NPM-BUNDLE: Non-Preemptive Multitask Scheduling for Jobs with BUNDLE-Based Thread-Level Scheduling

Corey Tessler
Wayne State University, Detroit, Michigan, USA
corey.tessler@wayne.edu

Nathan Fisher
Wayne State University, Detroit, Michigan, USA
fishern@wayne.edu

Abstract

The BUNDLE and BUNDLEP scheduling algorithms are cache-cognizant thread-level scheduling algorithms and associated worst case execution time and cache overhead (WCETO) techniques for hard real-time multi-threaded tasks. The BUNDLE-based approaches utilize the inter-thread cache benefit to reduce WCETO values for jobs. Currently, the BUNDLE-based approaches are limited to scheduling a single task. This work aims to expand the applicability of BUNDLE-based scheduling to multiple task multi-threaded task sets.

BUNDLE-based scheduling leverages knowledge of potential cache conflicts to selectively preempt one thread in favor of another from the same job. This thread-level preemption is a requirement for the run-time behavior and WCETO calculation to receive the benefit of BUNDLE-based approaches. This work proposes scheduling BUNDLE-based jobs non-preemptively according to the earliest deadline first (EDF) policy. Jobs are forbidden from preempting one another, while threads within a job are allowed to preempt other threads.

An accompanying schedulability test is provided, named Threads Per Job (TPJ). TPJ is a novel schedulability test, input is a task set specification which may be transformed (under certain restrictions); dividing threads among tasks in an effort to find a feasible task set. Enhanced by the flexibility to transform task sets and taking advantage of the inter-thread cache benefit, the evaluation shows TPJ scheduling task sets fully preemptive EDF cannot.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Real-time schedulability

Keywords and phrases Scheduling algorithms, Cache Memory, Multi-threading, Static Analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.15

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at https://dx.doi.org/10.4230/DARTS.5.1.2

Funding The research presented in this paper was supported by the National Science Foundation under Grant Nos. CNS-1618185 and IIS-1724227.

1 Introduction

Hard real-time multi-threaded task systems which incorporate cache memory, must account for the variation in execution time and cache related preemption delays found in single-threaded task systems. For multi-threaded task systems, the complexity of cache interactions is increased due to thread-level cache interference and preemptions. Worst-case execution time (WCET) and schedulability analysis of hard real-time multi-threaded tasks commonly treat threads independently [21] or utilize cache management techniques [33] to limit the cache interference.

Analysis techniques focusing on independent treatment or limiting of cache interference exclude the possible benefit of caches. Multi-threaded tasks may benefit from caches. By virtue of sharing the same address space one thread of a task may cache values on behalf of
another reducing the total execution time to complete both. This positive effect is referred to as the *inter-thread cache benefit* [26].

Currently, only the BUNDLE [26] and BUNDLEP [27] analysis techniques and cache cognizant thread-level scheduling algorithms incorporate the inter-thread cache benefit into WCET and schedulability analysis. These BUNDLE-based approaches are currently limited to a single multi-threaded task. The primary focus of this work is to provide a scheduling algorithm and schedulability test for multi-threaded task sets with multiple tasks, where individual jobs utilize BUNDLE-based scheduling. As the first scheduling algorithm to incorporate BUNDLE-based thread-level scheduling, a non-preemptive algorithm was chosen to avoid necessary modifications to BUNDLE and BUNDLEP. Non-preemptive EDF was selected as the task-level scheduler, as the proposed schedulability test presented in Section 4 is based upon Baruah’s limited-preemption for EDF [6] algorithm.

An additional consideration is made for alternative approaches and the unforeseen benefits to schedulability of thread-level schedulers of non-preemptive multi-threaded jobs. If the WCET of jobs can be expressed as a strictly increasing discrete concave function of the number of threads per job, the schedulability test developed for this work applies without modification to the BUNDLE-based approaches or non-preemptive EDF scheduling.

