Rethinking Tractability for Schedulability Analysis

Kunal Agrawal
Washington University in St. Louis
kunal@wustl.edu

Sanjoy Baruah
Washington University in St. Louis
baruah@wustl.edu

Pontus Ekberg
Uppsala University
pontus.ekberg@it.uu.se

***Abstract*—Algorithms that have been developed for solving computationally intractable schedulability analysis problems may be classified into two broad categories: exact algorithms that run in exponential time, and polynomial-time algorithms that provide approximate solutions. If exact algorithms are sought, it has traditionally been required that these algorithms have pseudo-polynomial running time. More recently, schedulability analysis algorithms that have polynomial running time but are allowed to make calls to an ILP solver have increasingly been considered tractable. When approximation algorithms are acceptable, an objective has been to obtain Fully Polynomial-Time Approximation Schemes, which are 'tunable' algorithms that provide a smooth transition between polynomial time and exponential time by letting the user of the algorithm set an appropriate value for a parameter.**

In this paper we take a fresh view on the connections between the various perspectives on what is considered to be tractable schedulability analysis. We seek to determine when the different forms of tractable analyses are applicable to a particular problem and what problem features rules them out, and demonstrate our findings upon concrete scheduling problems. We also suggest that 'pseudo-polynomial time' is perhaps a rather broad category, and propose a finer-grained classification of the class of pseudo-polynomial time algorithms.

I. Introduction

Safety-critical systems are generally required to have their safety properties validated correct prior to deployment. For real-time systems, the process of validating timing safety properties is commonly called *schedulability analysis*: GIVEN *(i)* the specifications of the computational demands of, and the timing constraints upon, the workload; *(ii)* the platform upon which this workload is to be implemented; and *(iii)* the run-time resource allocation and scheduling algorithms that will be used, DETERMINE (prior to run-time) whether the workload is guaranteed to always meet all its timing constraints.

In the early years of the discipline of real-time computing, schedulability analysis algorithms were required to have worst-case running times that are low-degree polynomials in the size of their inputs in order to be considered "efficient." Examples of efficient algorithms of this first generation include the utilization-based schedulability tests for Earliest-Deadline First (EDF) [1], [2] and Rate-Monotonic (RM) [1], [3] scheduling of collections of independent implicit-deadline sporadic tasks ("Liu and Layland tasks") upon preemptive uniprocessors. While the utilization-based EDF schedulability test is exact (i.e., necessary and sufficient), the RM test is approximate in the sense that it is sufficient but not necessary. This is, with the benefit of hindsight, not surprising: most schedulability-analysis problems, including RM schedulability analysis, have been shown [4]–[11] to be NP or coNP hard and hence unlikely to admit to exact polynomial-time schedulability tests.

By the mid- to late-1980s, however, computing capabilities had increased enough that schedulability analysis algorithms with *pseudo-polynomial* running times were considered efficient. Early examples of efficient algorithms of this kind include preemptive uniprocessor Response-Time Analysis that is an exact test for fixed-priority task systems [12]–[17] and the EDF schedulability test for 3-parameter sporadic task systems [4], [6], [18] for bounded-utilization task systems. Since then, there appears to have been a consensus within the safety-critical real-time computing community that pseudo-polynomial running time equates to efficiency, and there has been a continued quest to identify the most general scheduling algorithms and associated workload and platform models for which schedulability analysis can be done in pseudo-polynomial time, e.g., [19]–[21].

More recently, the increasingly complex nature of many schedulability-analysis problems that one encounters whilst seeking to verify the correctness of modern safety-critical cyber-physical systems, combined with the continued increase in computing capabilities that are available for the purposes of performing schedulability analysis, has motivated some researchers in the real-time computing community to move beyond this "pseudo-polynomial time barrier." Many such investigations seek to transform a schedulability-analysis problem to some other form such as an integer linear program (ILP – see, e.g., [22]–[26]), or some satisfiability modulo theories (SMT) [27], which can then be solved by an appropriate solver — i.e., an ILP solver or an SMT solver, respectively. (The real-time scheduling theory community has focused far more on ILPs than SMTs, and we will do likewise in this manuscript.) Although solving an integer linear program is itself computationally intractable (it is strongly NP-complete to decide if any feasible solution exists), excellent off-the-shelf solvers exist that, by incorporating a combination of expert techniques, special-purpose heuristics, and highly optimized implementation, are able to handle surprisingly large problem instances in reasonable amounts of time. We believe that the availability of such amazingly powerful ILP solvers, paired with advances in theoretical computer science since the mid 1980's (when the consensus had been reached on pseudo-polynomial time equating to efficiency), has opened up a

marvelous opportunity to develop new ILP-based algorithms for rapidly solving schedulability-analysis problems. The real-time scheduling community has had some noteworthy successes in this direction recently; such successes are leading to the emergence of a fresh community-wide consensus that it be considered 'efficient' to solve a schedulability-analysis problem by converting it to an ILP and calling an ILP solver.

Does the emergence of this consensus mean that all schedulability-analysis problems may now be considered efficiently solvable? By no means: a range of complexity classes have been defined in computational complexity theory that are widely believed to contain problems not in NP (in much the same manner that NP is widely believed to contain problems not in P, i.e., not solvable by polynomial-time algorithms). Showing a schedulability analysis problem to be hard for one of these complexity classes would offer strong evidence that it cannot be efficiently solved even with an excellent ILP solver. The complexity class Σ_2^{P} (we provide a brief primer on complexity classes in Section II below) is an example: it is widely conjectured that $\mathsf{NP} \subsetneq \Sigma_2^{\mathsf{P}}$. Woeginger [28] discusses the implications (our emphasis): "*The consequences of [this conjecture] are devastating: If you hit a Σ_2^{P}-complete problem, then there is no way of formulating it (in polynomial time) as an integer program (of polynomial size). [...] As a consequence, our powerful and well-developed toolkit for integer programming is of little use in the realm of Σ_2^{P}-completeness. Σ_2^{P}-complete problems are much, much, much, much, much harder than any problem in* NP *or* coNP *and anything that can be attacked via ILP solvers.*"

Summarizing the discussion above, current approaches towards developing exact schedulability-analysis algorithms appear to be centered on the following three beliefs:

1) It is 'best' (i.e., most efficient) to obtain a polynomial-time algorithm. If this is not possible, then one should look for a pseudo-polynomial-time one; failing that, it is becoming increasingly acceptable to settle for a 'polynomial-time + ILP-solver' algorithm – an algorithm that has polynomial running time and is additionally allowed to make calls to an ILP solver. If this, too, is not possible, then the problem should be considered to be truly intractable.
2) Showing a schedulability-analysis problem to be NP-hard indicates that it is not likely to be solvable by a polynomial-time algorithm, but leaves open the possibility that it is solvable by a pseudo-polynomial-time algorithm. Showing a problem to be NP-hard in the strong sense rules out this possibility, but leaves open the possibility that it is solvable by a 'polynomial-time + ILP-solver' algorithm.
3) Showing a problem to be Σ_2^{P}-hard or Π_2^{P}-hard[1] rules out the possibility of polynomial-time, pseudo-polynomial-time, or 'polynomial-time + ILP-solver' algorithms: a Σ_2^{P}-hard or Π_2^{P}-hard problem is truly intractable (as stated above, it is considered to be "much, much, much, much, much harder than any problem in NP or coNP" [29]).

[1]The complexity class Π_2^{P} is also described in Section II.

