Elasticity of Workloads and Periods of Parallel Real-Time Tasks

James Orr
Washington University in St. Louis
james.orr@wustl.edu

Chris Gill
Washington University in St. Louis
cdgill@wustl.edu

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

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

Christian Cianfarani
Brown University
christian_cianfarani@brown.edu

Phyllis Ang
University of Texas at Austin
phyllis.ang@utexas.edu

Christopher Wong
Brown University
christopher_wong2@brown.edu

ABSTRACT
The elastic task model allows sequential periodic real-time tasks, such as those found in multimedia players and adaptive control systems, to adjust their periods dynamically to manage quality of service or to accommodate other tasks. Recent theoretical advances show that parallel real-time tasks can adapt their periods similarly.

This paper further extends the concept of elasticity of parallel real-time tasks, to allow them to adapt their computational workloads instead of their periods, such as when a real-time video processing application can improve image quality if it can do more computation within a given period. This paper also presents a new concurrency platform in which each parallel real-time task can adapt either its period or its workload, supporting heterogeneous forms of elasticity for different application needs.

Empirical evaluations we have conducted (1) demonstrate the ability of this concurrency platform to enforce theoretical guarantees from both prior work and results developed in this paper, and (2) help to quantitatively and demonstrate trade-offs between temporal and computational elasticity.

CCS CONCEPTS
- Computer systems organization → Real-time system architecture; Real-time system specification; Embedded software;

KEYWORDS
Parallel real-time tasks, Computational and period elasticity

1 INTRODUCTION
The traditional sporadic real-time task model often assumes unchanging task characteristics such as a constant worst-case execution time (WCET), minimum inter-arrival time (or period), and relative deadline. This can help in defining theoretical bounds off-line to guarantee schedulability of a taskset. However, such assumptions do not always hold. For example, Buttazzo et al. [1] developed the elastic scheduling model to accommodate adaptive tasks that may need to change their periods dynamically at runtime due to overload conditions or user input (e.g., multi-media players with different frame rates, or adaptive control systems). Recently, our group extended Buttazzo's elastic task model to allow parallel real-time tasks to change their periods [9]. In this paper we further extend those results by developing similar theoretical foundations for elasticity of computational workload.

In the original elastic task model for sequential tasks [1], schedulability of a taskset is determined by system utilization (i.e., the sum of each task's computational workload divided by its period). Buttazzo et al. manipulate a task's period elasticity to change tasks' utilizations as needed to maintain system schedulability, while allowing e.g., a control algorithm to perform better when run at a higher periodic rate, or a multi-media player to offer a better picture when run at a higher frame rate.

However, a task can also change its utilization by adapting its computational load instead of its period, i.e., manipulating its computational elasticity. Either an increase in a task's computational load or a proportional decrease in its period will result in the same increase in its CPU utilization. Similarly, CPU utilization can be decreased by decreasing a task's computational load, or increasing its period. Although an increase or decrease in computational load may not make sense for some tasks, other tasks (e.g. a simulation that must iterate at a constant rate but whose accuracy depends on how much it can compute during each iteration) can use this capability to adjust their quality of service.

Although exploiting computational elasticity is possible in sequential tasks, it is perhaps more relevant for parallel tasks, where computational workload can be increased (while maintaining a constant period) by providing additional CPUs for the task to utilize (which sequential-task scheduling cannot do).

Towards a more comprehensive treatment of elasticity in parallel real-time tasks, this paper introduces and discusses the novel concept of computational elasticity in parallel real-time systems. It also identifies and encapsulates an equivalence between period elasticity and computational elasticity by adapting an algorithm from [9] (originally used to optimally schedule only period-elastic tasks) to...
now optimally schedule both period-elastic and computationally-elastic parallel real-time tasks.

In this paper, we also introduce platform support for interchangeable adaptation of either the period or the workload of each parallel real-time task, which allows some tasks to adjust their periods and others to adjust their workloads. We have designed and implemented an efficient runtime system that when tasks adaptively change their computational demands or their periods, makes the necessary elastic scheduling changes to other tasks in the system to ensure that all tasks in the taskset remain schedulable. We also demonstrate the equivalence of computational and period elasticity by scheduling two tasksets in which all tasks are identical with the exception of one task in each taskset: in one taskset this designated task modifies its period while in the other it modifies its computational load, within the same minimum and maximum utilization. Results of our evaluation show that the system adapts in the same way, and finds the same optimal schedule regardless of whether the designated task adapts its period or its computational workload.

The rest of this paper is structured as follows. Section 2 provides background on elastic scheduling and other related work. Section 3 introduces the expanded parallel elastic real-time task model and proves that the optimal scheduling algorithm introduced in [9] for period adaptation only, is easily extended to work correctly whether tasks change their computational workloads or their periodic rates. Section 4 presents the design of our elastic parallel real-time runtime system and Section 5 evaluates its effectiveness. Section 6 concludes and describes directions for future work.

2 BACKGROUND AND RELATED WORK

In this paper we expand our earlier work on elastic parallel-real time scheduling to support adaptive computational workloads. This section describes single-core elastic scheduling and the Federated Scheduling paradigm, and gives an overview of our prior work with period-elastic parallel real-time tasks.

