Elastic Scheduling for Parallel Real-Time Systems*

James Orr
Washington University in St. Louis, 1 Brookings Dr, St. Louis, MO 63130, USA
james.orr@wustl.edu

Chris Gill
Washington University in St. Louis, 1 Brookings Dr, St. Louis, MO 63130, USA
cdgill@wustl.edu

Kunal Agrawal
Washington University in St. Louis, 1 Brookings Dr, St. Louis, MO 63130, USA
kunal@wustl.edu

Jing Li
New Jersey Institute of Technology, University Heights, Newark, NJ 07102, USA
jingli@njit.edu

Sanjoy Baruah
Washington University in St. Louis, 1 Brookings Dr, St. Louis, MO 63130, USA
baruah@wustl.edu

Abstract

The elastic task model was introduced by Buttazzo et al. in order to represent recurrent real-time workloads executing upon uniprocessor platforms that are somewhat flexible with regards to timing constraints. In this work, we propose an extension of this model and apply it to represent recurrent real-time workloads that exhibit internal parallelism and are executed on multiprocessor platforms. In our proposed extension, the *elasticity coefficient* – the quantitative measure of a task's elasticity that was introduced in the model proposed by Buttazzo et al. – is interpreted in the same manner as in the original (sequential) model. Hence, system developers who are familiar with the elastic task model in the uniprocessor context may use our more general model as they had previously done, now for real-time tasks whose computational demands require them to utilize more than one processor.

2012 ACM Subject Classification Software and its engineering → Real-time schedulability, Computer systems organization → Real-time system architecture, Computer systems organization → Real-time system specification, Computer systems organization → Embedded software

Keywords and Phrases Parallel real-time tasks, multiprocessor federated scheduling, elasticity coefficient

Digital Object Identifier 10.4230/LITES-v006-i001-a005

Received 2018-09-24 **Accepted** 2019-03-08 **Published** 2019-05-14

1 Introduction

Advances in parallel real-time scheduling theory and concurrency platforms over the last couple of decades have allowed for previously unachievable combinations of high computational demands and fine-grained time-scales, in high-performance parallel real-time applications such as those in autonomous vehicles [13] and real-time hybrid simulation systems [9, 11]. However, current parallel real-time systems usually assign parallel tasks to fixed sets of processors and release them at statically determined periodic rates [13, 9, 10]. For systems that need to adjust individual tasks' computational requirements at run-time (e.g., control algorithms with multiple modes of

* This research was supported in part by NSF grant CCF-1337218 titled "XPS: FP: Real-Time Scheduling of Parallel Tasks" and NSF CNS1814739 titled "Dynamically Customizable Safety Critical Embedded Systems."

operation), current approaches may need to incorporate excessive pessimism to support such forms of dynamic and adaptive resource allocation.

The *elastic task model* was introduced in [4] with the specific aim of providing dynamic flexibility during run-time. The model is derived from an analogy to the expansion and contraction of a contiguous collection of springs when a common force is applied to them all, in order to bring their cumulative length down below a specified bound. The computational demand of a task is analogous to the length of a spring, and the available computational capacity to the bound on the cumulative length of the springs (see [4] for details).

In the elastic task model, each recurrent task is characterized by a worst-case execution time (WCET), lower and upper bounds on the values that the task period parameter may take, and an '*elasticity coefficient*' that represents the flexibility of the task (relative to other tasks) to reduce its run-time computational demand by increasing its effective period. Given a system comprising a collection of such tasks executing upon a shared processor, the elastic scheduling algorithm seeks to choose a value for each task's period parameter within the task's specified range, such that the overall system is schedulable.

The elastic task model was originally defined for task systems such as multimedia systems, control systems, and ad-hoc communication networks implemented on preemptive uniprocessors [1, 8, 5]. However, today's high-performance real-time applications (e.g. real-time hybrid simulation [9, 11]) must often execute upon multiprocessor platforms so as to be able to exploit internal parallelism of these tasks across multiple processors to meet high computational demand. Therefore, the original elastic task model, as well as algorithms that were developed by Buttazzo et al. [4, 5] along with accompanying schedulability analysis and run-time scheduling techniques, need to be appropriately extended in order to be useful for these kinds of high-performance real-time applications. In this paper, we consider multiprocessor scheduling under the *federated scheduling* paradigm (in which each task whose computational demand exceeds the capacity of a single processor is granted exclusive access to multiple processors); we propose a parallel multiprocessor extension to the elastic task model, and provide appropriate algorithms for federated schedulability analysis and federated scheduling of systems represented using our proposed model.

The central idea of elastic scheduling, originally defined by Buttazzo et al. [4], is that if the overall computational demand of a system exceeds the capacity of the implementation platform to accommodate it all, then individual tasks' computational demands are reduced and the available platform capacity is allocated in a flexible manner to accommodate these reduced demands. Upon multiprocessor platforms, there are several different interpretations possible, as to what an *elastic* manner of distributing the processors may mean. Our proposed extension aligns with earlier work in the sense that we are interpreting the elasticity coefficient parameters according to the semantics assigned to them in the uniprocessor context. We believe that this is a critical issue: the elasticity parameters characterize the relative flexibility –the 'hard-real-time'ness– of the tasks, and should bear common interpretation regardless of whether implemented on uni- or multi-processors. We, therefore, believe that the preservation of this interpretation is one of the major benefits of our extended model.