We have discussed above one approach for dealing with the fact that many schedulability analysis problems are not in the complexity class P and therefore unlikely to be solvable by polynomial-time algorithms: allow for the use of pseudo-polynomial time algorithms, or even for exponential-time algorithms provided the exponential part is restricted within calls to a well-crafted ILP solver. Another popular approach to dealing with such inherent intractability is via **approximation**: continue to enforce the restriction that algorithms have polynomial running time, but settle for *sufficient*, rather than exact, schedulability analysis algorithms. A *Fully Polynomial-Time Approximation Scheme* or FPTAS is a particularly attractive option in this regard since its running time is controllable by a user-set parameter $\epsilon > 0$. That is, the running time of an FPTAS is polynomial in the size of its input and $(1/\epsilon)$, while its accuracy increases with decreasing ϵ: the smaller the value of ϵ the closer the FPTAS is to an exact test. By adjusting the value of ϵ, one thereby achieves a smooth transition between polynomial and exponential running times. FPTAS's have been proposed for both EDF schedulability [30] and fixed-priority schedulability [31] of sporadic task systems upon preemptive uniprocessors.

This work. As the discussion above reveals, the perspective of the real-time systems community as to what should be considered to be tractable has evolved over the years. Motivated by recent developments in computing (with regards to both advances in algorithms, and the availability of computing hardware and software tools), we believe that the time is right for another examination; this manuscript reports on our efforts at doing so. Among the specific **contributions** with regards to *exact* schedulability analysis, we

1) Refute the third of the "beliefs" listed above, by showing that problems belonging to complexity classes that are believed to be even more intractable than Σ_2^{P} or Π_2^{P} may have pseudo-polynomial-time algorithms. Hence, a problem being Σ_2^{P} or Π_2^{P} hard does not rule out the possibility of there being a pseudo-polynomial-time algorithm for solving it.
2) Illustrate the above fact by developing a pseudo-polynomial time algorithm for a specific example problem– the ADVERSARIAL PARTITIONING problem (Definition 1) – that is known to be Σ_2^{P}-complete.
3) Make a case that not all pseudo-polynomial algorithms should be considered 'equally tractable,' and propose a finer-grained classification within the class of pseudo-polynomial time algorithms.

Our **main contributions** with regards to *approximate* schedulability analysis have been to

1) Show that FPTAS's may exist for problems that are harder than NP-hard (or coNP-hard), by constructing an FPTAS for the Σ_2^{P}-complete ADVERSARIAL PARTITIONING problem.
2) This FTPAS is obtained using a fairly standard technique for deriving an FPTAS from a pseudo-polynomial time algorithm (the one listed above as the second contribution

for exact schedulability analysis). However, this technique is not always applicable: we illustrate this on a scheduling problem, the MEMORY-CONSTRAINED TASK SELECTION PROBLEM (Definition 5), for which a pseudo-polynomial time algorithm is easily designed but which cannot have an FPTAS (under standard complexity-theoretic assumptions).

Taken together, our contributions serve (amongst other things) as a reminder that to understand tractability in real-time schedulability analysis, in either the exact or the approximate sense, we may require a more nuanced view than simply mapping problems to the traditional computational complexity hierarchy.

Organization. The remainder of this manuscript is organized as follows. In Section II we provide a brief background on those concepts in computational complexity that are relevant to us in this paper. In Section III, we present our findings regarding the existence of pseudo-polynomial time algorithms, and the implications of their existence. We do likewise concerning approximation algorithms in Section IV, and conclude in Section V by placing this work within a larger context of real-time schedulability analysis.

II. Some Background from Complexity Theory

In this section we briefly review some concepts from computational complexity theory [32], [33] that we will use in the remainder of this manuscript.[2] The class P of problems that are known to be *solved* by algorithms with running time polynomial in the size of their inputs, and the class NP of problems for which claimed solutions can be *verified* by algorithms with running time polynomial in the size of their inputs, are (along with coNP, the class of problems whose complements are in NP) the foundational cornerstones of computational complexity theory. It is very widely believed that $\mathsf{P} \subsetneq \mathsf{NP}$ (i.e., there are polynomial-time verifiable problems that cannot be solved in polynomial time); and that $\mathsf{coNP} \neq \mathsf{NP}$ (i.e., there are problems in NP that are not in coNP, and vice versa). The *polynomial hierarchy* [34] extends computational complexity theory beyond the classes P and NP by considering abstract computers equipped with an *oracle*: a "black box" that is able to solve a specific decision problem efficiently (usually it is assumed that the oracle solves its problem in constant time, but for our purposes it would be equivalent if it solves it in polynomial time). The complexity class P^{NP} (also called Δ_2^{P}) is the class of all problems that can be solved in polynomial time by an algorithm that is equipped with an oracle for solving some NP-complete problem. Since it is NP-complete to determine whether an integer linear program has a solution, the problems solvable by what we have referred to as 'polynomial time + ILP-solver' algorithms in Section I are exactly of this complexity class. The complexity class $\mathsf{NP}^{\mathsf{NP}}$ (also called Σ_2^{P}) denotes the class of all problems that can be verified in polynomial time by an algorithm that is equipped with an oracle for solving some NP-complete problem. Similar to how coNP relates to NP, complexity class $\mathsf{coNP}^{\mathsf{NP}}$ (also called Π_2^{P}) denotes the class of problems whose complement problems are in $\mathsf{NP}^{\mathsf{NP}}$. This idea is generalized for any $k \in \mathbb{N}$: Σ_k^{P} and Π_k^{P} are defined assuming access to an oracle that is complete for $\Sigma_{k-1}^{\mathsf{P}}$. The entire polynomial hierarchy (PH) is contained in many other classes, like EXP, the class of all problems solvable in exponential time. The relationship amongst the complexity classes considered in this paper is shown in Figure 1 as a Venn diagram.

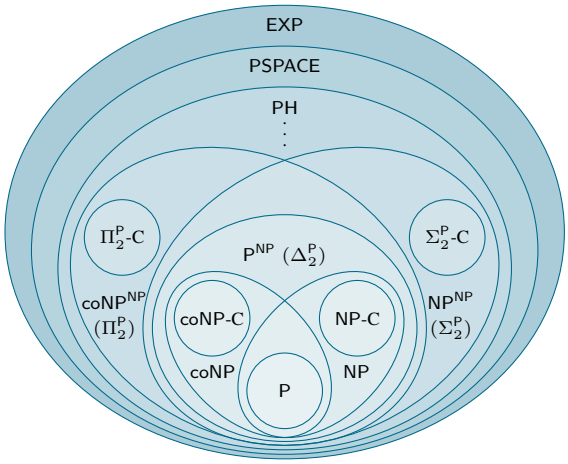

Fig. 1. Some complexity classes.

Strong and weak sense of hardness. The complexity classification above may be further sub-divided by specifying hardness in the weak or strong sense. Somewhat informally, if a problem's specification includes numbers, and there is an algorithm for solving the problem that has running-time polynomial in the *values* of the numbers of the input instance – then there is a *pseudo*-polynomial time algorithm for solving it. Problems that are shown to be NP- or coNP-hard in the *strong* sense cannot be solved exactly in pseudo-polynomial time (assuming $\mathsf{P} \neq \mathsf{NP}$), but problems that are (co)NP-hard in the *weak* (or *ordinary*) sense could have such algorithms.

III. Pseudo-polynomial Time Schedulability Analysis

In this section we investigate the existence of pseudo-polynomial time algorithms for schedulability analysis problems. Recall (Section I) that conventional wisdom considers such problems to be of intermediate complexity: easier than those with only 'polynomial-time + ILP solver' algorithms (or indeed, Σ_2^{P}-hard or Π_2^{P}-hard problems), but harder than problems in P. In Section III-A we refute this conventional wisdom by proving that there are Σ_2^{P}-hard or Π_2^{P}-hard problems (indeed, even EXP-complete problems) for which pseudo-polynomial time algorithms exist, but that pseudo-polynomial

[2] In order to keep things simple the presentation is intentionally informal and not always precise: for instance, while most of the concepts discussed here differ in their applicability to *decision problems* – those for which there is a "YES/ NO" answer – and *optimization problems*, we do not make this distinction here but treat both decision and optimization problems in similar fashion.

time algorithms cannot exist outside of EXP. These observations are made in general terms; in Section III-B we ground this to real-time scheduling by designing a pseudo-polynomial time algorithm for a schedulability-analysis problem that is Σ_2^{P}-complete.