2.1 Elastic Scheduling

The elastic task model was first introduced by Buttazzo et al. [1] to allow sequential tasks running on a single processor to adapt their periods in order for the system to remain schedulable in case a new task must be admitted to the system or a dynamic task must run at a faster rate. That model uses an extended analogy to compare schedulability of tasks in a taskset to a set of springs laid end-to-end with a common force applied to them in order to compress their combined length to a specified maximum. The utilization of each task becomes the length of its corresponding spring, and the desired system-wide utilization (e.g., 1.0 under preemptive EDF scheduling of a single core) is the target maximum combined length for the set of springs. Just as some springs are easier to compress than others, and each spring has physical bounds on how far it can be compressed or expanded, each task in the elastic scheduling model has an elasticity parameter to indicate how resistant it is to changing its period, and bounds on which values can be selected as its period (and therefore bounds on its utilization) [1].

The elastic task model itself is a generalization of the sporadic task model. Each of $n$ sporadic tasks $t_i = (C_i, T_i^{\text{min}}, T_i^{\text{max}}, E_i)$ has a current minimum inter-arrival time (or period) $T_i$, and a constant workload represented by its worst-case execution time (WCET) $C_i$. Under elastic scheduling each task’s period $T_i$ can vary over the range of $[T_i^{\text{min}}, T_i^{\text{max}}]$ where $T_i^{\text{min}}$ is the preferred period and $T_i^{\text{max}}$ is the slowest acceptable period. To ensure system-wide schedulability, overall system utilization $\sum_{i=0}^{n} \frac{C_i}{T_i}$ must remain below a desired utilization $U_d$ (e.g., 1.0 for single-core preemptive EDF scheduling). Each task’s period is lengthened from $T_i^{\text{min}}$ (thereby reducing task utilization) proportionally to its current utilization and elastic coefficient $E_i$. The elastic coefficient is again a measure of a task’s ability to change its period, similar to a spring’s ability to be expanded or contracted. The higher the value of $E_i$, the more elastic a task, and the more able it is to change its period.

In the original elastic scheduling work [1] Buttazzo et al. presented an efficient ($\Theta(n^2)$) iterative algorithm for task period selection when the system needed to adapt, which (if possible) finds each task $t_i$ an appropriate period $T_i$ such that $\sum_{i=0}^{n}(C_i/T_i) \leq U_d$ and $T_i^{\text{min}} \leq T_i \leq T_i^{\text{max}}$ for all tasks $t_i$. (As stated above $U_d$ is a threshold defined according to the scheduling algorithm that is used; for single-core preemptive EDF scheduling $U_d = 1.0$.) Chlamtak et al. [4, 5] proved that the iterative algorithm from [1] is exactly equivalent to solving the following optimization:

\[
\text{minimize} \quad \sum_{i=1}^{n} \frac{1}{E_i} (U_i^{\text{max}} - U_i)^2 
\]

such that:

\[
U_i^{\text{min}} \leq U_i \leq U_i^{\text{max}} \quad \text{for all } t_i, \quad \sum_{i=1}^{n} U_i \leq U_d
\]

where $T_i^{\text{max}} = \frac{C_i}{U_i^{\text{min}}}$ represents the maximum possible utilization of a task obtained from running at period $T_i = T_i^{\text{min}}$.

The original work involving elastic tasks [1] assumed implicit deadlines in which $D_i = T_i$, but theory involving the model has since been expanded to include: constrained deadlines in which $D_i \leq T_i$ [3], resource sharing [2], and unknown computational load [3]. Our work in this paper explores a similar (but orthogonal) direction to that in [3] except that we assume a variable, yet known and controlled workload.

2.2 Federated Scheduling

The Federated Scheduling paradigm is used to schedule sporadic parallel real-time tasks on multiple processors. It was designed by Li et al. [8] to schedule tasks whose computational demand is such that a single processor cannot possibly guarantee schedulability. As such, a set of processors is assigned to be used exclusively by each parallel real-time task whose computational requirements exceed the capacity of a single CPU.

Each task $t_i$ again has a minimum inter-arrival time (or period) $T_i$ and (in this paper) an implicit deadline $D_i = T_i$. The computational requirements of each task are represented as a directed acyclic graph (DAG) in which each node is a block of sequential computation, and no node can be performed until all of its predecessors have finished execution. Any two nodes whose predecessors have all finished running can be executed in parallel. In place of the WCET
parameter used in sequential tasks (such as for Buttazzo's elastic scheduling model). Federated Scheduling represents the computational workload of a DAG task with total work $C_i$ and critical path length (or span) $L_i$. The work represents the sum of the workloads of each of node in the DAG task. In other words, it is the amount of time it would take the DAG to run sequentially on a single processor. Similarly, the span is the highest-weighted (by summation of computation time) chain of nodes in a DAG. Because the nodes making up such a chain must be run sequentially, the span of a DAG represents the amount of time that the task would need to run on a theoretically infinite number of processors.

In Federated Scheduling, tasks that require more than one processor (i.e., utilization $U_i > 1.0$) are referred to as high utilization tasks. Similarly, tasks that can be scheduled feasibly on a single processor ($U_i \leq 1.0$) are low utilization tasks.