The remainder of this paper is organized in the following manner. We briefly provide some needed background and related work concerning the elastic task model and federated scheduling in Section 2; and describe the parallel workload model we are proposing for the representation of parallel elastic tasks. In Section 3 we present a relatively simple and efficient algorithm for scheduling such tasks upon multiprocessor platforms, which preserves the semantics that were intended for elastic tasks in the uniprocessor context. We also point out how this simple approach may result in an unnecessary degree of platform resource under-utilization. In Section 4 we propose an alternative approach that is able to make more efficient use of the platform to provide

a superior scheduling solution, at the cost of not being as faithful to the semantics of elasticity as originally defined for the uniprocessor case. We conclude in Section 5 with a brief summary, and place this work within a larger context of ongoing research efforts towards achieving dynamic flexibility in multiprocessor scheduling of parallelizable workloads.

2 Background, Related Work, and Task Model

In this paper, we extend the definition and applicability of real-time elastic scheduling to parallel real-time systems. We start out in this section by providing some background on both the elastic task model and the federated paradigm of parallel real-time scheduling on multiprocessor platforms. Doing so enables us to define our proposed elastic model for the federated scheduling of systems of parallel real-time tasks.

2.1 The Elastic Task Model

The elastic task model was first proposed by Buttazzo et al. in [4]. Tasks in this model may dynamically adapt their periods in response to system behavior, in order to keep system-wide utilization below a user-specified desired value U_d (which may be at or below a scheduling algorithm's threshold, e.g., 1.0 for preemptive uniprocessor EDF scheduling). The task model is a generalization of the implicit-deadline sporadic task model [15]: each task $\tau_i = (C_i, T_i^{(\min)}, T_i^{(\max)}, E_i)$ is characterized by a worst-case execution requirement C_i, a minimum (and preferred) period $T_i^{(\min)}$, a maximum period $T_i^{(\max)}$, and an elastic coefficient E_i that quantitatively characterizes how amenable a task is to a change in its period (similar to a measure of a spring's resistance to changes in length). A higher elastic coefficient implies a more elastic task, which is more willing to adapt its period. Any task τ_i that should not vary its period (and therefore its utilization) at all can set $T_i^{(\min)} = T_i^{(\max)}$, and τ_i will act like an ordinary (i.e., not elastic) implicit-deadline sporadic task with WCET C_i and period $T_i^{(\min)}$. An actual period must be assigned to each task; a task's assigned period is denoted as T_i and must fall within the range $[T_i^{(\min)}, T_i^{(\max)}]$. Furthermore, a task τ_i is considered to have an *implicit deadline* where the *relative deadline* D_i of τ_i is equal to its actual period, i.e., $D_i{=}T_i$.

Recall that the *utilization* U_i of an (ordinary – not elastic) implicit-deadline task $\tau_i = (C_i, T_i)$ is defined to be the ratio of its WCET to its period ($U_i = C_i/T_i$), and that the utilization $U(\Gamma)$ of an implicit-deadline sporadic task system $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is the sum of the utilizations of all the tasks in the system ($U(\Gamma) = \sum_{\tau_i \in \tau} U_i$). Buttazzo et al. have derived an iterative algorithm in [4] for task compression which (if possible) finds a way to assign each task τ_i in a system Γ of elastic tasks a period T_i in a manner that is compliant with the semantics of spring compression, such that $\sum_i (C_i/T_i) \leq U_d$ and $T_i^{(\min)} \leq T_i \leq T_i^{(\max)}$ for all tasks τ_i. (As stated above, U_d is a user-defined threshold, perhaps according to the scheduling algorithm that is used, e.g., $U_d = 1$ is suitable for preemptive EDF scheduling.)

Since the introduction of the elastic task model, the uniprocessor version has been expanded to include constrained deadlines [8], resource sharing [5] and unknown computational load [6]. We leave their parallel extensions as future work.

2.2 Federated Scheduling and Parallel Real-Time Task Model

Federated scheduling is a parallel real-time scheduling paradigm that was proposed by Li et al. [14] for scheduling collections of recurrent parallel tasks upon multiprocessor platforms, when one or more individual tasks may have a computational requirement that exceeds the capacity of a single processor to entirely accommodate it. Under federated scheduling, such tasks (i.e., those with

computational requirement exceeding the capacity of a single processor) are granted exclusive access to a subset of processors; the remaining tasks execute upon a shared pool of processors.

In parallel real-time task systems, the computational requirement of a task τ_i (the generalization of the WCET parameter for sequential tasks) is represented by the following two parameters:

1. The ***work*** parameter C_i denotes the cumulative worst-case execution time of all the parallel branches that are executed across all processors. Note that for deterministic parallelizable code (e.g., as represented in the sporadic DAG tasks model [2]; see [3, Chapter 21] for a textbook description) this is equal to the worst-case execution time of the code on a single processor (ignoring communication overhead from synchronizing processors).
2. The ***span*** parameter L_i denotes the maximum cumulative worst-case execution time of any sequence of precedence-constrained pieces of code. It represents a lower bound on the duration of time the code would take to execute, regardless of the number of processors available.
   The span of a program is also called the *critical-path length* of the program, and a sequence of precedence-constrained pieces of code with cumulative worst-case execution time equal to the span is a *critical path* through the program.

