbf{10}$
0
$\mathcal{I}_{0,0}$: (6, 30)
$\mathcal{I}_{0,1}$: (4, 10)
1
$\mathcal{I}_{1,0}$: (7, 50)
$\mathcal{I}_{1,1}$: (4, 20)
$\mathcal{I}_{1,2}$: (3, 10)

Figure 1 An instance discussed in Example 1: a 2-stage computation with a choice of two possible implementations for the first stage and three for the second, that must achieve a value of at least 10 ($\mathcal{V} = 10$). The ordered pairs represent the $(v_{i,j}, c_{i,j})$ parameters of the implementations.

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 $C_{i,j} \le c_{i,j}$ and $V_{i,j} \ge v_{i,j}$ 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 [11] that have been developed within the real-time computing community.[2]
- 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 later (in Expression 1) 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 1, will generate a schedule prior to run-time that can be verified for correctness, and shown to always have an acceptably small duration. What properties must such a schedule satisfy? Let us try to understand some of the issues involved via a simple 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. (As explained in the caption to the figure, each implementation is labeled here with an ordered pair denoting the minimum value that is

[2] **Note:** in much of the remainder of this paper we will make the simplifying assumption that the actual run-time of implementations does not vary much from their specified WCET's: $C_{i,j} \approx c_{i,j}$ for all (i, j). This simplifying assumption allows us to highlight the primary focus of this paper, which is that of dealing with the uncertainty that is inherent in a priori characterizing the *value* obtained from many LEC's. (This is also a reasonable assumption for the many Deep-Learning based LECs that are implemented as a known number of "layers" of matrix computations and therefore known to have very predictable and deterministic running times.) In Section 5 we briefly discuss how our work may be generalized by removing this simplifying assumption, and incorporating considerations of uncertainty along both the value and the timing dimensions.

guaranteed to be obtained by choosing to execute the implementation, and the maximum duration that the implementation may take to execute.) We note that since we have a choice of two implementations for the first stage and a choice of three implementations for the second stage, there is a total of $2 \times 3 = \mathbf{6}$ possible distinct schedules:

No.	Schedule	Min. cumulative value	Guaranteed Correct?	Max. delay
1.	$\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,0} \rangle$	$6 + 7 = \mathbf{13}$	**Y**	$30 + 50 = \mathbf{80}$
2.	$\boxed{\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle}$	$6 + 4 = \mathbf{10}$	**Y**	$\boxed{30 + 20 = \mathbf{50}}$
3.	$\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,2} \rangle$	$6 + 3 = \mathbf{9}$	N	—
4.	$\langle \mathcal{I}_{0,1}; \mathcal{I}_{1,0} \rangle$	$4 + 7 = \mathbf{11}$	**Y**	$10 + 50 = \mathbf{60}$
5.	$\langle \mathcal{I}_{0,1}; \mathcal{I}_{1,1} \rangle$	$4 + 4 = \mathbf{8}$	N	—
6.	$\langle \mathcal{I}_{0,1}; \mathcal{I}_{1,2} \rangle$	$4 + 3 = \mathbf{7}$	N	—

Of these six, three can assure a value of at least 10 are are thus guaranteed correct; from amongst these correct schedules, the schedule $\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$ has the lowest associated delay bound of 50.

If a ***static*** schedule –one in which the choice of implementations is finalized prior to run-time– is desired then it is reasonable to choose the schedule $\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$ since it is the correct schedule that guarantees the best (smallest) maximum delay. Such a static schedule has the benefit of not requiring any run-time monitoring: under the assumption that the worst-case characterizations (of both the value obtained and the duration required) of the implementations is correct, this schedule is guaranteed to obtain the required value of $\mathcal{V} = 10$ and is hence correct; additionally, from among all such static schedules the selected one is, informally speaking, clearly the "best" choice.

Suppose now that the schedules are permitted to be ***adaptive***: they may be changed between stages in response to additional information that is revealed during run-time by monitoring the actual behavior (recall that the c_{ij} and v_{ij} values represent conservative worst-case estimates: the actual behavior experienced during run-time may well turn out to be superior to these worst-case estimates). Consider the following possible scenarios.

1. If while proceeding to execute the chosen schedule, $\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$, upon completing the first stage it is discovered that this stage actually returns a value ≥ 7. Now since the value remaining to be acquired is ≤ 3, it is safe to switch to the implementation $\mathcal{I}_{1,2}$ for the second stage; doing so further reduces the maximum delay bound to $30 + 10 = \mathbf{40}$.
2. If we had instead initially chosen the schedule $\langle \mathcal{I}_{0,1}; \mathcal{I}_{1,0} \rangle$ (which is also guaranteed to be correct, albeit with a larger maximum delay bound of 60) and upon executing the first stage discovered that the value returned is ≥ 6, it then becomes safe to switch to the implementation $\mathcal{I}_{1,1}$ for the second stage, and doing so results in a maximum delay bound of $10 + 20 = \mathbf{30}$.

In the second scenario above, we started out with a sub-optimal schedule (from the static perspective), but run-time monitoring enabled the adaptive schedule to achieve a smaller delay than was achieved in the first scenario by starting out with the optimal static schedule. So when adaptive scheduling is permitted, which schedule should we start out with? In the absence of additional information (for instance, in our example above how *likely* is it that the value returned by $\mathcal{I}_{0,1}$ will actually exceed 6?), we cannot think of any reason to go with an initial schedule other than the one with the best worst-case static guarantee (in our example, the schedule $\langle \mathcal{I}_{0,0}; \mathcal{I}_{1,1} \rangle$). However, it may be the case that additional information (over and above the worst-case bounds $c_{i,j}$ and $v_{i,j}$) regarding run-time behavior is available prior to run-time; if so, it may be possible to use such additional information to

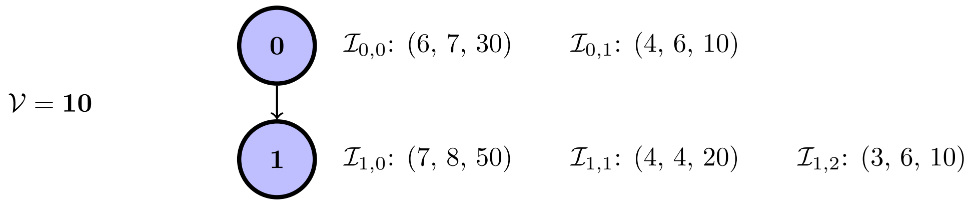
$\mathcal{V} = \mathbf{10}$
0
1
$\mathcal{I}_{0,0}$: (6, 7, 30)
$\mathcal{I}_{0,1}$: (4, 6, 10)
$\mathcal{I}_{1,0}$: (7, 8, 50)
$\mathcal{I}_{1,1}$: (4, 4, 20)
$\mathcal{I}_{1,2}$: (3, 6, 10)

Figure 2 The example instance of Figure 1, with the typical values obtained by implementations also specified. The 3-tuples pairs represent the $(v_{i,j}, v^T_{i,j}, c_{i,j})$ parameters of the implementations.

further optimize the initial schedule *provided* we are able to do so without compromising the correctness guarantee. An example of such additional information that may be available, that we consider to be particularly interesting and useful, is suggested by Quinton et al. [9] 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 pathological behaviors that are extremely unlikely to occur in practice.[3] Let us suppose that our multi-stage computation is subjected to such typical-case analysis, and let the parameter $v^T_{i,j}$ denote the typical value obtained by the implementation $\mathcal{I}_{i,j}$; the interpretation being that implementation $\mathcal{I}_{i,j}$ will 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.