2.3 Parallel Real-time Elastic Scheduling

Our recent work [9] has expanded Buttazzo’s elastic model to include parallel tasks scheduled under Federated Scheduling. Each task $r_i$ is characterized by the parameters

$$r_i = (C_i, L_i, U_i^{(\text{max})}, U_i^{(\text{min})}, E_i)$$

where $U_i^{(\text{max})} = C_i/T_i^{(\text{max})}$ and $U_i^{(\text{min})} = C_i/T_i^{(\text{min})}$ are the maximum (i.e., desired) and minimum acceptable utilization values, respectively. Note that using minimum and maximum utilization values is functionally equivalent to characterizing the task by its maximum and minimum period, but we use the utilization parameter as it is more directly applicable to our scheduling algorithm. As before, $C_i$ and $L_i$ are the work and span of a parallel task represented by a DAG, and $E_i$ is the elastic coefficient of a task, representing the ease with which a task’s period can be changed, analogous to a spring’s resistance to being compressed. Note that in the scheduling of sequential processes (including scheduling of traditional elastic tasks), system-wide utilization (and by extension individual tasks’ utilization) is used directly to determine whether a taskset is schedulable. However, under Federated Scheduling, a taskset is considered schedulable if and only if the $m$ processors available to the system are enough to assign each task its requisite number of dedicated processors, (largely) independent of the utilization of individual tasks. It was proved in [6] that the makespan of the schedule for a given DAG is guaranteed to be no larger than the difference between the work and span divided by the number of processors available, plus the span:

$$\frac{C_i - L_i}{m_i} + L_i$$

Therefore an upper bound on the makespan 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 $D$, $\left(\frac{C_i - L_i}{m_i} + L_i\right) \leq D$ is a sufficient test for determining whether the code will complete by its deadline upon an $m$-processor platform. Because we assume implicit deadlines with $D_i = T_i$, we can show

$$\frac{C_i - L_i}{m_i} + L_i \leq T_i$$

$$\Rightarrow \frac{C_i - L_i}{m_i} \leq T_i - L_i$$

$$\Rightarrow m_i \geq \frac{C_i - L_i}{T_i - L_i}$$

Under Federated Scheduling, since the number of processors assigned each task is an integer, we therefore have

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

(4)

Therefore a taskset is schedulable under Federated Scheduling if and only if $\sum_{i=1}^{n} m_i \leq m$.

As such our work in [9] solves the parallel version of the optimization equation given in Definition (1), namely:

$$\text{minimize} \sum_{i=1}^{n} \frac{1}{E_i} (U_i^{(\text{max})} - U_i)^2$$

(5)

such that:

$$U_i^{(\text{min})} \leq U_i \leq U_i^{(\text{max})}$$

for all $r_i$ and

$$\sum_{i=1}^{n} m_i \leq m$$

In [9] we also present a greedy algorithm that optimally solves Definition (5) where the period may be varied. In this paper we modify that algorithm to schedule tasks with elastic computational loads or periods: Algorithm 1 in Section 3.

3 COMPUTATIONAL ELASTICITY

This section further develops the concept of task elasticity, originally presented as sequential real-time tasks’ ability to dynamically adapt their periods by Buttazzo et al. [1] and expanded in our recent work [9] to allow similar period-only adaptation in parallel real-time tasks. We refer to such tasks as period-elastic tasks. This section further extends those prior results to allow for tasks to adapt their computational load instead of period. We refer to these as computationally-elastic tasks.

In this section we demonstrate that both period-elastic and computationally-elastic tasks can be encapsulated under elasticity of task utilization and thereby form an equivalence relationship. Utilization-based scheduling algorithms used in prior work [1, 2, 4, 5, 9] to schedule period-elastic tasks can then be modified in order to also schedule computationally-elastic tasks.

3.1 Computationally-Elastic Task Model

Our computationally-elastic task model in this paper is similar to the period-elastic task model from [9] shown in Definition 2. However, because we now consider tasks in which adaptation is driven by computational elasticity instead of period elasticity, we replace the fixed computational load $C_i$ with a fixed period $T_i$ and again allow utilization to vary between maximum (desired) and minimum values, which in turn implies a variable workload $C_i$.

This gives our new computationally elastic task model

$$r_i = (T_i, L_i, U_i^{(\text{max})}, U_i^{(\text{min})}, E_i)$$

(6)
in which the minimum inter-arrival time (or period) \( T_i \) is fixed. We again assume an implicit relative deadline \( D_i = T_i \), and each task is again represented by a directed acyclic graph (DAG) with (fixed) critical path length (or span) \( L_i \) and overall work \( C_i \). However, instead of a constant \( C_i \), each task now has a range of acceptable work values \([C_i^{(\text{min})}, C_i^{(\text{max})}]\) similar to the range of periods found in period-elastic tasks. This range of acceptable \( C_i \) values is encapsulated in Definition (6) by the interval of acceptable task utilizations \([U_i^{(\text{min})}, U_i^{(\text{max})}]\) where

\[
U_i^{(\text{min})} = \frac{C_i^{(\text{min})}}{T_i}
\]

and

\[
U_i^{(\text{max})} = \frac{C_i^{(\text{max})}}{T_i}.
\]