Although the complexity class P does not distinguish between problems for which the most efficient algorithms have running times that are linear, quadratic, cubic, etc., in problem instance size, we often find it meaningful to provide a finer-grained characterization of the algorithms within P (e.g., a $\Theta(n \log n)$ sorting algorithm such as Mergesort is generally considered to be more efficient than a $\Theta(n^2)$ one like Insertion Sort). We will take a similar finer-grained perspective on pseudo-polynomial time in Section III-C, drawing from examples in real-time scheduling theory to illustrate why it it meaningful to do so.

A. On the whereabouts of pseudo-polynomial time algorithms

There are problems in NP that are not solvable via pseudo-polynomial time algorithms (assuming P $\neq$ NP); indeed, it is well known that problems that are NP-complete *in the strong sense* cannot have pseudo-polynomial time algorithms (again, assuming P $\neq$ NP). But it turns out that there are problems harder than NP-hard for which pseudo-polynomial time algorithms exist. Indeed, we can find problems with pseudo-polynomial time algorithms essentially anywhere in EXP, as we demonstrate below.

A complexity class C is said to be closed under polynomial-time reductions if whenever problem X can be reduced to problem Y in polynomial time, and Y is in C, then X is also in C. Most of the complexity classes we usually consider are closed under polynomial-time reductions, for example P, NP, coNP, all other classes in the polynomial hierarchy, PSPACE and EXP. The following rather straightforward observation shows that all such classes in EXP must contain complete problems (albeit artificial ones) that can be solved in pseudo-polynomial time.

Observation 1. *If* C *is a complexity class contained in* EXP *and* C *is closed under polynomial-time reductions, then there exist* C*-complete problems with pseudo-polynomial time solutions.*

Proof. Let L_1 be a C-complete language. Since L_1 is in C and C $\subseteq$ EXP, there exists an algorithm for solving L_1 that runs in time

$$O\left(2^{n^k}\right) \tag{1}$$

for some constant k.

From L_1 we define language L_2 by padding every instance with a large number,

$$L_2 = \left\{ (x, a) \mid x \in L_1 \text{ and } a = 2^{|x|^k} \right\},$$

where $|x|$ is the length of the representation of x and k is the constant from Eq. 1.

First, we observe that it is trivial to reduce L_1 to L_2 in polynomial time: Given instance x of L_1 we simply calculate $a = 2^{|x|^k}$ and output (x, a) as an instance of L_2. Clearly, we then have $(x, a) \in L_2$ if and only if $x \in L_1$. Since L_1 is C-complete, the reduction shows that L_2 must be C-hard.

Second, we observe that it is also trivial to reduce L_2 to L_1 in polynomial time: Given instance (x, a) of L_2 we simply check that $a = 2^{|x|^k}$ and if so output x as an instance of L_1 (or otherwise output some predefined $x' \notin L_1$). Since L_1 is in C and C is closed under polynomial-time reductions, L_2 is also in C, and is therefore C-complete.

Finally, we can determine if an instance (x, a) belongs to L_2 in pseudo-polynomial time: We first check that $a = 2^{|x|^k}$ and if so use the algorithm for L_1 to see if $x \in L_1$ in time $O(2^{|x|^k}) = O(a)$. Therefore L_2 is C-complete and can be solved in pseudo-polynomial time. □

The problems constructed above are of course entirely artificial, in that they contain big numerical parameters that do not actually have any particular significance for the problem, other than acting as a way to "cheat" in the existence of pseudo-polynomial time algorithms. In Section III-B we will, however, see a scheduling problem that is complete at the second level of the polynomial hierarchy (Σ_2^{P}-complete), but still allows a pseudo-polynomial time algorithm.

An interesting observation to make is that, since there are EXP-complete problems that are solved by pseudo-polynomial time algorithms, the Time Hierarchy Theorem [35] tells us that P $\subsetneq$ *pseudo-polynomial time*, even if P = NP.

On the other hand, problems outside of EXP cannot have pseudo-polynomial time algorithms, as demonstrated by the following simple observation.

Observation 2. *If problem L is not in* EXP*, then L cannot be solved in pseudo-polynomial time.*

Proof. Assume to the contrary that L can be solved by a pseudo-polynomial time algorithm with running time $O(\max(n, N)^k)$, where n is the size of the representation of the input and N the largest numerical parameter. Since we need $\log N$ bits to represent N we must have $\log N \leq n$ and therefore $N \leq 2^n$. But then we can solve L in time $O(\max(n, 2^n)^k) = O(2^{kn})$, and L must be in EXP. □

In fact, the proof of Observation 2 shows that no problem outside of complexity class E (exponential time with a linear exponent) can have a pseudo-polynomial time algorithm. Since E $\subsetneq$ EXP, this may seem to contradict Observation 1, which states that there are EXP-complete problems with pseudo-polynomial time algorithms. However, these observations are in fact compatible, a straightforward padding argument (not dissimilar to the one used for Observation 1) show that all E-complete problems are also EXP-complete (see, for example, Theorem 1.1 in [36]).

B. A pseudo-polynomial time algorithm for a particular schedulability problem that is Σ_2^{P}-complete

We saw in Section III-A above that pseudo-polynomial time algorithms may exist not just for problems in NP or coNP, but also for problems that are believed to be strictly harder than such problems. The problems created to show this in Section III-A were completely artificial, created by padding instances with large numbers, but are all such examples artificial, or are there schedulability problems outside of NP and coNP that can be solved in pseudo-polynomial time? A potential candidate for this is the uniprocessor EDF-schedulability problem for bounded-utilization Digraph Real-Time (DRT) tasks. A pseudo-polynomial time exact test was derived in [21], and the problem is coNP-hard as it generalizes the corresponding schedulability problem for sporadic tasks [37]. The DRT schedulability problem is however not obviously in coNP, and we may indeed suspect that it is outside NP and coNP.

To make a stronger point we systematically derive, below, a pseudo-polynomial time algorithm for a real-time scheduling problem[3] that is known to be Σ_2^{P}-complete. This demonstrates that encountering a Σ_2^{P}-complete scheduling problem is not *necessarily* as "devastating" as the quote from Woeginger [28] (reproduced in Section I) would suggest.

The ADVERSARIAL PARTITIONING Problem [38] (defined below) is related to certain problems arising from security considerations in resource-constrained embedded systems. Let us suppose that we have set Γ_A of implicit-deadline sporadic tasks that is to be partitioned between a pair of processors, each of which is scheduled during run-time according to the preemptive uniprocessor EDF algorithm. A malicious adversary wishes to launch an attack that requires it to execute all the tasks in another set Γ_B of implicit-deadline sporadic tasks, also to be partitioned between the same pair of processors. Can we partition the tasks in Γ_A amongst the two processors in such a manner that all the tasks in Γ_B cannot fit upon the capacity that remains on the two processors?

Let us further suppose that the tasks in $\Gamma_A \cup \Gamma_B$ are harmonic, and hence that their utilizations may be expressed as integer multiples of $1/T_{\max}$, where $T_{\max}$ is the largest period among all tasks. We relate this described scheduling problem to the ADVERSARIAL PARTITIONING decision problem.