With this notion of typical-case analysis in mind, let us revisit the scenarios introduced above. We can now state that if $v^T_{0,1} \geq 6$ (in Figure 2, the example instance of Figure 1 is updated with the typical value parameters, the $v^T_{i,j}$'s, also specified), then choosing implementation $\mathcal{I}_{0,1}$ rather than $\mathcal{I}_{0,0}$ for the first stage

1. Assures *correctness* under all circumstances, and
2. *offers superior performance* – a smaller maximum delay – under typical circumstances.

Problem statement. We now summarize our workload model, and the problem we seek to solve. A problem *instance* is defined 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;
- the worst-case and typical values $v_{i,j}$ and $v^T_{i,j}$ for the value-obtained parameters for each implementation $\mathcal{I}_{i,j}$, $0 \leq i < n, 0 \leq j < m$; and
- the worst-case execution time $c_{i,j}$ of implementation $\mathcal{I}_{i,j}$, $0 \leq i < n, 0 \leq j < m$. (As previously noted –see footnote 2– we could, in principle, have worst-case and typical-case characterizations of the execution time parameters as well; we have chosen to not do so here in order to keep things simple. We will revisit and briefly discuss the implications of this choice in Section 6.)

To summarize, an ***instance*** I of our problem is characterized by

1. the $n \times m$ implementations $\mathcal{I}_{i,j}$, $(i, j) \in \{0, 1, \ldots, (n-1)\} \times \{0, 1, \ldots, (m-1\}$, with each $\mathcal{I}_{i,j}$ characterized by the 3-tuple $(v_{i,j}, v^T_{i,j}, c_{i,j})$; and
2. the target value $\mathcal{V}$.

[3] 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.

A *schedule* for such an instance is a function $\phi : [0, 1, \ldots n-1] \rightarrow [0, 1, \ldots, m-1]$, with $\phi(i)$ denoting the implementation of stage i that is selected for execution in the schedule. An instance is said to be *feasible* if it is possible to schedule it in a manner that guarantees that the cumulative value $\mathcal{V}$ is obtained for all possible actual run-time behaviors; it is obvious that *a necessary and sufficient feasibility condition* is that

$$\left(\sum_{i=0}^{n-1} \max_{j=0}^{(m-1)} \{v_{i,j}\} \right) \geq \mathcal{V} \tag{1}$$

Given a feasible instance, we seek a scheduling strategy for choosing an implementation $\phi(i)$ for each stage i of the instance. We require that our strategy only make *safe* choices: never choose an implementation that could result in an incorrect schedule. If several safe choices are available, then our choice is governed by our optimization objective of minimizing execution duration; in this paper we advocate for choosing an implementation that *guarantees the smallest overall delay under all typical circumstances*, and derive algorithms that enable us to do so. In the following example we illustrate some of the issues that arise in choosing to optimize for the typical case rather than the worst case.

▶ **Example 2.** Let us revisit the example of Figure 2. We note that

- If we were to choose implementation $\mathcal{I}_{0,0}$ for the first stage we are guaranteed to obtain a value of at least 6 units, which would leave us requiring the remaining 4 units of value from the second stage. Either of the implementations $\mathcal{I}_{1,0}$ or $\mathcal{I}_{1,1}$ is able to guarantee this; hence it is safe to choose implementation $\mathcal{I}_{0,0}$ for the first stage.
- If we were to instead choose to execute implementation $\mathcal{I}_{0,1}$ for the first stage, then we are guaranteed to obtain a value of at least 4 units. This would leave us requiring 6 units of value from the second stage. Since implementation $\mathcal{I}_{1,0}$ is able to guarantee this, we conclude that it is safe to choose implementation $\mathcal{I}_{0,0}$ for the first stage.

We have seen that from the perspective of safety we may therefore choose either implementation $\mathcal{I}_{0,0}$ or implementation $\mathcal{I}_{0,1}$ for the first stage. Let us examine how we would choose between the two implementations.

Worst-case analysis. We separately consider both choices of implementation for the first stage:

1. If we choose $\mathcal{I}_{0,0}$, then the lowest value that we would obtain is 6, and will then need an additional 4 units of value from the second stage. That requires that we choose $\mathcal{I}_{1,0}$ or $\mathcal{I}_{1,1}$ for the second stage; we prefer $\mathcal{I}_{1,1}$ since it has a lower delay, and hence end up with a bound of $30 + 20 = \mathbf{50}$ on the delay.
2. If, on the other hand, we were to choose $\mathcal{I}_{0,1}$ for the first stage, then the lowest value that we would obtain is 4. That would leave us 6 units of value short of the target, and we must choose implementation $\mathcal{I}_{1,0}$ for the second stage. Thus results in a cumulative bound of $10 + 50 = \mathbf{60}$ on the delay under typical circumstances.

Since $50 < 60$, we would choose the implementation $\mathcal{I}_{0,0}$ for the first stage, and are guaranteed a maximum cumulative delay of 50 time units.

Typical-case analysis. We again consider both choices for the first stage:

- If we choose $\mathcal{I}_{0,0}$, then the lowest value that we would obtain under typical conditions is 7. Since we will then need an additional 3 units of value from the second stage, we may choose implementation $\mathcal{I}_{1,2}$ for the second stage, for a bound of $30 + 10 = \mathbf{40}$ on the delay under typical circumstances.

- If, on the other hand, we were to choose $\mathcal{I}_{0,1}$ for the first stage, then the lowest value that we would obtain under typical conditions is 6. That would leave us 4 units of value short of the target; since these 4 units of value must be guaranteed to be obtained in the second stage, we must choose implementation $\mathcal{I}_{1,1}$ for the second stage.[4] This results in a cumulative bound of $10 + 20 = \mathbf{30}$ on the delay under typical circumstances.

