Full proofs of lemmas

Lemma 3 *Suppose length-n vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ differ at exactly k values, and for these values $y_i = x_i + \delta$, where δ is a positive constant. Denote $w = \min\{k, m^+ - 1\}$.*

Then, the following inequality holds:

$$L(\boldsymbol{x}) \leq L(\boldsymbol{y}) \leq L(\boldsymbol{x}) + \delta \cdot w. \tag{18}$$

Proof. We will define a *candidate sum* for $\boldsymbol{x}$ as any sum of $m^+ - 1$ distinct $l(\tau_i, x_i, p)$ values as in (3). By (4), $L(\boldsymbol{x})$ is the largest candidate sum for $\boldsymbol{x}$.

First, we prove $L(\boldsymbol{x}) \leq L(\boldsymbol{y})$. Consider the candidate sum S for $\boldsymbol{y}$ computed by selecting the same i and p values as in $L(\boldsymbol{x})$. Because for all $i, x_i \leq y_i$, $L(\boldsymbol{x}) \leq L(\boldsymbol{y})$. Because $L(\boldsymbol{y})$ must be the largest candidate sum for y, $S \leq L(\boldsymbol{y})$. Therefore, $L(\boldsymbol{x}) \leq L(\boldsymbol{y})$.

Next, we prove $L(\boldsymbol{y}) \leq L(\boldsymbol{x}) + \delta \cdot w$ by contradiction. Suppose $L(\boldsymbol{y}) > L(\boldsymbol{x}) + \delta \cdot w$. Consider the candidate sum T for $\boldsymbol{x}$ computed by selecting the same i and p values as in $L(\boldsymbol{y})$. Observe that at most w terms contribute to the difference between $L(\boldsymbol{y})$ and T. When two such terms differ, we have $x_i = y_i - \delta$ ($x_i = y_i$ otherwise). Thus, $T \geq L(\boldsymbol{y}) - \delta \cdot w$, and hence, $T > L(\boldsymbol{x})$, which contradicts the fact that $L(\boldsymbol{x})$ is a *maximal* candidate sum for $\boldsymbol{x}$.

Lemma 4 *If $\boldsymbol{y}$ is compliant and there is a j such that $y_j > (L(\boldsymbol{y}) + S(\tau) + U(\tau)D_i - C_i)/m$, then there exists a strictly smaller vector $\boldsymbol{x}$ that is also compliant.*

Proof. Define $\boldsymbol{x}$ such that $x_i = y_i$ for $i \neq j$, and

$$x_j = \frac{L(\boldsymbol{y}) + S(\tau) + U(\tau)D_j - C_j}{m}. \tag{19}$$

In this case, $\boldsymbol{x}$ and $\boldsymbol{y}$ are of the form of Lem. 3 with $k = 1$. Therefore, $L(\boldsymbol{x}) \leq L(\boldsymbol{y})$.

We now have for all $i \neq j$,

$$\begin{aligned}
&\frac{L(\boldsymbol{x}) + S(\tau) + U(\tau)D_i - C_i}{m} \\
\leq\ &\{\text{Since } L(\boldsymbol{x}) \leq L(\boldsymbol{y})\} \\
&\frac{L(\boldsymbol{y}) + S(\tau) + U(\tau)D_i - C_i}{m} \\
\leq\ &\{\text{Since } \boldsymbol{y} \text{ is compliant, by (6)}\} \\
&y_i
\end{aligned}$$

$$= x_i.$$

Also, by construction,

$$\begin{aligned}
&\frac{L(\boldsymbol{x}) + S(\tau) + U(\tau)D_j - C_j}{m} \\
\le\ &\{\text{Since } L(\boldsymbol{x}) \le L(\boldsymbol{y})\} \\
&\frac{L(\boldsymbol{y}) + S(\tau) + U(\tau)D_j - C_j}{m} \\
=\ &\{\text{By (19)}\} \\
&x_j.
\end{aligned}$$

Therefore, $\boldsymbol{x}$ is compliant.