In computationally-elastic tasks, elastic coefficient \( E_i \) indicates a task’s ability to handle its computational load change. A higher value of \( E_i \) indicates a task whose \( C_i \) value is more easily changed. The direct proportionality relationship between the minimum and maximum computational load \( (C_i^{(\text{min})} \text{ and } C_i^{(\text{max})}) \) and the minimum and maximum utilization \( (U_i^{(\text{min})} \text{ and } U_i^{(\text{max})}) \) of computationally-elastic tasks provides the key insight for adapting existing period-elastic scheduling techniques to also schedule computationally-elastic tasks. A similar (yet inverse) relationship exists between period and utilization of period-elastic tasks. Since both computational-elasticity and period-elasticity can therefore be encapsulated as utilization, it follows that either could be scheduled under a utilization-based scheduling algorithm. This subsection explores the adaptation of existing utilization-based scheduling algorithms used previously for scheduling exclusively period-elastic tasks to now schedule both period-elastic and computationally-elastic tasks.

As in our prior work with parallel real-time period-elastic tasks [9], we schedule high-utilization tasks using Federated Scheduling. Recall that under Federated Scheduling any task in which \( U_i > 1.0 \) is considered a high-utilization task; and likewise any task in which \( U_i \leq 1.0 \) is a low-utilization task. In this paper we assume that all tasks are always only high-utilization or low-utilization: a task’s elastic nature and adaptive period or computational load cannot carry it across the boundary from one to the other.

### 3.2 Scheduling of Low-Utilization Computationally-Elastic Tasks

Because low-utilization tasks have utilizations less than 1.0, they can be scheduled like sequential tasks. We now show that the algorithms presented by Buttazzo et al. [1, 2] can be adapted to schedule low-utilization computationally-elastic tasks.

The Task_Compress\( \Gamma, U_d \) algorithm presented in [1] converts period-elastic tasks in taskset \( \Gamma \) to their utilization-based abstraction. It then compresses each task’s utilization proportionally to its elastic coefficient (to the extent that each task can be compressed) until the summed utilization is at or below the desired system utilization \( U_d \). Once the desired system utilization has been achieved, the system is guaranteed to be schedulable. Each task’s compressed utilization becomes its assigned utilization, and the task is assigned to run with the corresponding period \( T_i \).

We showed in the previous subsection that computationally-elastic tasks can also be encapsulated as having a minimum and maximum utilization. Therefore, any computationally-elastic task can have its utilization compressed as is done in the Task_Compress algorithm discussed above. To successfully schedule computationally-elastic tasks, then, one must simply run the Task_Compress algorithm and convert utilization to an appropriate work value \( C_i \) instead of a period value \( T_i \).

### 3.3 Scheduling of High-Utilization Computationally-Elastic Tasks

For high-utilization computationally-elastic tasks, recall that under Federated Scheduling each task is assigned a set of dedicated processors. Definition (4) determines the number of CPUs needed to schedule a high-utilization task under Federated Scheduling at a given workload \( C_i \) and period \( T_i \). Because we are scheduling computationally-elastic tasks, each task with a constant period \( T_i \) but variable workload \( C_i \) may therefore use anywhere from \( m_i^{(\text{min})} \) to \( m_i^{(\text{max})} \) tasks where

\[
m_i^{(\text{min})} = \frac{C_i^{(\text{min})} - L_i}{T_i - L_i}
\]

and

\[
m_i^{(\text{max})} = \frac{C_i^{(\text{max})} - L_i}{T_i - L_i}.
\]

This matters because under Federated Scheduling of high-utilization tasks, system utilization does not directly determine whether a taskset is schedulable. Rather, a taskset is schedulable if and only if the system has enough processors to give each task enough dedicated processors such that each individual task is schedulable.

In the remainder of this subsection we discuss adapting algorithm Task_Compress\( \Gamma, U_d \) from [9] to include the scheduling of high-utilization computationally-elastic tasks. The resulting algorithm appears in this paper as Algorithm 1.

Task_Compress\( \Gamma, U_d \) is an optimal (see [9] for proof) and efficient \( \Theta(n+m+n\log n) \) greedy algorithm that directly solves the optimization equation given in Definition (5). The algorithm iteratively assigns a processor to the task that lowers \( \sum_{i \in \Gamma} \left( U_i - U_d \right)^2 \) the most, with the ultimate goal of assigning processors to tasks until all \( m \) processors in the system have been assigned, or all tasks have been given their maximum number of processors while minimizing the above-stated objective.

Although each task has a minimum and maximum number of CPUs on which it can run, \( m_i^{(\text{min})} \) and \( m_i^{(\text{max})} \), we note that any system has a finite number of CPUs \( m \) on which to schedule tasks, and all other tasks in the taskset also have a minimum number of CPUs on which they can run. Therefore, although a task may be theoretically able to run on up to \( m_i^{(\text{max})} \) CPUs, on any given platform it can be assigned at most \( m_i^{\text{practical}} \) where

\[
m_i^{\text{practical}} = m - \sum_{i \in \Gamma} m_i^{(\text{min})}. \tag{7}
\]

Because our prior work in [9] focuses on exclusively period-elastic tasks, the remainder of the discussion in this section will
Algorithm 1 Task_compress_par2(Γ, m)