In the following sections, the key contributions are:

1. A model of hard real-time multi-threaded tasks which is compatible with existing single-threaded models, where tasks sets may be transformed through division of threads.
2. A schedulability test named Threads Per Job (TPJ) that provides a schedulability result and transformed feasible task set if the specified task set could not be scheduled non-preemptively.
3. Proof of TPJ’s optimality with respect to non-preemptive multi-threaded feasibility.
5. An evaluation of over 500,000 task sets, comparing the schedulability ratio of TPJ to those of non-preemptive and (limited) preemptive EDF, with an accompanying implementation available for download [28].

These contributions are presented following the related research of Section 2. Section 3 introduces the proposed model, application of non-preemptive EDF scheduling for thread-level schedulers, and the requirements of task transformation. Section 4 introduces then improves upon the non-preemptive chunk algorithm [6], followed by the TPJ schedulability algorithm and proof of feasibility. Section 5 compares the schedulability ratio of TPJ to other non-preemptive and preemptive scheduling algorithms, before concluding with Section 6.

## 2 Discussion of Related Research

### Single-Threaded Tasks.

The challenge of dealing with the non-uniformity of execution times in real-time systems due to cache misses or hits has received considerable attention [34, 30]. In particular, much of the prior real-time systems work on understanding caches vis-à-vis scheduling has focused upon the contention in the cache due to tasks preempting each other. Roughly speaking, a large majority of this research can be classified into two categories: cache-related preemption delay (CRPD) analysis and deferred/limited-preemption scheduling. The goal of CRPD analysis is to bound the number of cache blocks of a task that need to be reloaded due to evictions caused by a preemtting task. The foundation of CRPD analysis is the development of techniques for counting and bounding the number of blocks affected by preemption; this is achieved by categorizing a task’s cache blocks into sets of useful cache blocks (UCBs) or evicting cache block (ECBs) [17, 31]. The size of these sets can be used as
an upper bound on the cache cost of a preemption. Subsequent research based upon this UCB and ECB categorization has refined these sets and incorporated the CRPD analysis into schedulability analysis [1, 2, 3, 20, 24, 25]. However, please note that these CRPD approaches only quantify the cache effect of preemption into existing scheduling approaches and do not change any scheduling decision based upon the knowledge of preemption.

In limited/deferred-preemption scheduling, a higher-priority task may preempt a lower-priority task only when some condition is satisfied. The overall effect of deferring or limiting preemptions is to reduce the number of times a task may be preempted during its execution. The hope is that by limiting the number of preemptions this will lead to a decrease in the execution time of job due to the cache overhead of preemption. Different conditions for deferring preemptions have been considered. The fixed preemption-point approach [11] selects specific locations in a task code that are most appropriate for the program but preserve the system schedulability. The preemption-threshold scheduling approach [32] sets a threshold that only task with higher-priority than this threshold may preempt a currently-executing lower priority task. The floating preemption-point model [6, 19] computes the maximum time duration that a lower-priority task may delay the preemption of a higher-priority task. Each of the deferred preemption approaches have been shown to limit the number of preemptions but do not incorporate the CRPD overhead cost in its decision on how to defer preemption.

More recently, a line of research has emerged to combine the aspects of CRPD analysis and limited/deferred preemption scheduling by explicitly placing preemption points in the code to minimize CRPD effects. Early heuristics were proposed by Simonson and Patel [23] and Lee et al. [17]. Bril et al. [10] integrated CRPD analysis into preemption-threshold scheduling. Bertogna et al. [8] provide a more formal approach for optimally determining preemptions in programs that can be represented by linear control flowgraphs given the CRPD overhead of each preemption and a bound on the maximum non-preemption region [6]. Later work, extended this to more general control flowgraphs [22] or more precise CRPD characterizations of the preemption costs [14]. However, all of this aforementioned research assumes each task is single-threaded. The techniques proposed in this paper extend the CRPD and limited preemption concepts to scheduling multi-threaded tasks by combining and extending the limited-preemption scheduling results of Baruah [6] to the cache-cognizant thread-level scheduling algorithms that minimize cache contention between threads called BUNDLE [26] and BUNDLEP [27].