Definition 1 (ADVERSARIAL PARTITIONING [38])**.**
Instance: A positive integer S and two (multi-)sets of positive integers

$$A = \{a_1, a_2, \ldots, a_{|A|}\} \quad \text{and} \quad B = \{b_1, b_2, \ldots, b_{|B|}\}.$$

Question: Can the elements in set A be partitioned between two bins, each of capacity S, such that all the elements in set B cannot be partitioned in the remaining capacity upon these bins? □

Letting $S = T_{\max}$ and letting multisets A and B denote the utilizations of the tasks in Γ_A and Γ_B, respectively expressed in integer multiples of the basic unit of utilization $1/T_{\max}$, the scheduling problem described above is directly equivalent to ADVERSARIAL PARTITIONING. We note that the assumption of harmonic periods is what makes it possible to (trivially) reduce from this scheduling problem to ADVERSARIAL PARTITIONING without any exponential blowup in the magnitudes of numerical parameters (we avoid representing utilizations as integer multiples of a potentially exponentially-large hyper period). Reducing in the other direction is also trivial.

ADVERSARIAL PARTITIONING was shown to be Σ_2^{P}-complete by Johannes [38, (Corollary 2.2.1, p. 40)], even in the restricted case of $S_A + S_B = 2S$, where $S_A = \sum_{a_i \in A} a_i$ and $S_B = \sum_{b_i \in B} b_i$. Hence it is one of those problems that, in all likelihood, cannot by formulated as an ILP in polynomial time, and cannot even be solved if we allow a polynomial number of calls to an ILP solver. It nevertheless admits to a pseudo-polynomial time algorithm; this algorithm is listed in pseudo-code form as Algorithm 1 and is explained below.

1) First, two for-loops (lines 2–5 and 7–9) apply the standard pseudo-polynomial time algorithm for SUBSET SUM to determine all possible sums for subsets of A and B that are no greater than S. These are stored in the sorted lists L_A and L_B.
2) Then, a third for-loop (lines 10–16) considers each such possible subset sum y_A of A, and finds the largest subset sum y_B of B such that y_A and y_B can be placed together in the same bin. Then it checks if the remaining subsets with sums $S_A - y_A$ and $S_B - y_B$ can be placed together in the second bin. If not, we have found a valid partitioning of A that prevents the partitioning of B.

The running time of this algorithm is pseudo-polynomial. More specifically, it is $O((|A| + |B| + \log S)S)$ since the size of the lists L_A and L_B never exceeds S, and since the search on line 11 can be done in time $O(\log S)$ using binary search.

C. A more fine-grained take on pseudo-polynomial time

Pseudo-polynomial time algorithms are only to be considered efficient for problems where numerical parameters are actually expected to be reasonably small. In real-time scheduling problems, numerical parameters are typically representing time in some time unit, and hence tend to be fairly small as the represented time should make sense on timescales for which systems are designed. For example, we should fully expect that there will never exist a real-world periodic task with some numerical parameter on the order of 2^{100}; though such a parameter takes only 100 bits to *represent*, it would have a *magnitude* that exceeds the age of the universe measured in picoseconds.

[3] We readily admit that this scheduling problem may also appear somewhat artificial in terms of immediate practical considerations. Our point is not that it should be a highly practical scheduling problem, but that it is entirely natural in its representation. It is a simplified version of patterns that can occur in security-cognizant scheduling.

```
1  L_A = ⟨0⟩
2  for i ← 1 to |A| do
3  |   L_A ← merge(L_A, L_A + a_i, S)
4  |   /* L_A + a_i returns a list with a_i
   |    added to each element in L_A.
   |    Function merge(L_1, L_2, x) returns the
   |    sorted list obtained by merging
   |    sorted lists L_1 and L_2, removing
   |    any duplicates and any elements
   |    greater than x. */
5  end
6  L_B = ⟨0⟩
7  for i ← 1 to |B| do
8  |   L_B ← merge(L_B, L_B + b_i, S)
9  end
10 for each y_A ∈ L_A do
11 |   y_B ← the largest y_B ∈ L_B such that y_A + y_B ≤ S
12 |   if S_A − y_A + S_B − y_B > S then
13 |   |   return success
14 |   |   /* This partitioning of A
   |   |    prevents partitioning of B. */
15 |   end
16 end
17 return failure
18 /* No partitioning of A prevents the
    partitioning of B. */
```

Algorithm 1: A pseudo-polynomial time algorithm for ADVERSARIAL PARTITIONING.

Even though many numerical parameters in scheduling problems tend to be fairly small in magnitude, they are often not *very* small. For example, the parameters of a periodic task, if measured in units of CPU clock cycles, easily ranges into the thousands, millions, or even billions. It is no surprise then that the practical running time of a pseudo-polynomial time algorithm, which is polynomial in the size n of the representation of the instance and the magnitude N of the largest numerical parameter, depends heavily on the exponent applied to N. Running times that are linear, quadratic, cubic, etc. in N all count as pseudo-polynomial, though anything more than linear could quickly get intractable even with the reasonably small magnitudes N that we may expect for some scheduling problems. With this in mind we make the case that we should strive not just to find pseudo-polynomial time algorithms for scheduling problems, but to try hard to find pseudo-*linear* ones (defined below) whenever possible. We believe that it is very meaningful to make a clear separation between pseudo-linear time algorithms and other pseudo-polynomial time algorithms.

Definition 2 (Pseudo-linear)**.** An algorithm is pseudo-linear time if its runtime is $O(n^k \times N)$, where n is the size of the representation of the problem instance, N is its largest numerical parameter, and k is a constant. □

Many well-known pseudo-polynomial time algorithms in real-time systems are in fact pseudo-linear, which helps to explain their good practical performance. Examples of this are the standard Response-Time Analysis (RTA) for FP-scheduled constrained-deadline sporadic tasks [12]–[17], and the Processor Demand Analysis (PDA) [7], [18] for EDF-scheduled arbitrary-deadline bounded-utilization sporadic tasks.

But closer examination of prior pseudo-polynomial time algorithms reveals that some are in fact not pseudo-linear. A couple of examples come from the EDF-schedulability of mixed-criticality sporadic tasks. The sufficient test presented in [39] runs in pseudo-linear time in itself, but is supposed to be used in conjunction with a virtual deadline-tuning procedure that makes it pseudo-quadratic. An improved test is presented in [40], which improves the precision of the test in [39] while keeping pseudo-polynomial runtime. The improved sufficient test is however pseudo-quadratic in itself, and the deadline-tuning procedure is pseudo-cubic.

Clearly, and unsurprisingly, the above examples show a trade-off between precision and analysis complexity. Another example of this comes from the mixed-criticality FP-schedulability analysis presented in [41], where two different sufficient tests are presented with different precision, one being pseudo-linear and the other pseudo-quadratic.

An example of where a pseudo-quadratic algorithm could be made into a pseudo-linear one without sacrificing any precision comes from [21]. There, an exact EDF-schedulability test for DRT tasks was presented that is pseudo-quadratic, but an optimization was then applied that turned it pseudo-linear, which led to, in the authors' words, "*a huge performance improvement using this optimization*."

Another aspect, often overlooked, of pseudo-polynomial time algorithms is whether they are sensitive to the units in which numerical parameters are specified. For an example, consider a task system that is specified with numerical parameters in units of milliseconds, and consider what happens if we change the time unit to microseconds instead. For a pseudo-linear schedulability analysis, as per Definition 2, it would be fine to take a thousand times longer to analyze the task system when it is specified in microseconds instead. For a pseudo-quadratic or pseudo-cubic analysis it could take a million or a billion times longer. However, this seems like a mostly undesirable effect, especially if the task systems specified in different units are in fact semantically equivalent. We would like the running time of a pseudo-polynomial time algorithm to be robust in this regard, and dependent on the largest numerical parameter only in the greatest possible time unit that it could be expressed in. This motivates the below definition.