1. for (τ ∈ Γ) do
2. $m^\text{min}_i = \lceil (C_i^\text{min} - L_i) / T_i^\text{max} \rceil$  
3. $m^\text{max}_i = \lceil (C_i^\text{max} - L_i) / (T_i^\text{min} - L_i) \rceil$  
4. $m_i = m^\text{min}_i$  
5. while $m_i \leq m^\text{max}_i$ do
   6. if $τ_i$ is period-elastic then
      7. $T_{i(m_i)} = C_i / m_i + L_i$
   8. else
      9. $C_{i(m_i)} = m_i(T_i - L_i) + L_i$
   10. end if
   11. $m_i = m_i + 1$
   12. end while
   13. $m_i = m^\text{max}_i$
   14. if $τ_i$ is period-elastic task then
      15. $T_i = T_{i(m_i)}$
   16. else
      17. $C_i = C_{i(m_i)}$
   18. end if
   19. $m = m - m^\text{min}_i$
20. end for
21. if (m < 0) then
   22. return UNSCHEDULED
   23. else if (m == 0) then
      24. return the processor allocation as determined in the $m_i$ values
   25. end if
26. end if
27. for (τ ∈ Γ) do
   28. $x_i = \frac{1}{m_i}(U_i^\text{max} - U_i)^2$
   29. $m_i += 1$
   30. if $τ_i$ is period-elastic then
      31. $T_i = T_{i(m_i)}$
   32. else
      33. $C_i = C_{i(m_i)}$
   34. end if
   35. $y_i = \frac{1}{T_i}(U_i^\text{max} - U_i)^2$
   36. $z_i = x_i - y_i$
   37. $m_i -= 1$
   38. if $τ_i$ is period-elastic then
      39. $T_i = T_{i(m_i)}$
   40. else
      41. $C_i = C_{i(m_i)}$
   42. end if
43. end for

focus on computationally-elastic tasks with an adaptive work parameter $C_i$. However, Algorithm 1 encapsulates both period elasticity and computational elasticity in terms of task utilization. The algorithm begins by determining $m^\text{min}_i$ and $m^\text{max}_i$ for each task (lines 2-3). We assume that for computationally-elastic tasks (with a fixed period $T_i$) that $T_i = T_i^\text{max}$, and likewise for period-elastic tasks (with a fixed computational workload $C_i$) that $C_i = C_i^\text{max}$. This allows reuse of the same calculations regardless of the task’s elastic nature, as the number of required CPUs increases proportionally with a task’s work and inversely with its period.

The algorithm then determines the maximum workload (line 9) that each task can accommodate at each integer over the range $[m^\text{min}_i, m^\text{max}_i]$ and stores them in a lookup table. The value $C_{i(m_i)} = m_i(T_i - L_i) + L_i$
Algorithm 1 Task_compress_par2(\(\Gamma, m\)), (continued)

44. Make a max heap of all tasks, with the \(z_i\) values as the key
45. \textbf{while} \(m > 0\) and heap not empty \textbf{do}
46. \quad \(r_{\text{most}} = \text{heap.pop}()\)
47. \quad \(m_{\text{most}} = m_{\text{most}} + 1\)
48. \quad \(m = m - 1\)
49. \quad \textbf{if} \(r_i\) is period-elastic \textbf{then}
50. \quad \quad \(T_{\text{most}} = T_{(\text{most}, m_{\text{most}})}\)
51. \quad \quad \(C_{\text{most}} = C_{(\text{most}, m_{\text{most}})}\)
52. \quad \quad \textbf{end if}
53. \quad \textbf{if} \(m > 0\) and \(m_{\text{most}} < m_{\text{most,max}}\) \textbf{then}
54. \quad \quad \(x_{\text{most}} = \frac{1}{m_{\text{most}}} (U_{\text{most}} - U_{\text{most}})^2\)
55. \quad \quad \(m_{\text{most}} + 1\)
56. \quad \quad \textbf{if} \(r_i\) is period-elastic \textbf{then}
57. \quad \quad \quad \(T_{\text{most}} = T_{(\text{most}, m_{\text{most}})}\)
58. \quad \quad \quad \(C_{\text{most}} = C_{(\text{most}, m_{\text{most}})}\)
59. \quad \quad \textbf{else}
60. \quad \quad \quad \(C_{\text{most}} = C_{(\text{most}, m_{\text{most}})}\)
61. \quad \quad \textbf{end if}
62. \quad \quad \(y_{\text{most}} = \frac{1}{m_{\text{most}}} (U_{\text{most}} - U_{\text{most}})^2\)
63. \quad \quad \(z_{\text{most}} = x_{\text{most}} - y_{\text{most}}\)
64. \quad \quad \textbf{Reinsert} \(T_{\text{most}}\) \textbf{into heap}
65. \quad \quad \(m_{\text{most}} - 1\)
66. \quad \quad \textbf{if} \(r_i\) is period-elastic \textbf{then}
67. \quad \quad \quad \(T_{\text{most}} = T_{(\text{most}, m_{\text{most}})}\)
68. \quad \quad \quad \(C_{\text{most}} = C_{(\text{most}, m_{\text{most}})}\)
69. \quad \quad \textbf{else}
70. \quad \quad \quad \(C_{\text{most}} = C_{(\text{most}, m_{\text{most}})}\)
71. \quad \quad \textbf{end if}
72. \textbf{end while}
73. \textbf{return} the processor allocation as determined in the \(m_i\) values

comes from solving Definition (4) for the value \(C_i\).
\[
m_i = \left[ \frac{C_i - L_i}{T_i - L_i} \right] \Rightarrow C_i - L_i \leq m_i < C_i - L_i + 1.
\]
Solving both inequalities for \(C_i\), we obtain the interval
\[
m_i(T_i - L_i) - T_i + 2L_i < C_i \leq m_i(T_i - L_i) + L_i.
\]
Since we always want to perform as much computation as possible, we assign the maximum of
\[
C_i(m_i) = m_i(T_i - L_i) + L_i.
\]