Multi-Threaded Tasks. Cache interference amplifies the variation in execution times of multi-threaded task sets. Threads of the same task share cache locations, with the potential to increase misses and hits depending on the order of execution of threads. This variability is an addition to the variation already present when considering CRPD with other tasks.

There are few works we are aware which directly address the inter-thread variability due to caches in multi-threaded task sets. The approaches focus on isolating execution or managing cache behavior. Memory-Centric Scheduling [4] isolates contentious execution by scheduling tasks according to their cache behavior. To create such isolation, tasks must be PREM-compliant [21], with distinct load and execution phases. Cache management utilize techniques that limit the contention in the cache, such as coloring and blocking found in [33]. These approaches come at a cost of modified or restricted executable objects, reduced cache sizes, or additional cache misses of blocked lines. Yet, with these limitations, the inter-thread variability is not accounted for within multi-threaded tasks.

BUNDLE [26] and BUNDLEP [27] address inter-thread variability due to cache interactions. These BUNDLE-based approaches analyze executable object coupled with a cache-cognizant thread-level scheduling algorithm without the added detriment of modified (or restricted)
objects, or cache management penalties. We are not aware of any other technique that addresses inter-thread variability, with the exception of Calandrino’s [13] limited cache spread. However, the results of [13] are strictly empirical.

3 Model

To permit non-preemptive jobs that utilize thread-level schedulers, a new model is proposed in this work. The set of \( n \) multi-threaded tasks is given by \( \tau = \{\tau_0, \tau_1, \ldots, \tau_{n-1}\} \). Each job of a task \( \tau_i = (p_i, d_i, m_i, c_i: \mathbb{N} \rightarrow \mathbb{R}^+) \) has a minimum inter-arrival time of \( p_i \) and relative deadline \( d_i \). For every job release of \( \tau_i \), a positive integer \( m_i \) identical threads are released. Each thread of \( \tau_i \) executes over the same object \( o_i \) on the shared processor. An object is a set of executable machine instructions, mapping to one set of in memory addresses, such that all threads execute the same instruction from the same address. All threads share the same deadline as their job. The worst-case execution time (WCET) of \( \tau_i \) is a function of the number of threads per job, \( c_i(m_i) \).

Scheduling and schedulability analysis proposed in this work relies upon a relationship between the number of threads scheduled per multi-threaded job and the WCET of the job executed non-preemptively. To clarify, the scheduling mechanism proposed in this work precludes preemptions between jobs of different tasks. For threads within a job of a task, a thread-level scheduler may execute threads preemptively. Figure 1a illustrates the scheduling behavior.

In Figure 1a, at \( t = 1 \) a job of \( \tau_2 \) is released. The job of \( \tau_2 \) cannot be preempted by the job of \( \tau_1 \) released at \( t = 5 \). During the execution of \( \tau_2 \), the two threads (given distinct colors) may preempt one another according to the thread-level scheduler, at \( t = 8 \) for instance. Thread-level scheduling and preemption decisions are not prescribed by this work. The thread-level scheduling policies of \( \tau_1 \) and \( \tau_2 \) are independent of the non-preemptive task-level scheduling of non-preemptive EDF used in this work.

Thread-level scheduling algorithms must be characterized by a WCET function \( c_i(m_i) \) for \( m_i \) threads per job and \( c_i(m_i) \) must be strictly increasing discrete and concave (detailed in Subsection 3.2). Thread-level schedulers that produce concave \( c_i(m_i) \) functions establish a relationship between the execution requirements of a task and the number of threads, where the requirement for one job of \( m_i \) threads is less than \( m_i \) jobs of one thread. For BUNDLE-based schedulers, concavity is the result of the inter-thread cache benefit, where \( c_i(m) - c_i(m - 1) \geq c_i(m + 1) - c_i(m) \); it is this relationship the proposed scheduling behavior and analysis seek to exploit.

Not all tasks and thread-level schedulers will produce concave WCET functions. For a task \( \tau_i \) with a convex WCET function (where there is no benefit in grouping threads together), the \( m_i \) threads of \( \tau_i \) may be replaced with \( m_i \) single-threaded tasks. These single-threaded have vacuously concave WCET functions by virtue of executing no more than one thread.