Since the delay bound is smaller if we choose $\mathcal{I}_{0,1}$, we would prefer the implementation $\mathcal{I}_{0,1}$ for the first stage. By so doing, we are able to guarantee a maximum cumulative delay of 30 time units under typical conditions (as opposed to the 50 time units that was guaranteed above, when we chose implementation $\mathcal{I}_{0,0}$ based on worst-case analysis). However if we were to encounter atypical conditions whilst executing implementation $\mathcal{I}_{0,1}$ and obtain less than $v^T_{0,1} = 6$ units of value, we may end up with a shortfall of more than four units of value and therefore need to execute implementation $\mathcal{I}_{1,0}$ for the second stage, for a cumulative delay of $10 + 50 = \mathbf{60}$ time units.

Example 2 above illustrates the difference in the kinds of performance guarantees that are made by "traditional" worst-case analysis, and the typical-case analysis that we are proposing:

> *Typical analysis provides superior performance* (in this example, smaller cumulative delay) *under typical conditions, but may provide worse performance under atypical conditions. However, it does guarantee correctness* (in our example, to obtain a cumulative value of at least 10) *under all conditions, typical or atypical.*

3 Scheduling Algorithms and Analysis

In this section we will develop algorithms that solve the problem described in Section 2 above. We derive two separate algorithms: one for solving the (preëxisting) problem of optimizing performance for the worst case and the other, for the novel problem that is the major focus of this paper, of optimizing performance for the typical case (while assuring correctness under all cases, typical or not). It turns out (see Section 3.1 below) that both these optimization problems are NP-hard; hence, we should not expect to be able to find polynomial-time algorithms for solving them. We will however show that both our algorithms have *pseudo*-polynomial running time.

The remainder of this section is organized in the following manner. We show that our problems are unlikely to be solvable using polynomial-time algorithms in Section 3.1, develop pseudo-polynomial time algorithms for solving them in Sections 3.2–3.3, and formally specify their optimality properties in Section 3.4.

3.1 Computational Complexity

The problem we seek to solve is easily seen to be computationally intractable – NP-hard, by reduction from the Multiple-choice Knapsack Problem (MCKP) [7], a well-known NP-hard problem. The Multiple-choice Knapsack Problem (MCKP) may be defined in the following manner:

> Given k *classes* $N_1 \ldots, N_k$ of *items*, and a *knapsack* of capacity c. Each item $j \in N_i$ has a *profit* p_{ij} and a *weight* w_{ij}, and the problem is to choose one item from each class such that the profit sum exceeds p without having the weight sum exceed w.

[4] We could also have chosen $\mathcal{I}_{1,0}$, but that has a larger delay bound, and hence offers inferior performance.

To determine whether a given instance of the form described above $\in$ MCKP, we could reduce it to an instance of our problem that comprises k stages with a choice of $|N_i|$ implementations for the i'th stage for each $i, 1 \leq i \leq k$. The implementation $\mathcal{I}_{i,j}$ is assigned the parameters $v_{ij} = p_{ij}$, $v_{ij}^T = p_{ij}$, and $c_{ij} = w_{ij}$; the desired target value $\mathcal{V} = p$. It is evident that this instance of our problem can guarantee a delay not exceeding w if and only if the given instance $\in$ MCKP. Since determining whether an instance $\in$ MCKP is NP-hard, it follows that determining a schedule that optimizes for performance under typical conditions, while concurrently assuring correctness, is also NP-hard.

Observe that the instance of our problem obtained in the above reduction from MCKP has $v_{ij} = v_{ij}^T$ for all (i,j); hence, its optimal solution would be exactly the same regardless of whether one were optimizing for performance under worst-case assumptions or typical-case ones. This establishes that the problem of optimizing for the worst case while assuring correctness is also an NP-hard problem.

3.2 Algorithm Description

As stated earlier, our objective is to obtain a schedule that optimizes for performance (i.e., minimizes the duration of computation) while assuring correctness – i.e., obtaining a cumulative value no smaller than the specified target $\mathcal{V}$. Our approach toward achieving this is to construct, prior to run-time, a *lookup table* that will subsequently be used during run-time in a manner that is elaborated upon in Section 3.3.

Let us suppose that we have completed execution of the first $(i-1)$ stages during some execution of the computation, having obtained a cumulative value $\widehat{\mathcal{V}}$ by so doing, and wish to determine which implementation $\mathcal{I}_{i,j}$ of the i'th stage we should choose in order to optimize for performance whilst continuing to assure correctness (i.e., guaranteeing that we will be able to obtain a cumulative value $\mathcal{V}' \stackrel{\text{def}}{=} (\mathcal{V} - \widehat{\mathcal{V}})$ under all circumstances, typical or not). Below, we first describe how one can identify, for each stage i and each possible value for $\mathcal{V}'$, which implementations are safe to execute (in the sense of not compromising correctness) in Section 3.2.1. Once we have figured out how to identify the safe implementations, we separately discuss, in Section 3.2.2 how to choose amongst them in order to optimize for the two different performance criteria – optimizing for the worst case and for the typical case.

3.2.1 Identifying safe implementations

For each $i, 0 \leq i \leq n$, let Δ_i denote the largest value that we can guarantee to obtain over the remaining stages – stages $i, (i+1), \cdots, (n-1)$ – of the computation, based upon the characterizations of the implementations that are available to us. It is evident that

$$\begin{aligned} \Delta_n &= 0 \text{ (Since there is no stage } n) \\ \text{and } \Delta_i &= \Delta_{i+1} + \max_{j=0}^{m-1}\left\{v_{ij}\right\} \end{aligned} \tag{2}$$

This computation is represented in pseudo-code form in Figure 3; since it comprises two nested loops, it is easily seen to take running time $\Theta(nm)$. (Notice that the feasibility condition of Expression 1 can be rewritten as $\mathcal{V} \geq \Delta_0$; i.e., the target value $\mathcal{V}$ needed for correctness is no smaller than the largest value that we can guarantee to obtain over [all] the stages $0, 1, \ldots, n-1$.)