We then assign each task its minimum number of CPUs (line 13) and corresponding computational load (line 17). If not all CPUs have been assigned to a task, the algorithm continues (lines 27–43) by temporarily assigning an additional processor to each task that has fewer than \(m_i^{\text{max}}\) CPUs currently assigned, to see how much each task will reduce the sum in Definition (5).

In line 44 each task is inserted into a max heap that is indexed on how much each task decreases the sum. We can consider tasks independently because the objective function in Definition (5) considers only each task’s current utilization \(U_i = C_i/T_i\) and maximum utilization \(U_{i}^{\text{max}} = C_{i}^{\text{max}}/T_{i}^{\text{max}}\), which are constant \((T_i)\) or determined only by each task’s number of currently assigned processors \((C_i = m_i(T_i - L_i) + L_i)\) and thus independent of other tasks.

Lines 45–72 then repeatedly pop \(r_{\text{most}}\), the task that most reduces the objective function in Definition (5), from the max heap and permanently assigns it a processor. If the task can still receive more processors, its next potential contribution to the objective function in Definition (5) is calculated, and it is reinserted into the heap, until all \(m\) processors have been assigned to a task and the algorithm returns. Note that Algorithm 1 decides how many processors each task gets, not which processors, as we discuss in Section 4.

4 CONCURRENCY PLATFORM SUPPORT

This section presents the concurrency platform we have developed to run both period-elastic and computationally-elastic high-utilization tasks. Each task in the system must be either computationally-elastic or period-elastic, but this runtime system allows for the same taskset to contain both types of elastic tasks.

We schedule tasks under Federated Scheduling using Algorithm 1 above. Because of the adaptive nature of elastic tasks, this scheduling algorithm must be rerun any time a system-wide adaptation occurs. Rather than taking away processing time from (and possibly affecting the schedulability of) a task, we dedicate one processor core to running Algorithm 1 and rescheduling the taskset when a need arises. Section 4.1 discusses the scheduler in greater detail.
The system was built in C/C++ atop Linux with OpenMP. It uses Linux shared memory and POSIX RT Signals for inter-task communication and scheduling. Section 4.2 describes the concurrency and synchronization techniques used.

### 4.1 Task Scheduler and Scheduling Mechanisms

Because elastic tasks must adapt on-line, off-line calculation of a static schedule, as is usually done with non-elastic fixed-priority scheduling, only provides an initial schedule, and we must also decide when and how to run an on-line scheduling algorithm.

We dedicate a processor (arbitrarily CPU 0) to running Algorithm 1 and make \( m \) (the number of CPUs available to the taskset for scheduling) one less than the number of physical processors available in the system. The Task Scheduler process begins by running Algorithm 1 to find an initial processor allocation for each task. It then assigns consecutive processors to each task in turn as determined by the results of Algorithm 1 (i.e., beginning by assigning CPU 1 to Task \( t_1 \) and then assigning CPU 2 to Task \( t_2 \) if it needs more CPUs or otherwise assigning it to Task \( t_2 \), etc.). Tasks then run on their assigned processors until a scheduling change must occur. Meanwhile the Task Scheduler process repeatedly polls whether it needs to reschedule the tasks, and does so as necessary.

When a new schedule must be found, the Task Scheduler first runs Algorithm 1 again to determine how many processors each task must now be assigned. It then looks at which tasks gained CPUs and which tasks lost CPUs when going from the old configuration to the new and determines which processor will go to which new task. Preference is given to any task with a passive thread already sleeping on a CPU currently occupied by an active thread of a task that must lose a processor. If so, the active thread of the task losing a processor can go to sleep; the passive thread of the task gaining a processor can wake up, and no thread migration occurs.

However, it sometimes unavoidable that the task gaining a processor has no passive threads sleeping on a CPU currently occupied by an active thread of a task that is losing a CPU. In this case, a passive thread from the task gaining a processor migrates to a processor currently occupied by the task losing a processor. The...
corresponding formerly active thread then goes to sleep and the thread that migrated becomes active on the CPU.

Figure 1(a) shows the initial state of an example taskset in which Task 1 has three CPUs, Task 2 has two CPUs, and Task 3 has 5 CPUs. At some point during execution, however, (as shown in Figure 1(b)) Task 1 notifies the Task Scheduler that it must adapt, and the Task Scheduler runs Algorithm 1 to determine how many CPUs each task should have after the transition. It is determined that Task 1 should gain a CPU at the expense of Task 3, as is shown in Figure 1(c). Running Algorithm 1 is $\Theta(n^2)$.