![Figure 1](image_url)
The task set $\tau$ provided by the system designer to schedulability analysis is referred to as the task set specification. Commonly \cite{5,18,6,8,11,12}, task set specifications are immutable in hard-real time models. The number of tasks, their WCET time, period, and deadline are provided by the system designer, not to be changed. Schedulability analysis determines if the task set specification is feasible. In this work, task sets are transformable (obeying some restrictions).

Transformation of a task set exploits the concavity of execution requirements, redistributing the threads of individual tasks to multiple tasks. A greater number of threads per job reduces the WCET of a task but increases the non-preemptive execution requirement. Conversely, a fewer number of threads per task increases the total WCET for all tasks while decreasing the non-preemptive execution requirement. Schedulability analysis in this non-preemptive setting encompasses the search for a distribution of the fixed number threads from the task set specification to a variable number of tasks, resolving the tension between a greater number of tasks and a greater number of threads per task to find a feasible task set.

Under the proposed model, schedulability analysis is a process that begins by considering the current task set named the anterior task set $\hat{\tau}$. If the set is schedulable, the set is unmodified and processing ceases with a positive result. If the task set $\hat{\tau}$ cannot be scheduled as described, the task set is transformed into a posterior task set $\tau$, and processed again as an anterior set. Processing ceases with a negative result when there are no available transformations of $\hat{\tau}$.

Figure 1b illustrates the schedulability analysis process. Division is the transformative operation of the process and is described in Subsection 3.1. The figure highlights the ability of a single task set to be both anterior and posterior to different sets during processing. To aid in explanation, properties of a task may be referred to in terms of the set the task was transformed from and to. By example, if the number of threads assigned to $\tau_i$ in the anterior set $\hat{\tau}$ is reduced by one in the posterior task set $\tau$, the posterior threads of $\tau_i$ may be written as $m_i = \hat{m}_i - 1$.

As a process, schedulability analysis of the specified task set serves two purposes under this model. The first, is to determine if there exists a posterior task set which is feasible. Second, to produce the feasible posterior task set if one exists. It is the feasible posterior task set $\tau$ found by schedulability analysis that is then deployed on the target architecture. From the system designer’s perspective, each task $\tau_i \in \tau$ of the specified task set is a request to execute $m_i$ threads of the object $o_i$ with shared periods $p_i$ and deadlines $d_i$ for any posterior task set $\tau$. A task set specification is flexible, for one object there may be multiple tasks with variable numbers of threads per job. However, the specified $m_i$ of a task is a ceiling on any $m_i$ of a posterior task.

### 3.1 Dividing and Task Parts

A task set may be transformed by dividing tasks of the set. Dividing a task reduces the number of threads executed by each job, splitting the anterior task into two or more tasks in the posterior set.

> **Definition 1 (Task Division).** In the anterior task set $\hat{\tau}$, a task $\tau_i = (p_i, d_i, c_i(m_i))$ may be divided into two (or more) posterior tasks $\tau_j$ and $\tau_k$ with three restrictions: 1.) the periods of $\tau_j$ and $\tau_k$ are equal to the period of $\tau_i$, 2.) the relative deadlines of $\tau_j$ and $\tau_k$ are equal to the deadline of $\tau_i$, 3.) the sum of threads of $\tau_j$ and $\tau_k$ are equal to $\tau_i$, and 4.) the objects of $\tau_i$, $\tau_j$, and $\tau_k$ are equal. Enumerated, the restrictions are:

1. $p_i = p_j = p_k$
2. $d_i = d_j = p_k$
3. $m_i = m_j + m_k$
4. $o_i = o_j = o_k$
Definition 2 (Partial Tasks). When an anterior task \( \tau_i \) is divided into \( \tau_j \) and \( \tau_k \) posterior tasks, \( \tau_j \) and \( \tau_k \) are referred to as partial tasks or parts of \( \tau_i \).

Definition 3 (Partial Task Set). For convenience, the set of posterior tasks of \( \tau_i \) is denoted \( \Phi_i \) and called the partial task set of \( \tau_i \), where \( m_i = \sum_{\tau_k \in \Phi_i} m_k \).