Suppose now that that we are in the midst of executing the computation during some run – we have completed the stages $0, \cdots, (i-1)$ in a safe manner, and have an amount $\mathcal{V}'$ remaining of the target value to be acquired (implying that the stages $0, \cdots, (i-1)$ together

```
ComputeDeltas($I$)
1  $\Delta_n = 0$
2  for $i = n-1$ downto $0$
3      $tmp = 0$
4      for $j = 0$ to $m-1$
5          if $(v_{i,j} > tmp)$ then $tmp = v_{i,j}$
6      $\Delta_i = \Delta_{i+1} + tmp$
```

Figure 3 Pseudo-code for computing the Δ_i values – the maximum value that can be guaranteed over the stages $i, i+1, \cdots, (n-1)$ of the computation.

must have yielded a cumulative value $\mathcal{V} - \mathcal{V}'$). If we were to now choose the implementation $\mathcal{I}_{i,j}$ for the i'th stage and were to discover, upon having completed its execution, that doing so yielded the lowest (i.e., worst-case) value of $v_{i,j}$, we will need to obtain an additional amount $(\mathcal{V}' - v_{i,j})$ of value over the remaining stages $(i+1), \cdots, (n-1)$. Hence for it to be safe for us to choose the implementation $\mathcal{I}_{i,j}$ for the i'th stage, it is necessary (and sufficient) that

$$\big(v_{i,j} + \Delta_{i+1}\big) \geq \mathcal{V}' \tag{3}$$

Expression 3 above can be thought of as constituting a run-time safety check: it is safe to use the implementation $\mathcal{I}_{i,j}$ in order to obtain a remaining cumulative value $\mathcal{V}'$ if and only if Expression 3 evaluates to true.

3.2.2 Choosing an implementation to execute

Based upon the cumulative value that has been obtained over the first $(i-1)$ stages, we saw above how one can identify which of the provided implementations of the i'th stage are safe to execute. We now describe how we should choose from amongst these safe implementations in order to optimize for performance. We first consider, in Section 3.2.2.1, the case when we seek to optimize for performance in the worst case; subsequently in Section 3.2.2.2 we consider optimizing for the typical case.

3.2.2.1 Optimizing for the worst case

Conceptually speaking, we will build a table W with $(n+1)$ rows and $(\mathcal{V}+1)$ columns with the following interpretation. For any $i \in \{0, i, \ldots, n\}$ and any $\boldsymbol{v} \in \{0, 1, 2, \ldots, \mathcal{V}\}$, $W[i, \boldsymbol{v}]$ denotes the smallest delay bound that we can guarantee if we are required to obtain a target value of at least $\boldsymbol{v}$ over the stages $i, (i+1), \ldots, (n-1)$. The following recurrence defines the values for $W[i, \boldsymbol{v}]$:

$$\begin{aligned}
W[n, 0] &= 0 && \text{(Doing nothing, we trivially obtain a value 0)} \\
W[n, \boldsymbol{v}] &= \infty \text{ for all } \boldsymbol{v} > 0 && \text{(No positive value can be obtained over the non-existent } n\text{'th stage)} \\
W[i, \boldsymbol{v}] &= \min_{\{j \mid v_{ij} + \Delta_{i+1} \geq \boldsymbol{v}\}} \Big\{ c_{ij} + W[i+1, \boldsymbol{v} - v_{ij}] \Big\}
\end{aligned} \tag{4}$$

The third –recursive– equation above asserts that if we were to execute the j'th implementation, we would spend a duration c_{ij} and obtain a value v_{ij} in the worst case, thereby requiring an additional value $(\boldsymbol{v} - v_{ij})$ in the following stages. And the worst-case duration for this is, by definition, given by $W[i+1, \boldsymbol{v} - v_{ij}]$.

```
ComputeT-Table($I$)
for $i = (n-1)$ downto $0$ by $(-1)$
    for $\boldsymbol{v} = 1$ to $\mathcal{V}$
        $T[i,\boldsymbol{v}] = \infty$ // Initializing
        for $j = 0$ to $m-1$
            if $(v_{i,j} + \Delta_{i+1} \geq \boldsymbol{v})$ // I.e., implementation $\mathcal{I}_{i,j}$ is "safe"
                if $(c_{i,j} + T[i+1, \boldsymbol{v} - v^T_{i,j}] < T[i,\boldsymbol{v}])$ // Better than current choice?
                    $T[i,\boldsymbol{v}] = c_{i,j} + T[i+1, \boldsymbol{v} - v^T_{i,j}]$ // Update best response duration
                    $I_T[i,\boldsymbol{v}] = j$ // Update choice of implementation
```

Figure 4 Pseudo-code for computing the tables $T[\ ,\ ]$ and $I_T[\ ,\ ]$ – see Section 3.2.2.2.

Observe that in this third equation we are restricting the choice of implementations (the values that the index j may take) to only those that satisfy the safety/ correctness condition of Expression 3, thereby ensuring that the choice of implementations in this i'th stage will not lead to an unsafe state.

For reasons that will become evident in Section 3.3, we concurrently also build a table I_W of the same dimensions as the table W, with entry $I_W[i,\boldsymbol{v}]$ storing the index j for which the RHS in the third equation in Recurrence 4 is minimized (i.e., choice of implementations for the i'th stage that minimizes the response time while guaranteeing a cumulative value $\boldsymbol{v}$ over the stages $i,\ldots,n-1$).

3.2.2.2 Optimizing for the typical case

The approach is very similar to the one shown in Section 3.2.2.1 above. Here we build a table T; for any $i \in \{0,i,\ldots,n\}$ and any $\boldsymbol{v} \in \{0,1,2,\ldots,\mathcal{V}\}$, $T[i,\boldsymbol{v}]$ denotes the smallest delay bound that we can guarantee under all typical circumstances, if we are to obtain a target value of at least $\boldsymbol{v}$ over the stages $i,(i+1),\ldots,(n-1)$. We have

$$\begin{aligned} T[n,0] &= 0 \\ T[n,\boldsymbol{v}] &= \infty \text{ for all } \boldsymbol{v} > 0 \\ T[i,\boldsymbol{v}] &= \min_{\{j \mid v_{ij}+\Delta_{i+1} \geq \boldsymbol{v}\}} \left\{ c_{ij} + T[i+1, \boldsymbol{v} - v^T_{ij}] \right\} \end{aligned} \tag{5}$$

The only difference from Expression 4, the recurrence relation for worst-case analysis, arises in the third (recursive) equation, in specifying the value that remains to be obtained in the stages $(i+1),\ldots,(n-1)$. While in Expression 4 this is the worst-case value guaranteed by $\mathcal{I}_{i,j}$ (i.e., $v_{i,j}$), it is instead the lowest value guaranteed under *typical* circumstances (i.e., $v^T_{i,j}$) in Expression 5.

Analogous to the table I_W in Section 3.2.2.1 above, we also build a table I_T that stores, in $I_T[i,\boldsymbol{v}]$, the index j for which the RHS in the third equation in Recurrence 5 is minimized.