Lemma 21 *$L(s)$ is continuous over $\mathbb{R}$*

Proof. Let $\epsilon > 0$ and $\delta_c \stackrel{\text{def}}{=} \frac{\epsilon}{m^+-1}$. Consider s_0 such that $|s - s_0| < \delta_c$. Without loss of generality, assume $s < s_0$ (otherwise we can swap them.) Then $\boldsymbol{v}(s)$ and $\boldsymbol{v}(s_0)$ are of the form of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, in Lem. 3, with $k = n$. Thus,

$$\begin{aligned}
L(\boldsymbol{v}(s)) \le\ &\{\text{By Lem. 3}\} \\
&L(\boldsymbol{v}(s_0)) \\
\le\ &\{\text{By Lem. 3}\} \\
&L(\boldsymbol{v}(s)) + \delta_c \cdot (m^+ - 1) \\
=\ &\{\text{By the definition of } \delta_c\} \\
&L(\boldsymbol{v}(s)) + \epsilon.
\end{aligned}$$

Therefore, $|L(s) - L(s_0)| \le \epsilon$, so $L(s)$ is continuous over $\mathbb{R}$.

Lemma 25 $s_1 \neq s_2$ implies $M(s_1) \neq M(s_2)$

Proof. Without loss of generality, assume $s_2 > s_1$ (otherwise, swap them). $\boldsymbol{v}(s_1)$ and $\boldsymbol{v}(s_2)$ are of the form of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, with $\delta = (s_2 - s_1)$ and $k = n$, in Lem. 3. Therefore,

$$L(s_2) \le L(s_1) + (s_2 - s_1)(m^+ - 1). \tag{20}$$

Thus,

$$M(s_2) - M(s_1)$$

Table 1: 2 CPU task system example for Sec. 6

	C_i	T_i	D_i
τ_1	6	10	10
τ_2	12	10	10
τ_3	4	20	20

$$
\begin{aligned}
&= \{\text{By (16)}\}\\
&\quad L(s_2) - ms_2 - L(s_1) + ms_1\\
&\le \{\text{By (20)}\}\\
&\quad L(s_1)+(s_2-s_1)(m^+-1)-ms_2-L(s_1)+ms_1\\
&= \{\text{Simplifying}\}\\
&\quad (s_2 - s_1)(m^+ - 1 - m)\\
&\le \{\text{Since } m^+ \le m\}\\
&\quad -1(s_2 - s_1)\\
&< 0.
\end{aligned}
$$

Therefore, $M(s_1) \neq M(s_2)$.

Computation Algorithm

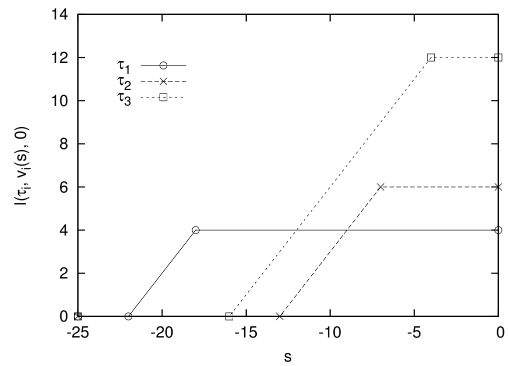
We now show how to compute the minimum compliant vector for a task system τ in time polynomial to the size of τ and the number of processors. $L(s)$ as defined in (15) is a piecewise linear function; our algorithm works by tracing $L(s)$ until we find a fixed point $L(s) = ms$.

In order to assist the reader's understanding of this algorithm, we provide an example task system in Table 1.[2] Simple calculations reveal that, for this system, $S(\tau) = 0$ and $U(\tau) = 2$. Furthermore, in a two-CPU system, by Def. 1, we only need to consider $p = 0$. A graph of the relevant $l(\tau_i, v_i(s), 0)$ functions with respect to s is provided in Fig. 3.

We define the *slope* at point s of a piecewise linear function $f(s)$ to be $lim_{\epsilon\to 0^+}\frac{f(s+\epsilon)-f(s)}{\epsilon}$. This definition differs from the common notion of derivative in that its limit is taken from the right; it is thus defined for all real s.

[2] In this system, the worst-case execution time of τ_2 exceeds its deadline, so it appears that it is impossible for τ_2 to meet its deadline. However, because execution times given are worst-case rather than exact, it is actually possible for this job to complete before its deadline. Furthermore, here we are interested in response-time bounds rather than hard deadlines.

Fig. 3: l functions for the system in Table 1



For example, $l(\tau_1, v_1(s), 0)$ in Fig. 3 has a slope of 1 at $s = -22$, but is not differentiable at $s = -22$.