Definition 3 (Robust pseudo-polynomial time)**.** An algorithm is *robustly pseudo-polynomial* if it runs in time that is polynomial in the size n of the representation of the instance and in N/G, where N is its largest numerical parameter and G is the greatest common divisor among all its numerical parameters. □

As with pseudo-polynomial time in general, it would be a good idea to be somewhat flexible in what we consider a numerical parameter in the above definition. For example, in a multiprocessor schedulability problem we may be given as input instance a tuple (Γ, m) consisting of a sporadic task system Γ and the number of processors m. In this case it seems most meaningful to consider only the numerical parameters in the task system Γ, and omit m from the calculation of the greatest common divisor G, as m does not share a unit with the other parameters.

To consider some examples again, RTA is easily seen to be robustly pseudo-linear, and so is PDA for bounded-utilization task systems if we apply the usual optimization of only evaluating discontinuous points of the demand bound function. The pseudo-polynomial time algorithm for ADVERSARIAL PARTITIONING given in Algorithm 1 is also robustly pseudo-linear.

On the other hand, the mixed-criticality EDF-schedulability tests from [39] and [40], exactly as stated, are not robustly pseudo-polynomial. An interesting middle-ground can be found in the exact semi-clairvoyant EDF-schedulability test for mixed-criticality tasks with graceful degradation in [42, Theorem 7], which is pseudo-quadratic but only linearly robust, i.e., it runs in time $O(\mathsf{poly}(n) \times N^2/G)$. To highlight some potential sources of (non-)robustness we dig a bit deeper into [42, Theorem 7] in the following. There we see that the presented schedulability test has two nested universal quantifiers. The first of those is quantifying over every possible integral length t of time windows, up to a pseudo-linear upper bound, while the other is quantifying over certain job release time points in such intervals of size t. The first quantifier is the source of non-robustness: if every numerical parameter of the tasks are increased by a joint factor x, then the number of elements quantified over also increases by the same factor x. The inner quantifier is however not a source of non-robustness: If all task parameters increase by a joint factor, then the intervals to consider job releases in also grow in size, but the total number of jobs stay exactly the same because the job releases will be further apart by the same factor. Similar patterns to the above can be seen in other schedulability tests in the literature, but non-robustness can also arise in many other ways.

Armed with Definitions 2 and 3, we believe it is reasonable to consider that **robust pseudo-linear time is the "best" kind of pseudo-polynomial time**, which we should, when possible, strive for when designing pseudo-polynomial time algorithms. Further, we believe that it is meaningful to differentiate pseudo-linear from non-pseudo-linear, and robust from non-robust, when reporting new results in real-time scheduling theory.

IV. SUFFICIENT SCHEDULABILITY ANALYSIS IN POLYNOMIAL TIME

Whereas Section III mainly dealt with exact tests for schedulability analysis, we now turn our attention to polynomial-time tests that are sufficient, rather than exact, for determining schedulability: a task system that is deemed schedulable by a sufficient schedulability test is guaranteed to indeed be schedulable, while the remaining task systems may or may not be schedulable. *Speedup factors* [43]–[45] are a commonly used quantitative metric of the effectiveness of sufficient schedulability tests. The speedup factor of a sufficient schedulability test $\mathcal{A}$ is defined to be the smallest real number $\delta \geq 0$ such that if any task system Γ is schedulable upon a unit-speed processor, then $\mathcal{A}$ will determine that Γ is schedulable upon a speed-$(1+\delta)$ processor.[4] Smaller speedup factors denote 'better' (i.e., closer to optimal in the worst case) sufficient tests. Thus, obtaining a good sufficient schedulability test may be thought of as obtaining a good approximation algorithm that minimizes the speedup factor.

FPTAS's. In the theory of approximation algorithms [46], [47], it is widely accepted that an FPTAS (see, e.g., [48, p. 1107] for a textbook description) is the 'best' kind of approximation algorithm: it allows for approximations that are arbitrarily close to the optimal by appropriately assigning a value to a parameter δ. An FPTAS for a sufficient schedulability test may be defined as follows:

Definition 4 (Schedulability Analysis FPTAS)**.** A fully polynomial-time approximation scheme (FPTAS) for a schedulability analysis problem is an algorithm that, given as input any problem instance Γ and a parameter $\delta > 0$, returns "unschedulable" if Γ is unschedulable on speed-1 processors, and returns "schedulable" if Γ is schedulable on speed-$(1/(1+\delta))$ processors. Its running time is bounded by a polynomial in the two parameters $|\Gamma|$ and $(1/\delta)$. □

A. An FPTAS for Adversarial Partitioning

Recall that in the ADVERSARIAL PARTITIONING problem, we are given two sets of positive integers, A and B, and two bins of capacity S. We ask the question, can the set A be partitioned into the two bins which makes it impossible for B to be partitioned without exceeding the capacity of at least one bin. In Section III, we described a pseudo-polynomial (pseudo-linear) time algorithm (Algorithm 1) for this problem. We will now see that this pseudo-polynomial algorithm can be converted into an FPTAS (depicted in its entirety as pseudo-code in Algorithm 2) for adversarial partitioning.

First, let us understand what it means for an algorithm to be an FPTAS for adversarial partitioning (this is not immediately obvious since adversarial partitioning is a decision problem). We will say that an algorithm X is an FPTAS for this problem if the following properties hold.

1) If the FPTAS declares success, then A can indeed be partitioned amongst two capacity-S bins such that B cannot be accommodated on these bins.

[4] In the real-time scheduling literature it is more common to refer to $1+\delta$ as the speedup factor, and not δ. We choose the latter form to make the relation to the FPTAS approximation parameter more obvious.

2) If the FPTAS declares failure, then OPT cannot partition A amongst two capacity-$(1+\epsilon)\times S$ bins such that B cannot be partitioned on the remaining space on these bins.

By equating bin-size to processor speed, it is evident that an FPTAS satisfying these properties will serve as an Schedulability Analysis FPTAS (Definition 4) for the scheduling problem discussed just before Definition 1 in Section III.

The basic idea of the algorithm is similar to Algorithm 1, and is inspired by an FPTAS for the SUBSET SUM problem [48, p. 1128]. The reason that Algorithm 1 has pseudo-polynomial running time is that it constructs lists L_A and L_B which may each contain as many as S elements. Instead, the FPTAS will construct smaller lists by 'discarding' some possible subsets of A and B whose sizes are close to other sets whose sum are maintained in the lists.

In particular, given a list L of integers sorted in increasing order and a parameter $\delta > 0$, a function trim(L, δ) does (see [48, p. 1130]) abstractly the following:

- The first element in L is retained.
- If an element y is retained, then all elements in L that are $> y$ and $\leq (1+\delta)\cdot y$ are discarded. Hence, the next retained element is the smallest element in L that is $> (1+\delta)\cdot y$.

The FPTAS in Algorithm 2 works as follows. At every step, it adds a_i to all elements of L_A and then it trims the list L_A. Let y_A be an element in L_A; if y_A is retained, then all subsequent elements which are larger than y_A but are smaller than $\left((1+\frac{\epsilon}{2|A|})\cdot y_A\right)$ are removed. List L_B is subject to a similar process. After these steps, the algorithm is similar to the pseudo-polynomial algorithm (Algorithm 1). In particular, for every element y_A in L_A, we check if we can show that it generates an adversarial partition by finding an element y_B in L_B which barely fits with y_A in S. Since y_B could represent an element upto size $((1+\epsilon)\cdot y_B)$ if the remaining $S_A - y_A$ elements do not fit with $(S_B - (1+\epsilon)\cdot y_B)$ remaining elements of B, then we have found an adversarial partition.