Algorithms are known for computing the *work* and *span* of a task represented as a DAG, in time linear in the DAG representation. The relevance of these two parameters arises from well-known results in scheduling theory concerning the multiprocessor scheduling of precedence-constrained jobs (i.e., DAGs) to minimize makespan. This problem has long been known to be NP-hard in the strong sense [16]; i.e., computationally highly intractable. However, Graham's *list scheduling* algorithm [12], which constructs a work-conserving schedule by executing at each instant in time an available job, if any are present, upon any available processor, performs fairly well in practice.

An upper bound on the makespan of a schedule generated by list scheduling is easily stated. Given the *work* and *span* of the DAG being scheduled, it has been proved in [12] that the makespan of the schedule for a given DAG upon m processors is guaranteed to be no larger than

$$\frac{work - span}{m} + span \tag{1}$$

Thus, a good upper bound on the makespan of the list-scheduling generated schedule for a DAG may be stated in terms of only its work and span parameters. Equivalently, if the DAG represents a real-time piece of code characterized by a relative deadline parameter D, $(\frac{work-span}{m} + span) \leq D$ is a sufficient test for determining whether the code will complete by its deadline upon an m-processor platform.

A parallel task τ_i is considered to be a *high-utilization task* if its *utilization* $U_i = \frac{C_i}{T_i} > 1$ and a *low-utilization task* otherwise. Each high-utilization task τ_i receives m_i dedicated processors on which to run; for implicit-deadlines tasks, we need the resulting makespan to be less than or equal to $D_i = T_i$; i.e.

$$\begin{aligned} & \frac{C_i - L_i}{m_i} + L_i \leq T_i \\ \Leftrightarrow \quad & \frac{C_i - L_i}{m_i} \leq T_i - L_i \\ \Leftrightarrow \quad & m_i \geq \frac{C_i - L_i}{T_i - L_i} \end{aligned}$$

Under federated scheduling, since the number of processors assigned to each high-utilization task is an integer, we therefore have

$$m_i = \left\lceil \frac{C_i - L_i}{T_i - L_i} \right\rceil . \tag{2}$$

Under the original federated scheduling model in [14], low-utilization tasks are treated as sequential and are scheduled using existing mechanisms such as global or partitioned EDF scheduling.

In this paper, we will consider the federated scheduling of task systems with elastic sporadic parallel tasks. Recall that each elastic task has a range of acceptable periods within the rage $[T_i^{(\min)}, T_i^{(\max)}]$. Let $U_i^{(\max)} = C_i/T_i^{(\min)}$ and $U_i^{(\min)} = C_i/T_i^{(\max)}$ denote the maximum (i.e., desired) and the minimum acceptable utilization for τ_i. Note that it is possible for some tasks to be either high-utilization or low-utilization depending on the selected period. We refer to these as tasks with *hybrid-utilization*. (Formally hybrid-utilization tasks are tasks such that $T^{(\min)} \leq C_i \leq T^{(\max)}$.) Scheduling of exclusively low-utilization elastic tasks is easily done via minor extensions to prior results [4, 5, 7, 8]. We therefore do not consider them for the remainder of this paper. Instead, *henceforth we consider only the scheduling of exclusively high-utilization tasks*. That is, we will consider a system $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n elastic parallel high-utilization tasks that is to be scheduled under federated scheduling upon m processors. We consider this to be a necessary and non-trivial step towards the scheduling of hybrid-utilization tasks, the treatment of which we leave for future work.

In the remainder of this paper we will often represent a task $\tau_i = (C_i, L_i, U_i^{(\max)}, U_i^{(\min)}, E_i)$ by its work and span parameters, its maximum and minimum utilizations,[1] and its elasticity coefficient. We will seek to compute m_i, the number of processors that are to be devoted to the exclusive use of task τ_i, for each τ_i such that $\sum_{i=1}^{n} m_i \leq m$.

3 A first attempt at elastic scheduling of parallel tasks

It is fairly straightforward to show that the desired elasticity property on the tasks that were defined in the original (uniprocessor) elastic tasks model [4] is that

$$\forall\ i, j, \quad \left(\frac{U_i^{(\max)} - U_i}{E_i}\right) = \left(\frac{U_j^{(\max)} - U_j}{E_j}\right) \tag{3}$$

That is, the elasticity coefficient E_i of task τ_i is a scaling factor on the amount by which it may have its actual utilization reduced from the desired value of $U_i^{(\max)}$.

We use λ to denote the desired equilibrium value for all tasks demonstrated in Expression (3); for all tasks $\lambda = \left((U_i^{(\max)} - U_i)/E_i\right)$. Expression (3) suggests that

$$U_i \leftarrow U_i^{(\max)} - \lambda E_i$$

However, we also require $U_i \geq U_i^{(\min)}$; hence for a given value of λ we choose

$$U_i(\lambda) \leftarrow \max\left(U_i^{(\max)} - \lambda E_i, U_i^{(\min)}\right) \tag{4}$$

Equation (4) suggests an algorithm for the federated scheduling of parallel task system $\Gamma = \{\tau_1, \ldots, \tau_n\}$ upon m processors. It is evident from visual inspection of Equation (4) that the 'best' schedule – the one that compresses tasks' utilizations the least amount necessary in order to achieve schedulability – is the one for which λ is the smallest. Now for a given value of λ, Algorithm 1 can determine, in time linear in the number of tasks, whether the task system can be scheduled upon the m available processors using federated scheduling.

[1] Note that representing the task by its maximum and minimum utilizations is equivalent to representing it by its minimum and maximum periods, since given C_i, one set of parameters can be derived from the other set.