### 4.2 Concurrency and Synchronization

As with any parallel system, concurrency and synchronization are essential. This subsection describes the mechanisms we use to prevent data races and deadlock when each task has not only active threads but also potentially passive threads that are on CPUs currently assigned to other tasks, and transitions in which sole active possession of a CPU is transferred from one task to another.

#### 4.2.1 Shared Memory

We use Linux shared memory for all information about task scheduling, including each task’s current period, computational workload, and processors with active or passive threads. Each task’s data can only be modified by the task itself (e.g., if a task must now run at a certain rate) and by the Task Scheduler, although tasks can read each other’s scheduling data if they must. Furthermore, tasks can only write to their own region if the Task Scheduler is currently polling for whether it needs to reschedule, while a re-schedule is occurring. (The task modifying its own data is what ultimately triggers a reschedule in the Task Scheduler.) This therefore ensures that a task and Task Scheduler never attempt to modify the same location at the same time.

#### 4.2.2 POSIX RT Signals

We use POSIX RT Signals for event notification between processes. Unlike standard POSIX signals which are ignored if the task is in the signal handler, RT Signals queue, which guarantees the signal handler is entered each time the signal is received, and no reschedule or adaptation is missed.

Whenever a task has finished modifying its own shared memory region a POSIX RT signal is sent to the Task Scheduler to notify it that a reschedule should occur. Similarly, whenever the Task Scheduler has finished creating the new schedule and assigning processors to tasks, it sends a different POSIX RT signal to each task (in parallel) to notify each task that it needs to read the shared memory region and potentially transition to a new set of CPUs.

#### 4.2.3 Thread Barrier

As mentioned earlier, each task has $m_1$ active threads at any given time, but also $m_{practical}^{(0)} - m_1$ passive threads asleep on other processors. Because we are using OpenMP (OMP) for parallelism, each thread, whether active or passive, must participate in each $\text{#pragma omp parallel region}$ and is immediately and automatically awoken by OpenMP at the beginning of each iteration. We resolve this issue by implementing a modified version of the parallel barrier introduced previously [7]. The first thing each OMP thread does during a parallel region is reach this barrier. If the thread is designated as active by the task, then the task may proceed past the barrier. However, if the thread is designated as passive, the thread immediately goes back to sleep. The active threads then do the necessary parallel work, and reach the barrier again at the end of the parallel region. The last active thread to reach the barrier then wakes up any sleeping passive threads which race through the already completed parallel work and to the barrier.

Just as all threads must enter the parallel region, no thread can leave the $\text{#pragma omp parallel region}$ until all threads are ready to leave. Therefore, both the barrier and waking up passive tasks are unavoidable. To minimize the amount of time spent in the barrier, each passive thread is given a higher real-time priority than each active thread so that a passive thread can immediately wake, pass through the barrier, and go back to sleep. The amount of time that each active thread can be interrupted is determined by the number of passive threads sleeping on each task and their task periods. An active thread will be interrupted at the beginning and end of each of the passive task’s iterations. This is a very small overhead each time, however (10 – 20μ sec).

### 4.3 Ensuring a Safe Transition

To ensure no task misses a deadline due to giving up a processor, transitions of CPUs must occur between iterations of a task. At the end of each iteration, a task checks to see whether rescheduling has occurred. If so, it attempts to transition to its new set of CPUs. If the task loses CPUs, it marks all active threads on them as passive and makes those threads sleep. This task can now begin its next iteration with fewer CPUs and its updated period or computational workload, depending on the nature of that task’s elasticity. Tasks gaining CPUs, however, cannot take possession of their new CPUs until the prior task has given it up, since under Federated Scheduling entire CPUs are dedicated to a single task. Therefore, if all of its gained CPUs are not ready, it keeps its prior set of CPUs and begins another iteration under its prior workload or period. The amount of time it takes an individual CPU to transition from its old owner to its new owner is therefore bounded by at most one iteration of its previous owner’s period and one iteration of its new owner’s period (both before the transition).

### 5 EVALUATION

This section evaluates the run-time system discussed in Section 4. We begin by measuring overheads to gauge the efficiency of our system. We also use Kernelshark to observe the adaptation working as expected. We then use this runtime system to show the equivalence between period-elastic and computationally-elastic tasks.

All experiments used OpenMP parallel programs written in C/C++ and compiled with GCC 4.8.2. They were run on a 32-core machine with four Intel Xeon E5-4620 processors running at a constant 2.20014 GHz with hyper-threading disabled. The RTOS is x86-64 Linux kernel version 4.1.7 with the RT-PREEMPT patch.

#### 5.1 Overheads and Efficiency

We begin by examining the overheads of scheduling, migration, and communication mechanisms used in the runtime system. We use POSIX RT signals to notify tasks of events. Figure 2(a) shows the distribution of the measured latency to deliver a POSIX RT signal. This experiment involved two tasks, a signal sender, and a signal receiver. The signal sender’s only job was to send signals to the signal receiver. It notes the current time, sends a signal, sleeps
for 250 µsec and then sends another signal. The signal receiver is constantly doing busy work but is repeatedly interrupted by the signal handler. Inside the signal handler, the receiver notes the current time and returns to its busy work. This is repeated 10,000 times. The values shown in Figure 2(a) are obtained by subtracting the time recorded in the signal sender before sending the signal from the time recorded in the signal handler by the signal receiver. We note a minimum reaction time of 11.23 µsec and a maximum of 110.03 µsec with over 1/3 falling in the range [18.0, 19.0] µsec.