To establish the correctness of the algorithm, we first observe that the lengths of these trimmed lists remain polynomial in the size of the input and $1/\epsilon$. Since the running time of the algorithm is polynomial in $L_A, L_B, |A|, |B|$ it follows that the entire algorithm has polynomial running time:

Observation 3. *The size of L_A is at most $O((|A|\log S)/\epsilon)$ at any time. Similarly for the size of L_B.*

Proof. If an element y exists in the list, then the next element is larger than $((1+\epsilon/(2|A|))\cdot y)$. The largest element is at most S. Therefore, if the total number of element is k, then $(1+\epsilon/(2|A|))^k \leq S$. Therefore, $k \leq \ln S/\ln(1+\epsilon/(2|A|)) = O((|A|\log S)/\epsilon)$. □

Now let us look at the properties of these trimmed lists. First, two observations that are proven in [48, p. 1132]. The first one is easily seen since we only ever discard elements from L_A:

Observation 4. *If $y_A \in L_A$, there is an actual subset of A with sum y_A.*

More importantly, we observe that the repeated trimmings are 'safe.'

Observation 5. *If there is a subset of A with sum z, then there must exist an element $y \in L_A$ such that $z/y \leq (1+\epsilon)$.*

Proof. After the i'th trimming, if z should be in L_A, then we will have an element y between $z/(1+\epsilon/(2|A|))^i$ and z. This can be seen by doing an induction on i. Therefore, after we add all element a_i and trim A times, for every possible subset sum z of A we will have an element y between $z/(1+\epsilon/(2|A|))^{|A|}$ and z in the list L_A. Applying some simple algebraic inequalities gives us the result. □

We now show that this algorithm is an FPTAS.

Lemma 1. *Algorithm 2 satisfies the FPTAS conditions for adversarial partitioning.*

Proof. Due to the previous observation, notionally, an element $y_A \in L_A$ 'represents' all actual subset sums of A in

$$\left(y_A, (1+\epsilon)\cdot y_A\right)$$

Therefore, for the rest of the algorithm, for every element $y_A \in L_A$, we check if putting this subset on one of the bins satisfies the adversarial partitioning property. We find the largest subset $y_B \in L_B$ that fits with y_A on one bin. Since y_B represents subsets of size upto $(1+\epsilon)y_B$, we optimistically assume that $y_A + (1+\epsilon)y_B$ fits on one bin and check if the rest of A and B fits on the other bin. If it doesn't, then the partition y_A and $S_A - y_A$ satisfies the adversarial partitioning condition since $S_A - (1+\epsilon)y_B$ was not able to fit on the second bin even after we optimistically allowed extra volume to fit on the first bin.

If, on the other hand, say no element $y_A \in L_A$ satisfies this condition. We must argue that no algorithm can partition A adversarially on bins of size $(1+\epsilon)S$. Assume for contradiction that there is some partition z_A^* and $S_A - z_A^*$ of A such that the remaining volume does not allow a partitioning of B. Therefore, there is no subset z_B^* of B such that $z_A^* + z_B^* \leq (1+\epsilon)S$ and $S_A - z_A^* + S_B - z_B^* \leq (1+\epsilon)S$

Since z_A^* is a subset of A, there must an item y_A in L_A such that $y_A \geq z_A^*/(1+\epsilon)$. Therefore, Algorithm 2 considers the partition y_A and $S - y_A$ and finds that there is a subset y_B such that $y_A + y_B \leq S$ and $S_A - y_A + S_B - (1+\epsilon)y_B \leq S$. Therefore, $(1+\epsilon)y_A + y_B \leq (1+\epsilon)S$ which implies that $z_A^* + y_B \leq (1+\epsilon)S$. In addition, $S_A - y_A + S_B - (1+\epsilon)y_B \leq S$ implies that $S_A - z_A^* + S_B - y_B \leq (1+\epsilon)S$. Since y_B and $S_B - y_B$ are partitions of B, this is a contradiction that the optimal algorithm can create an adversarial partition for bins of size $(1+\epsilon)S$ using z_A^*. □

```
L_A = ⟨0⟩
for i ← 1 to |A| do
    L_A ← merge(L_A, L_A + a_i, S)
    /* See Algorithm 1 for definition
     of function merge. */
    L_A ← trim(L_A, ε/(2|A|))
end
L_B = ⟨0⟩
for i ← 1 to |B| do
    L_B ← merge(L_B, L_B + b_i, S)
    L_B ← trim(L_B, ε/(2|B|))
end
for each y_A ∈ L_A do
    y_B ← the largest y_B ∈ L_B that is ≤ S − y_A
    /* S_A and S_B respectively denote
     the total sums of A and B. */
    if (S_A − y_A + S_B − (1 + ε) · y_B) > S then
        return success
        /* This partitioning of A
         prevents partitioning of B. */
    end
end
return failure
/* No partitioning of A prevents the
   partitioning of B. */
```

Algorithm 2: An FPTAS for ADVERSARIAL PARTITIONING.

B. A problem with a pseudo-polynomial time algorithm, but no FPTAS

In Section III-B, we had derived a pseudo-polynomial time schedulability analysis test for ADVERSARIAL PARTITIONING; in Section IV-A, we had used this test as a basis for obtaining an FPTAS for ADVERSARIAL PARTITIONING. Unfortunately, obtaining an FPTAS is not always as straightforward as first obtaining a pseudo-polynomial time test and then transforming it to an FPTAS: in this section we examine a scheduling problem that (i) is NP-complete (and hence computationally 'easier' than the Σ_2^{P}-complete ADVERSARIAL PARTITIONING problem); (ii) has a pseudo-polynomial time schedulability analysis test; but (iii) does not admit to an FPTAS (assuming $\mathsf{P} \neq \mathsf{NP}$). This scheduling problem relates to the memory-constrained tasks model [49], [50], in which we have implicit-deadline sporadic tasks that are each characterized by two parameters: a *processor utilization* that defines the fraction of the computing capacity of a processor that it needs, and a *code size* parameter that defines the amount of memory that must be reserved for its exclusive use.[5]

Memory-constrained task selection. We assume that there are several tasks, for each of which we have a choice of multiple implementations. Each implementation is specified in the memory-constrained tasks model [49], [50] discussed above, and hence is characterized by its own processor utilization parameter u_i and its code-size parameter s_i. As we did for the scheduling problem discussed just before Definition 1 in Section III, we assume that these parameters are *integer*-valued. This could, for instance, be because the parameters for each task are given as integer multiples of some basic unit of processor capacity and memory size respectively. We are required to choose one implementation for each task; the chosen implementations will be executed upon a single processor using preemptive EDF. Let I denote the set of chosen implementations. An overall cost bound $\mathcal{C}$ is specified, and the chosen implementations are required to satisfy the cost constraint

$$\left(\sum_{i \in I} u_i\right)^2 + \left(\sum_{i \in I} s_i\right)^2 \leq \mathcal{C}$$

Observe that $\left(\sum_{i \in I} u_i\right)$ in the LHS of this expression represents the total computing capacity needed, and $\left(\sum_{i \in I} s_i\right)$, the total memory needed. This cost constraint reflects the reality that cost of computing tends to be super-linear with capacity: for instance, it generally costs more than twice as much to purchase a processor that is twice as fast, or to buy memory twice as large, etc. (The precise choice of exponent — in the cost-constraint expression above, two — is not important for our purposes: our results below will hold for any exponent that is larger than one.)