Algorithm 1 Elastic-1(Γ, m, λ).

▷ Γ is the task system and m the number of processors that are available
▷ λ is the compression factor permitted
$m' \leftarrow 0$ ▷ Number of processors needed
for ($\tau_i \in \Gamma$) **do**
  $U_i = \max\left(U_i^{(\max)} - \lambda E_i, U_i^{(\min)}\right)$ ▷ See Eqn 4
  $T_i = C_i/U_i$
  $m_i = \lceil (C_i - L_i)/(T_i - L_i) \rceil$
  $m' \leftarrow m' + m_i$
end for
if ($m' > m$) **then** ▷ Not enough processors.
  return UNSCHEDULABLE
else
  return $\langle m_1, m_2, \ldots, m_n \rangle$ ▷ τ_i gets m_i processors
end if

Note the value of λ can be bounded to the range of $[0, \phi]$ where $\lambda = 0$ represents all tasks receiving their maximum utilizations and ϕ is the maximum value among all tasks of the equation $\left(\frac{U_i^{(\max)} - U_i^{(\min)}}{E_i}\right)$. $\lambda = \phi$ thus represents all tasks receiving their minimum utilization. By bounding the potential values of λ, we can use binary search within this range and make repeated calls to Algorithm 1 and thereby determine, to any desired degree of accuracy, the smallest value of λ for which the system is schedulable.

3.1 Discussion

Semantics-preservation. Algorithm 1 for the federated scheduling of parallel elastic tasks that we have presented above is *semantics preserving* in the following sense: the assignment of actual period values to the tasks (the T_i's) is done in accordance with Equation (4), which is the same manner in which periods are assigned in uniprocessor scheduling of elastic tasks. Hence the system developer who seeks to use our proposed elastic task model to implement flexible parallel tasks upon multiprocessor platforms need not 'learn' new (or additional) semantics for the elasticity coefficient: this coefficient means exactly the same thing in the parallel multiprocessor case as it did in the system designer's previous experiences with sequential uniprocessor tasks (the value of this parameter for each task is a relative measure of its degrees of tolerance to having its period increased and its computational demand thereby reduced).

Run-time platform capacity under-utilization. Despite these advantages, however, one can identify two sources of resource under-utilization by Algorithm 4.

- First, observe that the number of processors assigned to a task must be *integral*, and is hence equal to the ceiling of an expression. If the expression $(C_i - L_i)/(T_i - L_i)$, which lies within the ceiling operator ($\lceil \cdot \rceil$) when computing the number of processors assigned to task τ_i, is not itself an integer, then one could further reduce the actual period (the T_i value) that is assigned to the task τ_i and thereby assign τ_i more computational capacity than is afforded by Algorithm 1. However, we do not permit this to happen since the resulting assignment may no longer be semantics-preserving in the sense that different tasks may see a reduction in allocated capacity that is not consistent with their relative elasticity coefficients. This difference between $\lceil (C_i - L_i)/(T_i - L_i) \rceil$ and $(C_i - L_i)/(T_i - L_i)$ is thus 'wasted' capacity.

- Second, consider the case with two identical elastic tasks, and an odd number of processors. Semantics-preservation dictates that both tasks be treated in the same manner; however, doing so would correspond to assigning the same number of processors to each task and therefore leaving one processor unused. More generally, Algorithm 1 may leave up to $n-1$ processors unallocated to n identical tasks.

Thus, the simple semantics-preserving scheme presented in this section may under-utilize platform resources. In Section 4 we discuss an alternative scheme that makes more efficient use of platform capacity at the cost of additional complexity in the semantics of elasticity.

4 More resource-efficient scheduling

The notion of semantic preservation with uniprocessor elastic task scheduling presented in Section 3 is simple and intuitive, and very strong: the elasticity coefficient of a task directly indicates the task's tolerance to having its period parameter increased. However, as we saw, remaining faithful to such a strong notion of semantic equivalence comes at the cost of some computing capacity loss and cannot guarantee full utilization of a platform's computing capacity. We now consider a more generalized interpretation of the semantics of uniprocessor elastic tasks. This interpretation was provided by Chantem et al. [8], who proved that the algorithm of Buttazzo et al. [5] for scheduling sequential elastic tasks upon preemptive uniprocessors is equivalent to solving the following constrained optimization problem:

$$\textbf{minimize} \sum_{i=1}^{n} \frac{1}{E_i} \left(U_i^{(\max)} - U_i\right)^2 \tag{5}$$

such that:

$$U_i^{(\min)} \leq U_i \leq U_i^{(\max)} \quad \text{for all } \tau_i, \text{ and}$$
$$\sum_{i=1}^{n} U_i \leq U_d$$

where U_d is the desired system utilization. We believe that this is a somewhat less natural interpretation of elasticity in task scheduling than the interpretation considered in Section 3: it is unlikely that a typical system designer is thinking of the elasticity coefficients (the E_i parameters) that they assign to the individual tasks as coefficients to a quadratic optimization problem. Nevertheless, we adopt this notion of elastic interpretation in this section; for this interpretation, we are able to derive a federated scheduling algorithm that makes far more efficient use of platform computing capacity than was possible under the earlier more intuitive interpretation considered in Section 3.