For each value of s we will define $l(\tau_i, v_i(s), p)$ as being in one of three states, depending on the value of $v_i(s) + C_i - pT_i$:

- If $v_i(s)+C_i-pT_i < 0$, then $l(\tau_i, v_i(s), p)$ is in state 0, is equal to 0, and has a slope of 0. $l(\tau_1, v_1(s), 0)$ in Fig. 3 is in state 0 in the interval $(-\infty, -22)$.
- If $0 \le v_i(s) + C_i - pT_i < C_i$, then $l(\tau_i, v_i(s), p)$ is in state 1, is equal to $v_i(s) + C_i - pT_i$, and has a slope of 1. $l(\tau_1, v_1(s), 0)$ in Fig. 3 is in state 1 in the interval $[-22, -18)$.
- If $C_i \le v_i(s) + C_i - pT_i$, then $l(\tau_i, v_i(s), p)$ is in state 2, is equal to C_i, and has a slope of 0. $l(\tau_1, v_1(s), 0)$ in Fig. 3 is in state 1 in the interval $[-18, \infty)$.

In order to analyze the piecewise linear function $L(s)$, we will need to determine where the slope changes. To do so, we need to determine which $l(\tau_i, v_i(s), p)$ components contribute to $L(s)$ for various intervals. For some intervals, the choice is arbitrary. For example, the task system in Fig. 3 has only one $l(\tau_i, v_i(s), p)$ component contributing to $L(s)$, because $m-1 = 2-1 = 1$. However, for $s < -22$ all $l(\tau_i, v_i(s), p)$ components equal zero. We provide a sufficient solution by arbitrarily tracking some valid set of $l(\tau_i, v_i(s), p)$ components.

We will create a set *Points* of tuples, one for each possible change in the slope of $L(s)$. (Each will have an associated s value, but there could be multiple possible changes at the same s value.) Each tuple will identify a point where some $l(\tau_{i_0}, v_{i_0}(s), p_0)$ in state h_0 is replaced by some $l(\tau_{i_1}, v_{i_1}(s), p_1)$ in state h_1. Such a tuple will be of the form $\{s, i_0, p_0, h_0, i_1, p_1, h_1\}$. In some cases, more than one old component may be appropriate. To handle these cases efficiently, any of i_0, p_0, or h_0 may be set to $*$, which is defined as matching any value of the relevant parameter. For example, the tuple $\{s, *, *, 0, i_1, p_1, 1\}$ indicates that any arbitrary $l(\tau_{i_0}, v_{i_0}(s), p_0)$ in state 0 should be replaced by $l(\tau_{i_1}, v_{i_1}(s), p_1)$ in state 1.

The slope of $L(s)$ may change in any of the following cases:

1. Some $l(\tau_i, v_i(s), p)$ changes from state 0 to state 1. This occurs where $v_i(s) + C_i - pT_i = 0$. The resulting tuple will be $\{s, *, *, 0, i, p, 1\}$, as we can view $l(\tau_i, v_i(s), p_i)$ as replacing any $l(\tau_j, v_j(s), p_j)$ in state 0 in the system—they all have value 0. This change occurs exactly once per $l(\tau_i, v_i(s), p)$ and therefore $m - 1$ times per task (once per value of p), for a total of $O(mn)$ times for the system. In Fig. 3, this state change occurs for $l(\tau_1, v_1(s), 0)$ at $s = -22$, for $l(\tau_2, v_2(s), 0)$ at $s = -13$, and for $l(\tau_3, v_3(s), 0)$ at $s = -16$.
2. Some $l(\tau_i, v_i(s), p)$ changes from state 1 to state 2. This occurs where $v_i(s) + C_i - pT_i = C_i$ (so $v_i(s) = pT_i$). The resulting tuple will be $\{s, i, p, 1, i, p, 2\}$. As above, this change occurs $O(mn)$ times for the system. In Fig. 3, this state change occurs for $l(\tau_1, v_1(s), 0)$ at $s = -18$, for $l(\tau_2, v_2(s), 0)$ at $s = -7$, and for $l(\tau_3, v_3(s), 0)$ at $s = -4$.
3. Some $l(\tau_i, v_i(s), p_i)$ is in state 1 and crosses C_j, and thus potentially crosses $l(\tau_j, v_j(s), p_j)$ (for some p_j) where the latter is in state 2. This occurs when $C_i > C_j$ and $v_i(s) + C_i - p_iT_i = C_j$. The resulting tuple will be $\{s, j, *, 2, i, p, 1\}$. This point may exist at most $n - 1$ times per $l(\tau_i, v_i(s), p)$ (in the worst case, $l(\tau_i, v_i(s), p)$ crosses one $l(\tau_j, v_j(s), p_j)$ for each other τ_j), so occurs at most $O(mn^2)$ times for the system. In Fig. 3, this point does not occur for τ_1 (as C_1 is the smallest value in the system), occurs for $l(\tau_2, v_2(s), 0)$ with τ_1 at $s = -9$, and occurs for $l(\tau_3, v_3(s), 0)$ with τ_1 at $s = -12$ and with τ_2 at $s = -10$. (Although $l(\tau_3, v_3(s), 0)$ does not actually cross $l(\tau_2, v_2(s), 0)$ at $s = -10$, our algorithm nonetheless records the point where $l(\tau_3, v_3(s), 0)$ crosses C_2.)