Implementation Details, and Running time

Tables W and I_W for worst-case analysis, and tables T and I_T for typical-case analysis, are constructed similarly in a bottom-up fashion. The procedure for computing the T and I_T tables is depicted in pseudo-code form in Figure 4 (the pseudo-code for computing the W and I_W tables is analogous and hence omitted). The bottom-most row of the table ($i == n$) is not explicitly stored, but rather implicitly assumed initialized to all zeros. Row i is filled in once all the rows $(i+1), \ldots, n-1$ have been filled. To compute each entry in row i of the table, we may need to examine each of the m alternative implementations $\mathcal{I}_{i,0}, \mathcal{I}_{i,1}, \ldots, \mathcal{I}_{i,m-1}$ (this is done in the **for** loop of lines 4–8); hence computing each entry in the table takes $\Theta(m)$ time. Since there are $(n+1) \times (\mathcal{V}+1)$ entries in the table, the entire table can be fill in with an overall running time of $\Theta(mn\mathcal{V})$.

(Note that in order to compute the entries in the table it is necessary that the values $\Delta_0, \Delta_1, \ldots, \Delta_n$ be known. Hence the pseudo-code of Figure 3 must be executed prior to calling this procedure. It was previously argued, in Section 3.2.1, that the pseudo-code of Figure 3 has a running time $\Theta(nm)$; the overall computational complexity is therefore dominated by the cost of computing the tables W and/ or T, and remains $\Theta(mn\mathcal{V})$.)

3.3 Run-time algorithms

We now explain how the tables W or T, constructed as described in Section 3.2 above, are used during run-time. Section 3.3.1 discusses how the tables W and I_W may be used to optimize for performance in the worst case, and Section 3.3.2 discusses how the tables T and I_T may be used to optimize for performance in the typical case (while guaranteeing correctness in all cases, typical or not).

3.3.1 Optimizing for the worst case

Let us first suppose that we are optimizing for the worst case, and consider some actual execution of the system during run-time. A **static** schedule would pre-select an implementation for each stage prior to commencing execution of the first stage, and not change this decision during run-time, regardless of the actual values obtained at each stage. Such a schedule is readily determined using the tables W and I_W that were computed as discussed above, in Section 3.2:

```
Let $j = I_W[0, \mathcal{V}]$
choose implementation $\mathcal{I}_{0,j}$ for stage 0
$sumVal = v_{0,j}$
for $i = 1$ to $n-1$
    Let $j = I_W[i, \mathcal{V} - sumVal]$
    choose implementation $\mathcal{I}_{i,j}$ for stage $j$
    $sumVal = sumVal + v_{i,j}$
```

Such a static schedule has the advantage of not requiring run-time monitoring: since the scheduling choices are not changed regardless of how much value is actually obtained at each stage, there is really no reason to determine this via run-time monitoring.

An **adaptive** schedule, in contrast, would, at each stage, only select which implementation of that stage should be executed in order to optimize for worst-case running time subject to the constraint that the remaining value needed for assuring safety be obtainable. For stage 0, the choice is identical to the one selected by the static schedule above. However, the choice of implementations for future stages would depend upon the actual values obtained, as determined by run-time monitoring, by the chosen implementations of already-executed stages:

```
1 Let $j = I_W[0, \mathcal{V}]$
2 execute implementation $\mathcal{I}_{0,j}$ for stage 0. Let $\widehat{v_0}$ denote the value so obtained
3 $sumVal = \widehat{v_0}$
4 for $i = 1$ to $n - 1$
5     Let $j = I_W[i, \mathcal{V} - sumVal]$
6     execute implementation $\mathcal{I}_{i,j}$ for stage $j$. Let $\widehat{v_i}$ denote the value so obtained.
7     $sumVal = sumVal + \widehat{v_i}$
```

Since $\widehat{v_i}$ may be larger than v_{ij} for each i (recall that the v_{ij} values are assumed to be very conservative *lower* bounds), an adaptive schedule will, in general, tend to be different from the static one; additionally, different executions of the instance may result in different schedules (since the $\widehat{v_i}$ values may be different on different executions).

3.3.2 Optimizing for the typical case

It is not generally possible to synthesize a completely static schedule that optimizes for the typical case, since run-time monitoring may reveal that typical-case assumptions are violated upon executing some stage and when that happens, it is essential that correctness be preserved at the cost of abandoning the pre-computed schedule. A form of "***semi-static***" schedule could be conceived of, which would

(a) Precompute a static schedule assuming typical behavior, in a manner analogous to the manner in which the static schedule optimizing for the worst case was synthesized.

(b) Corresponding to each i, $0 \leq i < n - 1$, pre-synthesize an alternative schedule for the stages $i + 1, \cdots, n - 1$ in the event that typical conditions are observed to have been violated while executing the selected implementation for the i'th stage. This alternative schedule would only seek to assure correctness without regard for performance; hence, it may simply execute at each stage the implementation characterized by largest worst-case value (the v_{ij} parameter).

However since such a semi-static schedule does not obviate the need for run-time monitoring (unlike static schedules optimizing for the worst case), there is no significant advantage to not going fully adaptive, as is done by the following run-time algorithm:

```
1 Let $j = I_T[0, \mathcal{V}]$
2 execute implementation $\mathcal{I}_{0,j}$ for stage 0. Let $\widehat{v_0}$ denote the value so obtained
3 $sumVal = \widehat{v_0}$
4 for $i = 1$ to $n - 1$
5     Let $j = I_T[i, \mathcal{V} - sumVal]$
6     execute implementation $\mathcal{I}_{i,j}$ for stage $j$. Let $\widehat{v_i}$ denote the value so obtained.
7     $sumVal = sumVal + \widehat{v_i}$
```

3.4 Characterization of Optimality

As stated in Sections 1 and 2, our objective has been to optimize for performance, measured as the duration of the computation, whilst assuring correctness: guaranteeing that the target value $\mathcal{V}$ will be obtained under all circumstances. We had pointed out that (at least) two distinct interpretations of the optimization objective seem reasonable – one optimizing for performance under all possible conditions and the other, optimizing for performance under all typical conditions– and had advocated in favor of adopting the latter interpretation (while accepting that the former may be more appropriate for certain systems). In Sections 3.2-3.3

above we defined a pair of pseudo-polynomial time algorithms that compute optimal solutions according to these two different objectives; in this section we will precisely state in what manner these solutions are optimal.

As stated in Section 3.3, the pair of tables W and I_W, and the pair of tables T and I_T, could each be used in two different ways at runtime. Tables W and I_W could be used to construct a **static** schedule or an **adaptive** scheduling strategy, both of which are optimized for the worst case; Tables T and I_T could be used to construct either a **semi-static** or an **adaptive** scheduling strategy, both optimized for the typical case. We now state the performance guarantees that are made by each of these choices of scheduling strategies.