Note that sequential elastic task scheduling only considers CPU utilization when attempting to schedule tasks on a single processor. Specifically, system-wide utilization $\sum_{i=1}^{n} U_i$ must stay below a desired utilization U_d at all times in order to maintain schedulability. As such, task utilizations are decreased by (when possible) increasing individual task periods in proportion to their fraction of system-wide elasticity until either (1) an acceptable schedule is found such that $\sum_{i=1}^{n} U_i \leq U_d$ or (2) each task τ_i has period $T_i = T_i^{(\max)}$ with $\sum_{i=1}^{n} U_i > U_d$. If a schedule cannot be found the taskset is declared unschedulable.

In federated scheduling of high-utilization tasks, however, system schedulability is no longer a function only of cumulative utilization but rather whether n tasks can be successfully scheduled on m cores. We now give an algorithm for determining processor allocation and schedulability of a task system that allocates the processors *one at a time* to the tasks, Algorithm 2. Algorithm 2

Algorithm 2 Task_compress_par(Γ, m).

for ($\tau_i \in \Gamma$) **do**
  $m_{i_{min}} = \lceil (C_i - L_i)/T_{i_{max}} - L_i) \rceil$ ▷ Minimum number of processors
  $m_{i_{max}} = \lceil (C_i - L_i)/(T_{i_{min}} - L_i) \rceil$ ▷ Maximum number of processors
  $m_i = m_{i_{min}}$
  while $m_i <= m_{i_{max}}$ **do** ▷ Compute the shortest period for τ_i
    ▷ for each possible value of m_i
    $T_{(i,m_i)} = (C_i - L_i)/(m_i) + L_i$ ▷ $T_{(i,m_i)}$ = shortest with m_i processors
    $m_i = m_i + 1$
  end while
  $m_i = m_{i_{min}}$ ▷ Assign minimum number of processors
  $T_i = T_{(i,m_i)}$ ▷ Assign corresponding shortest period
  $m = m - m_{i_{min}}$ ▷ m keeps count of processors remaining
end for
if ($m < 0$) **then** ▷ There weren't enough processors
  return UNSCHEDULABLE
else if ($m == 0$) **then**
  return *processor allocation with m_i values*
end if

The remainder of this pseudocode
allocates processors one at a time

for ($\tau_i \in \Gamma$) **do**
  Determine δ_i, the potential
  decrease to Problem 7 for each task
end for

Make a max heap of all tasks, with the δ_i values as the key

while m > 0 and heap not empty **do** ▷ Assign remaining processors
  $\tau_{most} = heap.pop()$ ▷ Task that would most benefit
  $m_{most} = m_{most} + 1$ ▷ Permanently assign processor
  $m = m - 1$
  $T_{most} = T_{(most,m_{most})}$
  if ($m > 0$ and $m_{most} < m_{most_{max}}$) **then** ▷ Able to receive more processors?
    Determine δ_{most}, the potential
    decrease to Problem 7 for task τ_{most}
    Reinsert τ_{most} into heap
  end if
end while
return *the processor allocation with m_i values*

starts out by determining, for each task τ_i, the minimum number of processors $m_{i_{min}}$ needed to be meet its minimum acceptable computational load (i.e., having $T_i \leftarrow T_i^{(\max)}$) in Line 2, and the number $m_{i_{max}}$ needed to meet its desired computational load (i.e., having $T_i \leftarrow T_i^{(\min)}$) in Line 3. Since the assigned period T_i satisfies $T_i^{(\min)} \leq T_i \leq T_i^{(\max)}$, the actual number of CPUs m_i assigned to τ_i is also bounded by $m_{i_{min}} \leq m_i \leq m_{i_{max}}$.

Because of the ceiling function in Equation (2), each range of values for T_i maps to a given m_i for each task. In this work we assume that it is beneficial for each task to run as frequently as possible. As such, we assign task τ_i the minimum period T_i available on m_i allocated processors. We denote this period value as $T_{(i,m_i)}$, which is derived directly from Equation (2):

$$T_{(i,m_i)} = \frac{C_i - L_i}{m_i} + L_i \tag{6}$$

All possible values of $T_{(i,m_i)}$ for $m_{i_{min}} \leq m_i \leq m_{i_{max}}$ are computed first and stored in lookup tables. This is accomplished during the while loop (Lines 5–9) in Algorithm 2.

Next (Lines 10–12), each task is assigned the minimum number of processors it needs, and this number of processors is subtracted from m; hence at the end of the loop, m denotes the number of processors remaining for additional assignment (above and beyond the minimum needed per task). If $m < 0$ the instance is unschedulable, while if $m = 0$ there is nothing more to be done – the system is schedulable with each task receiving its minimum level of service. These conditions are tested in Lines 14–18 of the pseudocode.

If $m > 0$, however, we will individually assign each of these remaining m processors to whichever task would benefit 'the most' from receiving it. This is determined in the following manner. Similar to scheduling sequential tasks [8], our goal is to find task utilizations (and therefore periods) that solve the optimization problem:

$$\textbf{minimize} \sum_{i=1}^{n} \frac{1}{E_i} \left(U_i^{(\max)} - U_i\right)^2 \tag{7}$$

such that:

$$U_i^{(\min)} \leq U_i \leq U_i^{(\max)} \text{ for all } \tau_i, \text{ and}$$
$$\sum_{i=1}^{n} m_i \leq m$$

In allocating each processor we calculate, for each task τ_i, a quantity δ_i which represents the *decrease* in $\frac{1}{E_i}\left(U_i^{(\max)} - U_i\right)^2$ if the next processor were to be allocated to task τ_i – this is done in Lines 23–26 of Algorithm 2. We then assign the processor to whichever task would see the biggest decrease. (As a consequence, the objective of the optimization problem 7 would decrease the most.) To accomplish this efficiently, we

- Place the tasks in a max heap indexed on the value of δ_i (Line 28); and
- while there are unallocated processors and the heap is not empty (checked in Line 30)
  - assign the next processor to the task at the top of the heap (Lines 31–34) and, if this task is eligible to receive more processors (checked in Line 35), recompute δ_i for this task (Line 36) and reinsert into the heap (Line 38).