In order to track $L(s)$, we order the tuples in *Points* by s value, breaking ties in favor of tuples indicating a change in state for a particular $l(\tau_i, v_i(s), p)$ component. We create a list *Active* containing tuples $\{i, p, h\}$, each representing the corresponding $l(\tau_i, v_i(s), p)$ in state h that contributes its value to $L(s)$. For s smaller than the smallest in *Points*, we may arbitrarily make $m^+ - 1$ choices of $l(\tau_i, v_i(s), p)$ components, each in state 0. Therefore, we initialize *Active* to an arbitrary choice of $m^+ - 1$ tuples of the form $\{i, p, 0\}$.

The appropriate s value is computed using Algorithm 1, which works by tracing the piecewise linear function and checking for $L(s) = ms$ (as per (12), (14), and (15)) in each segment.

As an example, suppose *Active* is initialized to $\{\{3, 0, 0\}\}$, which represents $l(\tau_3, v_3(s), 0)$ in state 0. The first tuple in *Points* is $\{-22, *, 0, 0, 1, 0, 1\}$, representing the leftmost slope change in Fig. 3. This tuple will match the single tuple in *Active*, so *Active* will become $\{1, 0, 1\}$. *slope* is used to track the slope between s_1 and the next s value in *Points* (which is called s_2). *current* is used to represent the correct value of $L(s_1)$. In this case, the current interval of interest is $-22 \leq s < -18$. The new state h_2 is 1, so the *slope* (which was initially 0) will be incremented by 1, resulting in a new *slope* of 1. We now know the slope $slope = 1$ of $L(s)$ over $[-22, -18)$ and its value $L(s_1) = current = 0$ at $s_1 = -22$. We therefore compute the point where $L(s) = ms$ would hold, assuming a linear function that is equal to the correct piecewise linear function over the interval of interest. In this case, s is assigned the value $\frac{0-(-22)}{2-1} = 22$, which is not in $[-22, -18)$, so the desired value of s for the algorithm is not in the current interval of interest. We do not return, so we update the value *current* to match the value of $L(s_2)$ at the end of the current interval of interest (and thus in the next iteration the correct value of $L(s_1)$). In this case, *current* will be assigned to $0 + 1 \cdot 4 = 4$.

Points is of size $O(mn^2)$ and *Active* of size $O(m)$, so checking for matches will require $O(m^2n^2)$ operations over the execution of the algorithm. Each match requires $O(1)$ time to process, so the complexity of Algorithm 1 is $O(m^2n^2)$. Computing *Points* requires $O(mn^2)$ time, and sorting requires $O(mn^2 \log(mn))$ time, so the complexity of computing s is $O(mn^2 log(mn) + m^2n^2)$. Once an s value has been computed using Algorithm 1, the correct minimum compliant vector is simply $\boldsymbol{v}(s)$, which can be computed in $O(n)$ time.

Algorithm 1 Compute s value

tuple set *Active*, *Points*, described in text
integer *slope*, *current* **initially** 0
real s, s_2
for all $\{s_1, i_1, p_1, h_1, i_2, p_2, h_2\} \in \mathit{Points}$ **do**
  if $\{i_1, p_1, h_1\}$ matches some $\{i, p, h\}$ in *Active* **then**
    Replace $\{i, p, h\}$ with $\{i_2, p_2, h_2\}$
    if $h_2 = 1$ **then**
      {Changing to state 1 means slope increases}
      $\mathit{slope} := \mathit{slope} + 1$
    else
      {Must be changing away from state 1 or $\{s_1$, i_1, p_1, h_1, i_2, p_2, $h_2\}$ wouldn't be in *Points*}
      $\mathit{slope} := \mathit{slope} - 1$
    end if
    $s_2 :=$ next s value from *Points*, or $C_{\max}$ if there is no such value
    $s := \frac{\mathit{current} - \mathit{slope} \cdot s_1}{m - \mathit{slope}}$
    if $s \in [s_1, s_2)$ **then**
      return s
    end if
    $\mathit{current} := \mathit{current} + \mathit{slope} \cdot (s_2 - s_1)$
  end if
end for