1. **Static schedule, using Tables $\boldsymbol{W}$ and $\boldsymbol{I_W}$:** Our run-time schedule guarantees a correct schedule[5] with response time no greater than $W(0, \mathcal{V})$; additionally, no non-clairvoyant scheduling strategy can guarantee a correct schedule with smaller response time.
2. **Semi-static schedule, using Tables $\boldsymbol{T}$ and $\boldsymbol{I_T}$:** Our run-time scheduling strategy guarantees a correct schedule with response time no greater than $W(0, \mathcal{V})$ for all executions in which no implementation's behavior violates the typical-case assumptions (i.e., each implementation $\mathcal{I}_{ij}$ chosen for execution returns a value $\geq v_{ij}^T$); additionally, no non-clairvoyant scheduling strategy can guarantee a correct schedule with smaller response time for all typical executions.
3. **Adaptive scheduling, using Tables $\boldsymbol{W}$ and $\boldsymbol{I_W}$:** Our run-time schedule guarantees a correct schedule with response time no greater than $W(0, \mathcal{V})$.
   Additionally, suppose that a value $\widehat{\mathcal{V}}$ has been obtained over the first $(i - 1)$ stages during some execution of the system. Our scheduling strategy guarantees a correct schedule with remaining response time no larger than $W(i, \mathcal{V} - \widehat{\mathcal{V}})$, and no non-clairvoyant correct scheduling strategy can guarantee a smaller remaining response time.
4. **Adaptive scheduling, using Tables $\boldsymbol{T}$ and $\boldsymbol{I_T}$:** Our run-time scheduling strategy guarantees a correct schedule with response time no greater than $W(0, \mathcal{V})$ for all executions satisfying the typical-case assumptions.
   Additionally, suppose that a value $\widehat{\mathcal{V}}$ has been obtained over the first $(i - 1)$ stages during some execution of the system. Our scheduling strategy guarantees a correct schedule with remaining response time no larger than $T(i, \mathcal{V} - \widehat{\mathcal{V}})$ for all executions that satisfy the typical-case assumptions for the remaining stages, and no non-clairvoyant correct scheduling strategy can guarantee a smaller remaining response time under all typical-case executions of the remaining stages.

4 Heuristic Approaches: A Brief Exploration

We have seen (Section 3.1) that for the kinds of computations studied in this paper the problem of scheduling in a manner that optimizes for performance while assuring correctness is NP-hard; however we were able to develop (Sections 3.2–3.3) pseudo-polynomial time algorithms for solving this problem optimally. Our solutions additionally have the desirable feature that all pseudo-polynomial time processing is performed prior to run-time: during run-time once the actual computation has commenced, the schedule, defined as the choice of implementation for each stage, may be determined via rapid (constant-time) table lookup operations.

[5] Recall that a correct schedule guarantees to obtain a cumulative value $\mathcal{V}$ over all the stages of the computation.

What are our alternatives if one is does not wish to do pseudo-polynomial processing even before run-time? In that case a number of greedy heuristics suggest themselves; in this section, we briefly examine a few such heuristics:

G1: At each stage i, choose the safe implementation (recall that Section 3.2.1 explains how such safe implementations are identified) with the largest guaranteed value (i.e., largest value for v_{ij}). Stop[6] upon having obtained the target value $\mathcal{V}$.

G2: At each stage i, choose the safe implementation with the smallest execution time (i.e., smallest value for c_{ij}). As above, stop once the target value $\mathcal{V}$ has been obtained.

G3: At each stage i, choose the safe implementation with the largest typical value density per unit of execution time (i.e., largest value of the ratio v_{ij}^T/c_{ij}). Once again, stop once the target value $\mathcal{V}$ has been obtained.

For each of these heuristics, it is relatively straightforward to identify circumstances (i.e., generate problem instances) upon which the heuristic performed arbitrarily poorly. However, it often appears to be the case that for randomly-generated instances that are generated according to some particular stochastic methodology at least one out of these three greedy heuristics (or out of a few other equally obvious ones, not listed above) does not suffer too much of a performance degradation in comparison to our pseudo-polynomial optimal algorithm under typical-case conditions.[7] This leads us to conjecture that *for any particular application system for which it is reasonable to assume that the implementation choices are characterized by parameters drawn from some underlying probabilistic distribution, it may suffice to test the system's behavior in the typical case when scheduled using a number of such greedy heuristics and simply choose the heuristic with which it performs the best*: it is likely (although of course not guaranteed) that this heuristic will provide performance that is close to that provided by the optimal pseudo-polynomial time algorithmic approach described in Sections 3.2–3.3.

In the remainder of this section we will provide some evidence to back up our conjecture that some greedy heuristic seems to perform reasonably well on problem instances whose parameters are drawn from some underlying probabilistic distributions. We do so via simulation experiments upon randomly-generated workloads of a particular kind, as detailed below. We will see that our observations in these experiments reveal that for this kind of workload, the greedy heuristic **G3** offers performance that (with rare exceptions) tends to lie within 10% or so of the performance offered by the optimal algorithm; hence if we have good reason to believe that the actual workloads we will encounter are appropriately modeled by the probabilistic model we are using to generate our workload $\underline{\text{and}}$ we are willing to pay a $\approx 10\%$ performance penalty, then it is reasonable to use heuristic G3 rather than the pseudo-polynomial time optimal algorithm. (We emphasize that it remains *safe* to use the heuristic even if the probabilistic model turns out to be incorrect: assuming that the worst-case characterizations of each implementation $\mathcal{I}_{ij}$ – its v_{ij} parameter – is indeed a true lower bound on the actual value that will be obtained upon executing $\mathcal{I}_{ij}$, correctness remains assured even if performance is far from optimali due to a mismatch between the assumed probabilistic model and reality.)

[6] This assumes an interpretation of our computational model that if the required $\mathcal{V}$ units of computation are obtained upon completing i stages of the computation, then the remaining stages do not need to be executed. An alternative interpretation is to require that all stages be executed: under this interpretation, we subsequently execute that implementation of each remaining stage that has the smallest execution duration (c_{ij} parameter). Our broad conclusions remain unchanged for both interpretations.

[7] Of particular note, the performance of the heuristic tends to be *superior to* that of the optimal (pseudo-polynomial time) algorithm optimizing for the worst-case. While this is not particularly surprising – we are evaluating under typical-case conditions while that algorithm optimizes for the worst case – we felt that it merits a note.