Run-time complexity. The first for-loop in the algorithm (Lines 1–13 in the pseudocode listing in Algorithm 2) takes $\Theta(m{*}n)$ time. The for-loop in Lines 23–26 and the making of the max heap (Line 28) each take $\Theta(n)$ time. The running time of the remainder of the algorithm (Lines 30–40) is dominated by the max-heap operations; the overall running time is therefore $\Theta(n * m + m \log n)$.

4.1 Proof of Optimality

In this section we prove in Theorem 3 that Algorithm 2 solves the optimization problem given in Equation (7) optimally. The optimality of Algorithm 2 then follows from the result of Chantem et al. [8] showing the equivalence of uniprocessor elastic scheduling of sequential tasks with the optimization problem given in Equation (5).

The dependency amongst the three results in this section – Lemma 1, Lemma 2, and Theorem 3 – is strictly linear: Lemma 1 is needed to prove Lemma 2, which is needed to prove Theorem 3.

▶ **Lemma 1.** *The utilization U_i of elastic task τ_i strictly increases towards maximum utilization as the number of processors m_i assigned to it increases.*

Proof. Since $U_i = C_i/T_i$, (and C_i is constant), U_i increases as T_i decreases. By Equation (6), $T_i = ((C_i - L_i)/m_i) + L_i$. C_i and L_i are constant for task τ_i. Therefore, T_i strictly decreases as m_i increases. Therefore, an increase of m_i decreases T_i and increases U_i. ◀

▶ **Lemma 2.** *In assigning processors one at a time (in the while loop of Lines 30–40 of Algorithm 2), the consecutive assignment of the $(k+1)$'st and $(k+2)$'nd to the same task τ_i with k currently assigned processors will result in diminishing returns of δ_i, the decrease in $\frac{1}{E_i}(U_i^{(\max)} - U_i)^2$ for τ_i. (i.e., the benefit of assigning a processor to a task is never as high as the already-incurred benefit of assigning prior processors.)*

Proof. This is readily observed by algebraic simplification.[2] Let x_k be the value of $\frac{1}{E_i}(U_i^{(\max)} - U_{i_k})^2$ where U_{i_k} is the task utilization with k processors. Let x_{k+1} be the value of $\frac{1}{E_i}(U_i^{(\max)} - U_{i_{k+1}})^2$ with new utilization $U_{i_{k+1}}$ after assigning processor $k+1$ to τ_i, and similarly let x_{k+2} be the value of $\frac{1}{E_i}(U_i^{(\max)} - U_{i_{k+2}})^2$ with new utilization $U_{i_{k+2}}$ after subsequently assigning processor $k+2$ to τ_i. From Lemma 1, we know that $U_{i_k} < U_{i_{k+1}} < U_{i_{k+2}}$.

Define the benefit of adding processor $k+1$ to τ_i as $\delta_{i_{k+1}} = x_k - x_{k+1}$, and the later benefit of assigning processor $k+2$ as $\delta_{i_{k+2}} = x_{k+1} - x_{k+2}$. To prove diminishing returns, we must show that $\delta_{i_{k+1}} > \delta_{i_{k+2}}$.

Note that the math is equivalent, so we temporarily ignore the constant scalar $\frac{1}{E_i}$. Thus, both

$$\delta_{i_{k+1}} = (U_i^{(\max)} - U_{i_k})^2 - (U_i^{(\max)} - U_{i_{k+1}})^2 \tag{8}$$

and

$$\delta_{i_{k+2}} = (U_i^{(\max)} - U_{i_{k+1}})^2 - (U_i^{(\max)} - U_{i_{k+2}})^2 \tag{9}$$

are of the form

$$(x-z)^2 - (x-y)^2 \tag{10}$$

where $x > y > z$. We can therefore say that $z + \alpha = y$ and $y + \beta = x$.

Re-stating Equation (10) in terms of z, α, and β, we obtain:

$$(z+\alpha+\beta-z)^2 - (z+\alpha+\beta-z-\alpha)^2$$

which simplifies to

$$\alpha^2 + 2\alpha\beta. \tag{11}$$

[2] The algebra, while straightforward, is rather tedious and the reader may choose to just skim it at first reading.

Therefore, to prove $\delta_{i_{k+1}} > \delta_{i_{k+2}}$, it is sufficient to show that

$$\alpha_{k+1}^2 + 2\alpha_{k+1}\beta_{k+1} > \alpha_{k+2}^2 + 2\alpha_{k+2}\beta_{k+2} \tag{12}$$

where α_{k+1}, β_{k+1}, α_{k+2}, and β_{k+2} are $(U_{i_{k+1}} - U_{i_k})$, $(U_i^{(\max)} - U_{i_{k+1}})$, $(U_{i_{k+2}} - U_{i_{k+1}})$, $(U_i^{(\max)} - U_{i_{k+1}})$, respectively. (These values come from the definitions of α and β and the substitutions of x, y, and z in Equation (10) into their actual values from Equations 8 and 9.) Note that as α_{k+1}, β_{k+1}, α_{k+2}, and β_{k+2} are all positive numbers, Equation (12) will be satisfied if we can individually prove $\alpha_{k+1} > \alpha_{k+2}$ and $\beta_{k+1} > \beta_{k+2}$, which we now proceed to do.