Consider now the following problem definition, which is closely related to the problem we have discussed above:

Definition 5 (THE MEMORY-CONSTRAINED TASK SELECTION PROBLEM)**.**
Instance: A set of 3-tuples of non-negative integers (v_i, u_i, s_i), denoting respectively the *value*, utilization, and code-size parameter of an implementation, partitioned into N disjoint subsets. A cost bound $\mathcal{C}$. A target value $\mathcal{V}$.
Question: Can one 3-tuple from each partition be chosen such that

$$\left(\sum_{i \in I} u_i\right)^2 + \left(\sum_{i \in I} s_i\right)^2 \leq \mathcal{C}$$

and $\left(\sum_{i \in I} v_i\right) \geq \mathcal{V}$, where I denotes the set of N 3-tuples that are chosen? □

It is readily seen that the MEMORY-CONSTRAINED TASK SELECTION PROBLEM generalizes the problem we had discussed earlier in this section: we can map an instance of that earlier problem to an instance of the MEMORY-CONSTRAINED TASK SELECTION PROBLEM by

- for each task in the original problem instance, having a single partition in the MEMORY-CONSTRAINED TASK SELECTION PROBLEM instance containing all the different implementations of the task in the original problem instance, each assigned a value parameter 1 (i.e., the implementation

[5] Please see [49], [50] for a justification of this model and a discussion of its applicability in certain situation arising in embedded computing.

with utilization u_i and code-size s_i yields the 3-tuple $(1, u_i, s_i)$), plus an additional 3-tuple $(0, 0, 0)$;
- setting the cost bound for the MEMORY-CONSTRAINED TASK SELECTION PROBLEM instance equal to the cost bound for the original problem instance; and
- setting the value bound for the MEMORY-CONSTRAINED TASK SELECTION PROBLEM instance to be equal to *the number of tasks* in the original problem instance.

We now briefly describe a **pseudo-polynomial time algorithm** for solving the MEMORY-CONSTRAINED TASK SELECTION PROBLEM. Given an instance of the MEMORY-CONSTRAINED TASK SELECTION PROBLEM with 3-tuples partitioned into N partitions and cost bound and target value $\mathcal{C}$ and $\mathcal{V}$ respectively, this algorithm is essentially a dynamic program that constructs a table T with N rows, one corresponding to each partition. The columns in the table are labeled with ordered pairs (U, S) where U and S are non-negative integers satisfying $U^2 + S^2 \leq \mathcal{C}$. We point out that the number of columns in this table is no more than $\mathcal{C}$: for $U^2 + S^2 \leq \mathcal{C}$, we must have each of U and $S \leq \sqrt{\mathcal{C}}$ and hence there are at most $(\sqrt{\mathcal{C}})^2$, i.e., $\mathcal{C}$, columns.

The dynamic program seeks to fill in all the entries in the table to have $T[i, (U, S)] = V$, where *(i)* there is a choice of one 3-tuple each for each of the first i partitions for which the utilization parameters sum to U and the code-size parameters sum to S; and *(ii)* from amongst all such choices of 3-tuples, the choice with maximum cumulative value has cumulative value V. If there is no choice of one 3-tuple each for each of the first i partitions for which the utilization parameters sum to U and the code-size parameters sum to S, then $T[i, (U, S)]$ should be set equal to $-\infty$. This is achieved by first initializing each entry in the table to $-\infty$, and subsequently filling in the table row by row from top ($i = 1$) to bottom ($i = N$), in the following manner:

1) For the first row, $T[1, (U, S)] = V$ only for those columns with label (U, S) such that there is a 3-tuple in the first partition with utilization U and code-size parameter S.
2) To fill in the $(i + 1)$'th row for each $i \geq 0$, examine each entry the i'th row. If the column labeled (U, S) in this row has value $V \neq -\infty$, then for each 3-tuple (v, u, s) in the $(i + 1)$'th partition, set the column labeled $(U + u, S + s)$, if it exists, in the $(i + 1)$'th row to equal the maximum of the value it currently holds and $V + v$.

Once the table has been completed, we simply check whether the largest value in the last row of the table is $\geq \mathcal{V}$. As we have at most $\mathcal{C}$ columns and N rows, the algorithm runs in pseudo-linear time.

The non-existence of an FPTAS. Since FPTAS's are intended for optimization rather than decision problems, we need to first define an optimization version of the MEMORY-CONSTRAINED TASK SELECTION PROBLEM. The natural optimization version is to not pre-specify a value for $\mathcal{V}$, but rather require that one 3-tuple per partition be chosen in a manner that satisfies the cost constraint and maximizes the sum of the values of the chosen 3-tuples.

Above we had obtained a pseudo-polynomial time algorithm for the MEMORY-CONSTRAINED TASK SELECTION PROBLEM. Despite the existence of this pseudo-polynomial time algorithm, we will now show that there is no FPTAS for the MEMORY-CONSTRAINED TASK SELECTION PROBLEM, thereby establishing that simply demonstrating the existence of a pseudo-polynomial time algorithm for a schedulability analysis problem does not imply that an FPTAS exists for that problem.

In order to show that no FPTAS can exist for the MEMORY-CONSTRAINED TASK SELECTION PROBLEM, we make use of a similar non-existence result from [29]. The 2-WEIGHTED KNAPSACK PROBLEM is defined in [29] as follows: *"... the input consists of n triples of positive integers (p_k, v_k, w_k) and a positive integer W. The p_k are called profits, the v_k and w_k are called weights, and W is called the weight bound. The goal is to select an index set $K \subseteq \{1, 2, \ldots, n\}$ such that the selected weight obeys the weight bound*

$$\left(\sum_{k \in K} v_k\right)^2 + \left(\sum_{k \in K} w_k\right)^2 \leq W,$$

and such that the selected profit $\sum_{k \in K} p_k$ is maximized." It was shown in [29] that the 2-WEIGHTED KNAPSACK PROBLEM does not admit to an FPTAS:

Lemma 2 ([29, Lemma 8.2]). *Unless* P = NP, *the* 2-WEIGHTED KNAPSACK PROBLEM *does not have an FPTAS.*

Given an instance of the 2-WEIGHTED KNAPSACK PROBLEM, we define below a reduction to an instance of the MEMORY-CONSTRAINED TASK SELECTION PROBLEM:

> For each tuple (p_k, v_k, w_k) in the 2-WEIGHTED KNAPSACK PROBLEM instance, have a task with two implementations: one of which has value p_k, utilization v_k, and code-size w_k, and the other has value, utilization, and code-size all equal to zero. Set the cost bound $\mathcal{C}$ to be equal to the weight bound W of the 2-WEIGHTED KNAPSACK PROBLEM.

Given this reduction, it should be evident that there is an index set K for the 2-WEIGHTED KNAPSACK PROBLEM instance with total profit P if and only if there is a selection of one 3-tuple from each partition of the MEMORY-CONSTRAINED TASK SELECTION PROBLEM with cumulative value also equal to P; it therefore follows that the MEMORY-CONSTRAINED TASK SELECTION PROBLEM has an FPTAS if and only if the 2-WEIGHTED KNAPSACK PROBLEM has an FPTAS. But Lemma 2 above asserts that the the 2-WEIGHTED KNAPSACK PROBLEM does not haven an FPTAS; it therefore follows that the MEMORY-CONSTRAINED TASK SELECTION PROBLEM does not, either.

V. Context and Conclusions

Schedulability analysis is amongst the foundational cornerstones of real-time scheduling theory. The perspective of the real-time scheduling theory community as to what should be considered to be a tractable schedulability analysis problem has not remained static over time: as computing technology and algorithmic knowledge advance, our interpretation as to what is considered an efficient algorithm changes. We believe that recent developments in computing (with regards to both advances in algorithms, and the availability of computing hardware and software tools) call for another fresh examination of this issue; this manuscript has reported on some of the findings of such an examination that we have been conducting.