Figure 2(b) shows a distribution of the overhead associated with changing a task’s real-time priority and migrating it to another processor, as must be done in the worst-case when a passive thread must change its priority and migrate to another CPU to run. This experiment consisted of a task randomly selecting a real-time priority (from 1 to 98), randomly selecting a processor (from 0 to 31) then taking note of the current time, changing its real-time priority, migrating to its new processor, and taking note of the time again. This happened 10,000 times. The times are noted in Figure 2(b). The minimum observed time was 2.67 µsec (it is likely that the randomly selected CPU was the one on which the task was already running and therefore no migration was necessary—note this also happens in our system when a task obtaining a CPU already has a passive thread sleeping on the CPU it will be gaining), and the maximum observed time was 76.77 µsec.

5.2 Adaptation of a Taskset

We randomly generated several tasksets in a similar fashion to prior work [7, 10]. Each task had a 50% chance of being computationally-elastic vs. period-elastic. Each task had a maximum ratio of span to minimum period of $p_{\text{max}} = \frac{1}{2(M+V)}$. The actual span to period ratio is first generated as a percentage of $p_{\text{max}}$: 40%, 50%, 70%, or 100% with probability 0.4, 0.3, 0.2, or 0.1, respectively. Once this ratio has been determined, an actual span value is computed by repeatedly generating segments of work from a log normal distribution with mean of 5ms until the sum reaches the chosen percentage of one second. Each time a segment’s length is chosen, it also generates a number of strands (how many times each segment must be run) from a log normal distribution with a mean of $1 + \sqrt{m}/3$. The sum of the length of each strand times its number of segments becomes the work for period-elastic tasks. For computationally-elastic tasks, two numbers of strands are generated for each segment. The lower is added to the minimum work, and the higher is added to the maximum work. For computationally-elastic tasks a period is generated uniformly between a minimum of 50ms and a maximum of 1s. For period-elastic tasks two periods are generated and the higher value becomes the maximum period and the lower value becomes the minimum period. Each task’s elasticity value was randomly generated over the interval [0.0, 1.0]. If at any time a task does not have a minimum of at least 2 CPUs and a maximum of at least 3 CPUs, the task is discarded and another task is generated.

We generated hundreds of tasksets, and Algorithm 1 found a suitable schedule for each. For each taskset we randomly selected one task that would set its period or workload (depending on the elastic nature of the task) to a randomly selected value between its minimum and maximum. All other tasks would adapt and reschedule accordingly. No deadlines were ever missed, indicating that the transition system described in Section 4 indeed provides a safe and efficient reassignment of CPUs from one task to another.

Kernelshark was used to verify that tasks were behaving as they were supposed to, and after a mode change each CPU each transfer happened exactly as described in the previous section: the task giving up a CPU did so at the end of an iteration, and the task receiving the CPU would begin using it at the beginning of the following iteration. Furthermore, we saw regular periodic behavior at exactly the period expected of each task.

5.3 Functional Equivalence of Period- and Computationally- Elastic Tasks

To assess equivalence of period-elastic and computationally-elastic tasks we performed the following experiments: First, two identical tasksets were generated with the exception of one task in each taskset. In one taskset the designated task is period-elastic, and in the other taskset it is computationally-elastic. These two tasks are functionally equivalent in that they have the same elasticity and minimum and maximum utilization. All other tasks were generated as described in the previous subsection. We adapted the period-elastic task to run at the constant period of the computationally-elastic task. Likewise the computationally-elastic task adapted to
run the constant workload of the period-elastic task. As expected the remaining tasks in each taskset adapted in the same way, regardless of the designated task’s elastic nature. We show the results of four such experiments below. The period and work values before and after adaptation are shown in the charts above. For taskset 1 we also plot each task in terms of its current work and period both before and after a reschedule. In the graph on the left the computationally-elastic task (triangle) and period-elastic task (square) are at different locations before the transition, but in the graph on the right, they have adapted to the star location.

6 CONCLUSIONS AND FUTURE WORK
This paper has extended the state of the art in elastic scheduling by introducing the concept of computational elasticity. We show that computational elasticity is functionally equivalent to period elasticity, as both can be encapsulated as utilization elasticity. We then modify existing scheduling algorithms for period-elastic tasks to include computationally-elastic tasks. We validated the equivalence of period and computational elasticity using a runtime system we built for elastic parallel real-time systems.

Directions for future work include adapting discrete sets of utilizations, a much harder problem to solve but more applicable in practice, as relatively few tasks can run arbitrary amounts of work. Another possible direction is allowing for a task’s span to change along with its work, which would also be relevant to a larger set of real-world applications. Scheduling of tasks that can be either low-utilization or high-utilization depending on their currently-assigned period or workload also remains an open problem.
ACKNOWLEDGMENTS

This research was supported in part by NSF grant CCF-1337218 titled “XPS: FP: Real-Time Scheduling of Parallel Tasks.”

REFERENCES