We first prove $\beta_{k+1} > \beta_{k+2}$, where

$$\beta_{k+1} = (U_i^{(\max)} - U_{i_{k+1}}),$$

and

$$\beta_{k+2} = (U_i^{(\max)} - U_{i_{k+2}}).$$

We know from above that $U_{i_{k+2}} > U_{i_{k+1}}$. Therefore

$$(U_i^{(\max)} - U_{i_{k+1}}) > \ (U_i^{(\max)} - U_{i_{k+2}})$$

and $\beta_{k+1} > \beta_{k+2}$.

We next prove $\alpha_{k+1} > \alpha_{k+2}$. Note that

$$U_i = \frac{C_i}{T_i = \frac{C_i - L_i}{m_i} + L_i} \tag{13}$$

Consider Equation (13) which shows the complete derivation of a task's utilization as a function of the number of processors assigned to it. By definition, if $\alpha_{k+1} \overset{?}{>} \alpha_{k+2}$, [3] then

$$U_{i_{k+1}} - U_{i_k} \overset{?}{>} U_{i_{k+2}} - U_{i_{k+1}}.$$

Substituting into Equation (13), this becomes

$$\frac{C_i}{\frac{C_i - L_i}{k+1} + L_i} - \frac{C_i}{\frac{C_i - L_i}{k} + L_i} \overset{?}{>} \frac{C_i}{\frac{C_i - L_i}{k+2} + L_i} - \frac{C_i}{\frac{C_i - L_i}{k+1} + L_i}.$$

Factoring out a constant C_i and simplifying, we get

$$\frac{k+1}{C_i + kL_i} - \frac{k}{C_i + kL_i - L_i} \overset{?}{>} \frac{k+2}{C_i + kL_i + L_i} - \frac{k+1}{C_i + kL_i}.$$

Letting $X = C_i + kL_i$ (to enhance readability), this becomes

$$\frac{k+1}{X} - \frac{k}{X - L_i} \overset{?}{>} \frac{k+2}{X + L_i} - \frac{k+1}{X}.$$

We can combine fractions and simplify this further to

$$\frac{-kL_i + X - L_i}{X(X - L_i)} \overset{?}{>} \frac{-kL_i + X - L_i}{X(X + L_i)}.$$

[3] We use $\overset{?}{>}$ to indicate that the inequality is not yet proved.

Since $-k * L_i + X - L_i = -kL_i + C_i + kL_i - L_i = C_i - L_i > 0$ for high-utilization tasks, we can now factor out $-k * L_i + X - L_i$ from both sides and are left with asking whether

$$\frac{1}{X(X - L_i)} \stackrel{?}{>} \frac{1}{X(X + L_i)}.$$

This is unequivocally true. Hence, we prove that $\alpha_{k+1} > \alpha_{k+2}$. Therefore, Equation (12) is satisfied and $\delta_{i_{k+1}} > \delta_{i_{k+2}}$. The Lemma follows. ◀

▶ **Theorem 3.** *Algorithm 2 optimally minimizes the optimization problem given in Equation (7).*

Proof. For Algorithm 2 to be non-optimal, there must be some point at which our greedy algorithm and the optimal algorithm diverge. (Algorithm 2 begins optimally with the only valid assignment of processors to tasks when considering only the minimum amount of processors each task can have.) Note that each task's contribution to the sum of Equation (7) is independent of other tasks: the value of $\frac{1}{E_i}\ (U_i^{(\max)} - U_i)^2$ for a given task τ_i is independent of how many processors have been assigned to other tasks. Thanks to this property, we need only consider two tasks. Let us suppose, without loss of generality, that at the point of divergence our greedy algorithm assigns the processor to τ_i, while the optimal algorithm would assign the processor to τ_j.

Because the greedy algorithm assigns the processor to τ_i, we know that the added benefit (amount decreased from the sum) is greater than if we had given the processor to τ_j. Hence the current value of the objective function of optimization problem 7 the greedy algorithm is necessarily lower than that of the optimal algorithm upon assignment of the number of processors assigned thus far. By the assumption regarding the non-optimality of our greedy strategy, there must be some point in the future at which the optimal algorithm makes up the difference since the optimal solution to a minimization problem must end with the lowest value for the objective function.

However, we saw in Lemma 2 above that the benefits of assigning a new processor under the greedy Algorithm 2 diminish. At each iteration, the greedy algorithm chooses to assign the processor to the task with the greatest available benefit. Because tasks' benefits are considered independently and do not change regardless of the allocation of CPUs to other tasks, after the greedy algorithm assigns the k'th processor to τ_i, no other task τ_j will have a higher benefit of receiving the $(k+1)$'st processor than it did when the greedy algorithm elected to give the k'th processor to τ_i. Similarly, by Lemma 2 the diminishing returns of assigning multiple processors to the same task guarantees that the benefit of assigning the $(k+1)$'st task to τ_i is also less than the benefit gotten by assigning the k'th processor to τ_i. Therefore, if the optimal algorithm and the greedy algorithm diverge and the current value of the objective function of optimization problem 7 for Algorithm 2 is better than the optimal algorithm, it is impossible for the optimal algorithm to subsequently 'catch up' and do better than the greedy algorithm. Hence the current value of the objective function of optimization problem 7 may never diverge between an optimal algorithm and our greedy algorithm; the optimality of Algorithm 2 immediately follows. ◀

This completes the proof of optimality of Algorithm 2 for the federated scheduling of parallel elastic tasks.