§1: Workload Generation. We consider workloads in which there is a choice of exactly two implementations per stage (in the notation introduced in Section 2, $m = 2$) – one corresponding to "traditional" deterministic implementation and the other, to a learning-enabled component (LEC) that guarantees a smaller value but is likely to provide a greater value under typical conditions, than the traditional implementation. The parameters characterizing the workload generator are as follows:

(1) The number of stages n ($n \in \mathbb{N}$).

(2) Two positive integer parameters C_L and C_H ($C_L \leq C_H$).
The c_{ij} parameters of the implementations are drawn uniformly at random as integers from the range $[C_L, C_H]$.

(3) Two positive integer parameters V_L and V_H ($V_L \leq V_H$).
The v_{ij} parameters of the implementations are drawn uniformly at random as integers from the range $[V_L, V_H]$. For each stage, two numbers are so drawn: the smaller becomes the v_{ij} for the LEC and the larger, for the traditional (deterministic) component.

(4) Two positive integer parameters VT_L and VT_H ($\mathrm{VT}_L \leq \mathrm{VT}_H$).
- The v_{ij}^T parameters of the LEC implementations are obtained by multiplying their v_{ij} parameters by an integer drawn uniformly at random from the range $[\mathrm{VT}_L, \mathrm{VT}_H]$.
- The v_{ij}^T parameters of the traditional (deterministic) components are set equal to their v_{ij} values.

(5) A real number *Slack* ($0 < \textit{Slack} \leq 1$).
This is used to assign a value to the target parameter $\mathcal{V}$; this assigned value is *Slack* times the largest value that can be guaranteed across all the stages.

§2: The Experiments Performed. An assignment of values to each of the eight parameters $\langle n, C_L, C_H, V_L, V_H, \mathrm{VT}_L, \mathrm{VT}_H, \textit{Slack}\rangle$ constitutes a single configuration for our experiment. The precise configurations we examined are enumerated below; for each such examined configuration, we generated one hundred instances and determined the durations of the schedules that would be generated by the following five algorithm/ heuristics:

1. tOpt: The adaptive algorithm that optimizes for performance in the typical case.
2. wOpt: The adaptive algorithm that optimizes for performance in the worst case.
3. G1: The adaptive greedy heuristic in which we choose, for each state, that safe implementation that guarantees the largest value.
4. G2: The adaptive greedy heuristic in which we choose, for each state, that safe implementation that has the smallest execution duration.
5. G3: The adaptive greedy heuristic in which we choose, for each state, that safe implementation that returns, in the typical case, the largest value per unit of execution time.

in a run-time scenario in which each chosen implementation $\mathcal{I}_{ij}$ executes for a duration exactly equal to c_{ij} and returns a value exactly equal to v_{ij}^T. Note that tOpt, by design, generates the optimal schedule under these circumstances; hence for each configuration our experimental setup reports the average, over all hundred instances, of duration taken by each of the five algorithms normalized by the duration taken by tOpt. Here is one example of the kind of data reported by our experimental setup:

```
n= 5; CL=10; CH=50; VL=1; VH=5; VTL=2; VTH=10; Slack=0.5

W-OPT: 1.34
G1:    4.44
G2:    6.26
G3:    1.14
```

Table 1 Reported average of the durations of the schedules generated by Algorithm wOPT and the heuristics G1, G2, and G3, relative to the duration of the schedule generated by Algorithm tOPT, across 100 instances generated according to the configuration $\langle n = 5, C_L = 10, C_H = 50, V_L = 1, V_H = 5, \mathrm{VT}_L = 2, \mathrm{VT}_H = 10, \mathit{Slack}\rangle$, for the different values of *Slack* listed in the first column.

Slack	wOPT	G1	G2	G3
0.2	1.17	2.24	6.41	1.04
0.3	1.39	3.10	6.31	1.04
0.4	1.59	3.73	6.37	1.05
0.5	1.34	4.44	6.26	1.14
0.6	1.36	4.17	5.34	1.08
0.7	1.48	4.47	4.79	1.04
0.8	1.78	5.08	5.42	1.02
0.9	1.72	5.14	4.98	1.06

These data report that, across one hundred instances generated with the configuration

$$\langle n = 5, C_L = 10, C_H = 50, V_L = 1, V_H = 5, \mathrm{VT}_L = 2, \mathrm{VT}_H = 10, \mathit{Slack} = 0.5\rangle\,,$$

the duration taken by Algorithm wOPT was 1.34 times that taken by tOPT on average, the duration taken by heuristic G1 was 4.44 times that taken by tOPT on average, the duration taken by heuristic G2 was 6.26 times that taken by tOPT on average, and the duration taken by heuristic G3 was 1.14 times that taken by tOPT on average.

§3: Observations. We examined a wide range of configurations, and the effect of changing one or a few parameters (while keeping the other unchanged) in order to explore the performance of the two optimal algorithms and the three heuristics upon workloads that are compliant with the modeling assumptions stated in §1 above. A typical set of observations is reported in Table 1: this reports on the outcomes upon eight closely-related configurations in which only the *Slack* parameter is changed. As can be seen from the numbers in Table 1, heuristic G3 consistently offers performance close to the optimal algorithm, with the poorest relative performance for $\mathit{Slack} = 0.5$ where it is a mere 14% poorer than the optimal. The other heuristics, in contrast, may be sub-optimal by factors exceeding five and six; even the pseudo-polynomial time optimal algorithm that optimizes for the worst case is off by as much as 78% (for $\mathit{Slack} = 0.8$).

Another set of results is depicted in Table 2 – here, the parameter *Slack* is fixed (at 0.8), while the number of stages n is varied. It may be seen that once again it is heuristic G3 that performs well with a maximum performance degradation of 10% (for $n = 9$), while wOPT may be off by up to 74% and the heuristics G1 and G2 by more than a factor of six. It is also noteworthy that the performance of heuristic G3 does not appear to drop off steeply as the number of stages increases: this is significant since the setup of our experiments means that the target value $\mathcal{V}$ tends to increase linearly with the number of stages, meaning that the running time of the optimal algorithms, which are polynomial in the value of $\mathcal{V}$, will increase with increasing n. Hence, for large values of n (multistage computations with a large number of stages) the run-time savings of using the efficient greedy heuristic G3 may be particularly worth our while.