Among the findings we have sought to highlight are the proper place of *pseudo-polynomial* time algorithms. On the one hand, we have seen that pseudo-polynomial time algorithms exist for a far wider class of problems than one might initially think (while they are, to our knowledge, previously considered only for problems that are NP or coNP-complete in real-time scheduling theory, we have seen that pseudo-polynomial time algorithms may even exist for problems that are EXP-complete). On the other hand, we have argued that all pseudo-polynomial time algorithms should not be considered as being more-or-less equivalent from the perspective of tractability: while a strong case can be made that pseudo-linear time algorithms are indeed highly tractable for a range of schedulability analysis problems, it becomes a lot harder to make a similar case for pseudo-quadratic time, pseudo-cubic time, or pseudo-some-higher-power time algorithms. And amongst the class of pseudo-polynomial time algorithms, those for which the running time is not dependent on the units chosen for representing the numerical parameters are further particularly desirable – we refer to such algorithms as *robust* pseudo-polynomial time algorithms.

With respect to *approximation algorithms* where FPTAS's are widely accepted as being the desired ideal, we have similarly seen that FPTAS's can exist for harder problems than may be expected, by deriving an FPTAS for the Σ_2^{P}-complete ADVERSARIAL PARTITIONING problem. This positive observation is accompanied by a negative one illustrating a point first made in [29], that simply having a pseudo-polynomial (even pseudo-linear) time algorithm does not in itself imply the existence of an FPTAS — we have shown that the MEMORY-CONSTRAINED TASK SELECTION PROBLEM bears witness to this.

In closing, we point out that this manuscript is by no means giving any definitive answers to how we should approach tractability in real-time scheduling theory. Instead it should be viewed as an attempt to widen and nuance the view of tractability in this field, and as a complement to some other recent work in this vein. This recent work include [51], which applied the algorithmic technique of *fixed-parameter tractability* to real-time schedulability analysis problems, and [52], which instead of focusing, as is usual, on the tractability of *solving* schedulability problems, instead considered the tractability of *verifying* purported solutions. By nuancing the view of tractability we may find that tractable problems exist where we did not expect to find them, and vice versa.

Acknowledgements

This research was funded in part by the US National Science Foundation (Grants CNS-2141256, CPS-2229290, CCF-2106699, and CCF-2107280) and the Swedish Research Council grant 2018-04446.

References

[1] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[2] Michael Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[3] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. Rate monotonic scheduling: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, 2003.

[4] Joseph Y.-T Leung and M. Merrill. A note on the preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11:115–118, 1980.

[5] Joseph Y.-T Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[6] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, 1990. IEEE Computer Society Press.

[7] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems: The International Journal of Time-Critical Computing*, 2:301–324, 1990.

[8] F. Eisenbrand and T. Rothvoss. Static-priority real-time scheduling: Response time computation is NP-hard. In *Proceedings of the Real-Time Systems Symposium*, Barcelona, December 2008. IEEE Computer Society Press.

[9] Friedrich Eisenbrand and Thomas Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2010.

[10] Pontus Ekberg. *Models and Complexity Results in Real-Time Scheduling Theory*. PhD thesis, Uppsala University, 2015.

[11] Pontus Ekberg and Wang Yi. Fixed-priority schedulability of sporadic tasks on uniprocessors is NP-hard. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 139–146. IEEE Computer Society, 2017.

[12] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, October 1986.

[13] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium - 1989*, pages 166–171, Santa Monica, California, USA, December 1989. IEEE Computer Society Press.

[14] A. Wellings, M. Richardson, A. Burns, N. Audsley, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[15] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems: The International Journal of Time-Critical Computing*, 6:133–151, 1994.

[16] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, May 1995.

[17] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority preemptive scheduling: An historical perspective. *Real-Time Systems*, 8:173–198, 1995.

[18] S. Baruah, R. Howell, and L. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993.

[19] Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. In *Proceedings of the 17th Real-Time Systems Symposium*, Washington, DC, 1996. IEEE Computer Society Press.

[20] Sanjoy Baruah. A general model for recurring real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 114–122, Madrid, Spain, December 1998. IEEE Computer Society Press.

[21] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 71–80, Chicago, 2011. IEEE Computer Society Press.

[22] Wei Zheng, Qi Zhu, Marco Di Natale, and Alberto Sangiovanni Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 161–170, 2007.

[23] Sanjoy Baruah and Enrico Bini. Partitioned scheduling of sporadic task systems: an ilp-based approach. In *Proceedings of the 2008 Conference on Design and Architectures for Signal and Image Processing*, 2008.

[24] Sanjoy Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. Ilp-based approaches to partitioning recurrent workloads upon heterogeneous multiprocessors. In *Proceedings of the 2016 28th EuroMicro Conference on Real-Time Systems*, ECRTS '16, Toulouse (France), 2016. IEEE Computer Society Press.

[25] Sanjoy K. Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. *Journal of Scheduling*, Dec 2018.

[26] S. Baruah. An ILP representation of a DAG scheduling problem. *Real-Time Systems: The International Journal of Time-Critical Computing*, 2021.

[27] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[28] Gerhard J Woeginger. The trouble with the second quantifier. *4OR: A Quarterly Journal of Operations Research*, 19(2):157–181, 2021.

[29] Gerhard J. Woeginger. When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (FPTAS)? *INFORMS Journal on Computing*, 12(1):57–74, 2000.

[30] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 187–195, Catania, Sicily, July 2004. IEEE Computer Society Press.

[31] Nathan Fisher. *The Multiprocessor Real-Time Scheduling of General Task Systems*. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, 2007.

[32] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[33] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[34] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

[35] J Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[36] Steven Homer. Structural properties of complete problems for exponential time. *Complexity Theory: Retrospective II*, 2:135, 1997.

[37] P. Ekberg and W. Yi. Uniprocessor feasibility of sporadic tasks remains coNP-complete under bounded utilization. In *2015 IEEE Real-Time Systems Symposium*, pages 87–95, 2015.

[38] Berit Johannes. *New Classes of Complete Problems for the Second Level of the Polynomial Hierarchy*. PhD thesis, Technischen Universität Berlin, 2011.

[39] Pontus Ekberg and Wang Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 135–144, jul 2012.

[40] A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 78–87, Dec 2013.

[41] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the 32nd Real-Time Systems Symposium (RTSS)*, pages 34 –43, 2011.

[42] Sanjoy Baruah and Pontus Ekberg. Graceful Degradation in Semi-Clairvoyant Scheduling. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[43] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 214–223, Los Alamitos, October 1995. IEEE Computer Society Press.

[44] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 140–149, El Paso, Texas, 4–6 May 1997.

[45] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 37(4):617–643, 2000.

[46] Giorgio Ausiello, Alberto Marchetti-Spaccamela, Pierluigi Crescenzi, Giorgio Gambosi, Marco Protasi, and Viggo Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer Verlag, 1999.

[47] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin-Heidelberg-New York-Barcelona-Hong Kong-London-Milan-Paris-Singapur-Tokyo, 2001.

[48] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, third edition, 2009.

[49] Sanjoy Baruah and Nathan Fisher. A dynamic-programming approach to task partitioning among memory-constrained multiprocessors. In *Proceedings of the International Conference on Real-time Computing Systems and Applications*, Gothenburg, Sweden, August 2004. Springer-Verlag.

[50] N. Fisher, J. Anderson, and S. Baruah. Task partitioning upon memory-constrained multiprocessors. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 416–421, Hong Kong, August 2005. IEEE Computer Society Press.

[51] Sanjoy Baruah, Pontus Ekberg, and Abhishek Singh. Fixed-parameter analysis of preemptive uniprocessor scheduling problems. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society Press, 2022.

[52] Sanjoy Baruah and Pontus Ekberg. Towards efficient explainability of schedulability properties in real-time systems. In *Proceedings of the 35th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 2:1–2:20, 2023.