5 Summary & Conclusions

In the two decades since it was first introduced, the elastic task model [4] has proved a useful abstraction for representing flexibility in the computational demands of recurrent workloads. It was originally proposed for representing sequential tasks executing upon uniprocessor platforms;

as high-performance real-time computer applications are increasingly becoming parallelizable (and need to have their parallelism exploited by being implemented upon multiprocessor platforms in order to meet timing constraints), there is a need to extend the applicability of the elastic task model to parallel tasks that execute upon multiprocessor platforms.

In this paper, we have proposed one such extension. The salient features of our model are:

- Multiprocessor scheduling under the federated paradigm, in which each task needing more than one processor is assigned exclusive access to all processors upon which it executes. Federated scheduling frameworks can generally be implemented in a more efficient manner than global scheduling (e.g., with less run-time overhead) with only limited loss of schedulability (as measured by speedup bounds of capacity augmentation bounds).
- Representation of a parallel task's workload using just the cumulative workload (its '*work*' parameter) and its critical path length (its '*span*' parameter). Such representation allows for efficient schedulability analysis in the federated scheduling framework, with a bounded loss of schedulability as compared to DAG representations (for which schedulability analysis is strongly NP-hard.
- Retention of the elasticity coefficient parameter that was the main innovation introduced in [4] to capture the flexibility in computational demands.

We have proposed and studied two schemes for assigning processors to tasks in a system of elastic parallel real-time tasks that are to be scheduled upon a given multiprocessor platform under federated scheduling. One of these schemes is completely semantics-preserving with respect to model semantics as introduced in the uniprocessor case [4]; the other allows for some deviation from uniprocessor semantics and thereby is able to better use the computational capabilities of the implementation platform.

Possible future extensions of this work include the scheduling of hybrid-utilization tasks, each of whose potential utilization range is such that it can be treated as either a low-utilization or a high-utilization task, depending on the system load. This necessarily involves the co-scheduling of low-utilization and high-utilization tasks. It may also be worth investigating different ways of scheduling low-utilization tasks on multi-core systems. Buttazzo's algorithms provide an optimal way to schedule a task set of low-utilization tasks on a single processor but say nothing about how to assign low-utilization tasks to multiple processors. Each of these can also be explored with constrained deadlines and resource sharing.

References

1 P. Antsaklis and J. Baillieul. Guest Editorial Special Issue on Networked Control Systems. *IEEE Transactions on Automatic Control*, 49(9):1421–1423, September 2004. doi:10.1109/TAC.2004.835210.

2 Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leem Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS 2012, pages 63–72, San Juan, Puerto Rico, 2012.

3 Giorgio Buttazzo, Enrico Bini, and Darren Buttle. Rate-Adaptive Tasks: Model, Analysis, and Design Issuess. In *Proceedings of DATE 2014: Design, Automation and Test in Europe*, March 2014.

4 Giorgio C. Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic Task Model for Adaptive Rate Control. In *1998 IEEE Real-Time Systems Symposium (RTSS)*, 1998.

5 Giorgio C. Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic Scheduling for Flexible Workload Management. *IEEE Trans. Comput.*, 51(3):289–302, March 2002. doi:10.1109/12.990127.

6 M. Caccamo, G. Buttazzo, and Lui Sha. Elastic feedback control. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 121–128, 2000. doi:10.1109/EMRTS.2000.853999.

7 T. Chantem, X. S. Hu, and M. D. Lemmon. Generalized Elastic Scheduling. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 236–245, 2006.

8 T. Chantem, X. S. Hu, and M. D. Lemmon. Generalized Elastic Scheduling for Real-Time Tasks. *IEEE Transactions on Computers*, 58(4):480–495, April 2009. doi:10.1109/TC.2008.175.

9 D. Ferry, G. Bunting, A. Maqhareh, A. Prakash, S. Dyke, K. Aqrawal, C. Gill, and C. Lu. Real-

time system support for hybrid structural simulation. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10, October 2014. doi:10.1145/2656045.2656067.

10 David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A Real-time Scheduling Service for Parallel Tasks. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '13, pages 261–272, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/RTAS.2013.6531098.

11 David Ferry, Amin Maghareh, Gregory Bunting, Arun Prakash, Kunal Agrawal, Chris Gill, Chenyang Lu, and Shirley Dyke. On the performance of a highly parallelizable concurrency platform for real-time hybrid simulation. In *The Sixth World Conference on Structural Control and Monitoring*, 2014.

12 R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

13 J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 31–40, April 2013.

14 Jing Li, Abusayeed Saifullah, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Analysis Of Federated And Global Scheduling For Parallel Real-Time Tasks. In *Proceedings of the 2012 26th Euromicro Conference on Real-Time Systems*, ECRTS '14, Madrid (Spain), 2014. IEEE Computer Society Press.

15 C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

16 J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.