Another useful lesson learned from our experimental evaluation relates to the performance of wOPT relative to tOPT under typical-case conditions. The numbers in the columns labeled wOPT in Tables 1 and 2 indicate that one pays a significant performance penalty

Table 2 Reported average of the durations of the schedules generated by Algorithm wOpt and the heuristics G1, G2, and G3, relative to the duration of the schedule generated by Algorithm tOpt, across 100 instances generated according to the configuration $\langle n, C_L = 10, C_H = 50, V_L = 1, V_H = 5, \mathrm{VT}_L = 2, \mathrm{VT}_H = 10, \mathit{Slack} = 0.8\rangle$, for the different values of n listed in the first column.

n	wOpt	G1	G2	G3
5	1.53	4.56	4.89	1.01
6	1.51	4.84	5.21	1.04
7	1.74	5.41	5.77	1.03
8	1.41	6.15	6.46	1.07
9	1.38	6.01	6.40	1.10
10	1.50	5.99	6.52	1.02
15	1.50	5.13	5.67	1.04
20	1.54	5.44	6.11	1.05
30	1.52	5.55	6.31	1.05
40	1.42	5.37	6.14	1.10
50	1.49	5.47	6.26	1.08
100	1.51	5.64	6.49	1.08

(at least 17%, more typically in the 35% – 60% range and as high as 78%) under typical conditions by optimizing for the worst case; this provides further support for our advocating for explicitly optimizing for the typical case (while assuring safety in all cases, typical or not).

Limitations of our evaluation. Our experimental evaluation has been brief and somewhat cursory: we are not suggesting that these experiments are exhaustive enough, or have examined enough combinations of parameter values, to be able to draw authoritative or general conclusions. All that can be said is that these particular experiments do provide some support for our conjecture that perhaps efficient greedy heuristics are adequate for finding near-optimal solutions for workloads that are characterized by parameters drawn from certain well-behaved distributions. Hence our experimental evaluation is, at best, very preliminary: we plan to conduct a far more thorough evaluation in the future, considering a wider range of probabilistic models that are inspired by the observed characteristics of actual LECs and basing inferences upon more rigorous statistical methods (confidence intervals; refutable null hypotheses; etc.) than just the simple means (and standard deviations – not reported here) that we have collected and looked at thus far.

5 Generalizations

In this paper we have restricted our consideration of LEC-enabled computations (i.e., computations incorporating Learning-Enabled Components) to those that can be modeled as multi-stage computations with a choice of implementations per stage, and with each implementation $\mathcal{I}_{ij}$ characterized by a single worst-case execution time parameter c_{ij} and a pair of parameters v_{ij} and v_{ij}^T denoting the minimum value the implementation is guaranteed to yield under all and typical conditions respectively. We believe that analysis of this simple basic model is a necessary first step towards enabling the safe use of LECs in safety-critical systems. We now discuss several extensions to this basic model that would generalize it significantly in several directions and extend its applicability; we have been working on extending our analysis techniques to deal with these generalizations.

Typical and worst-case characterization of running time. The model used in this paper characterizes the running time of an implementation $\mathcal{I}_{ij}$ with a single worst-case execution time parameter c_{ij}. There is no reason why worst-case execution time could also not be determined by both worst-case and typical-case analysis – indeed, the idea of typical-case analysis was first proposed [9] in the context of estimating worst-case execution time. Although many currently-popular LECs tend to be relatively deterministic with regard to their timing behavior (see footnote 2), we could in principle have a model in which each implementation $\mathcal{I}_{ij}$ is characterized by both a c_{ij} and a c_{ij}^T denoting execution-time bounds under worst-case and typical-case circumstances. All the techniques developed in earlier sections of this paper generalized in a straight-forward manner to this situation.

State. It is possible for some *state* to be generated by each stage of a multi-stage computation and communicated to subsequent stages, with the behavior of these subsequent stages dependent upon the communicated state. For instance, the typical value that is obtained by an implementation at a particular stage may be dependent upon the particular computational operations performed by the implementations that were chosen for execution 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 of LECs may have multiple modes in which they are capable of operating: 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.

6 Conclusions

Learning-Enabled Components (LECs) based upon deep learning and similar AI-based principles are poised to play a very significant role in safety-critical autonomous CPS's; it is therefore highly desirable that the safety-critical systems research community come up with techniques that enable the analysis of such systems to both assure safety (which is essential) and optimize performance (which, for cost and related reasons, is highly desirable). This paper reports on some of our ongoing efforts in this direction. Building off recent work on *typical-case analysis* pioneered in [9] and continued in [4, 2, 5], we have argued that safety-critical systems whose run-time behavior incorporates a great deal of uncertainty should be designed to optimize for performance in the typical case (while guaranteeing safety in all cases, typical or not). We have further refined and expanded on a formal model that we had first proposed in [3], for representing the functional as well as the timing properties of some kinds of LECs that exhibit such uncertainties in a quantitative manner. 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 developed an optimal algorithm for solving this problem – i.e., scheduling the computation in a manner that is safe under all circumstances and guarantees optimal performance (in our case, the duration taken to complete the computation) under all typical circumstances. We have also proposed three greedy heuristics that are sub-optimal but can be implemented to execute very efficiently with polynomial running time. We have compared our algorithm with these heuristics (and another algorithm – one that optimizes for performance in the worst, rather than typical, case) 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 5 that has come out of our efforts in this direction).

References

1 Assuring autonomy international programme. `https://www.york.ac.uk/assuring-autonomy/`. Accessed: 2020-01-17.

2 Kunal Agrawal and Sanjoy Baruah. Adaptive real-time routing in polynomial time. In *Real-Time Systems Symposium (RTSS), 2019 IEEE*, December 2019.

3 Kunal Agrawal, Sanjoy Baruah, Alan Burns, and Abhishek Singh. Minimizing execution duration in the presence of learning-enabled components. In *Proceedings of the Second International Workshop on Autonomous Systems Design (ASD 2020)*, 2020.

4 Sanjoy Baruah. Rapid routing with guaranteed delay bounds. In *Real-Time Systems Symposium (RTSS), 2018 IEEE*, December 2018.

5 Sanjoy Baruah and Nathan Fisher. Choosing preemption points to minimize typical running times. In *Proceedings of the Twenty-Fourth International Conference on Real-Time and Network Systems*, RTNS '19, New York, NY, USA, 2019. ACM.

6 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.

7 Robert M. Nauss. The 0-1 knapsack problem with multiple choice constraints. *European Journal of Operational Research*, 2(2):125–131, 1978. `doi:10.1016/0377-2217(78)90108-X`.

8 Dr. Sandeep Neema. Assurance for Autonomous Systems is Hard. `https://www.darpa.mil/attachments/AssuredAutonomyProposersDay_ProgramBrief.pdf`. Accessed: 2019-03-07.

9 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. URL: `http://dl.acm.org/citation.cfm?id=2492708.2492836`.

10 John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Syst.*, 2(4):247–254, October 1990. `doi:10.1007/BF01995673`.

11 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 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.