3.2 Worst-Case Execution Time Function Growth

Motivation for the task model and schedulability analysis process proposed in this work stems from the inter-thread cache benefit of BUNDLE-based scheduling [26, 27]. The previous works [26, 27] are limited to a single task; this work extends the method (non-preemptively) to multiple tasks. Schedulability analysis for BUNDLE-based scheduling algorithms produce, for each task \( \tau_i \), a worst-case execution time combined with cache overhead (WCETO) function \( c_i(m) \) in terms of \( m \) the number of threads per job scheduled in a cache-cognizant manner. For tasks that benefit from BUNDLE-based scheduling and analysis, \( c_i(m) \) is a strictly increasing discrete concave function. Tasks that do not are made vacuously concave by restricting jobs to release one thread.

In the WCETO analysis of BUNDLE and BUNDLEP, threads are assigned to paths through the conflict-free region graph of the executable object which maximize their contribution to \( c_i(m_i) \). When considering the addition of a thread \( m_i + 1 \), only the greatest increase in \( c_i(m_i) \) is permitted. Subsequently, the addition of thread \( m_i + 2 \) must increase \( c_i(m_i) \) by less than or equal to the increase from \( m_i + 1 \) or the increase of \( m_i + 1 \) would not have been maximal. Therefore, for any \( m_a < m_b < m_c \) the point \((m_b, c_i(m_b))\) lies above the straight line described by \((m_a, c_i(m_a))\) and \((m_c, c_i(m_c))\) - subsequently, \( c_i(m_i) \) is concave.

A consequence of \( c_i(m) \)'s strictly increasing discrete concavity is a limit on the increase of the WCET as the number of threads increases. This property is referred to as the concave restricted growth (concave growth for brevity) of \( c_i(m) \) and is leveraged in Sections 4 and 5.

Property 1 (Concavity Restriction on WCET Growth). For a strictly increasing discrete concave WCET function \( c_i(m) \):

\[
\forall m \in \mathbb{N}^+ \; | \; c_i(m) - c_i(m - 1) \geq c_i(m + 1) - c_i(m) 
\]

It then follows for \( m_x \geq m_y > 0 \)

\[
\begin{align*}
c_i(m_x + 1) - c_i(m_x) & \leq c_i(m_x) - c_i(m_x - 1) \\
& \leq c_i(m_x - 1) - c_i(m_x - 2) \\
& \ldots \\
& \leq c_i(m_y) - c_i(m_y + 1) \\
& \leq c_i(m_y) - c_i(m_y - 1)
\end{align*}
\]

A WCET function \( c_i(m) \) that obeys Property 1, will produce a value for \( c_i(m + 1) \) threads which is greater than \( c_i(m) \). The difference between \( c_i(m + 1) \) and \( c_i(m) \) must be less than or equal to the difference of \( c_i(m) \) and \( c_i(m - 1) \). As the number of threads increase, \( c_i(m) \) increases at a decreasing (or stable) rate.

For the purposes of comparison and evaluation in Section 5, an upper bound on the growth of \( c_i(m) \) is called the growth factor \( F_i \) of \( \tau_i \). Growth factors relate the WCET of one thread \( c_i(1) \) to the WCET of an arbitrary number of threads \( c_i(m) \) for \( m > 0 \). A growth factor \( F_i \in [0, 1] \), for a task \( \tau_i \), is a real number that satisfies Equation 2.

Definition 4 (Growth Factor for \( \tau_i \)).

\[
\forall m \; | \; c_i(m) \leq c_i(1) + (m - 1) \cdot F_i \cdot c_i(1) 
\]
For an \( F \) satisfying Equation 2, the pessimistic upper bound provides a linear function that can be rearranged to find an upper bound on the WCET of one thread in terms of \( m \) threads. The result is Equation 3, which will be used in the evaluation Section 5 when constructing task sets. Note, since \( m \in \mathbb{N} \) each increase of \( m \) increases \( c_i(m) \) by \( F \cdot c_i(1) \).

\[
c_i(m) = c_i(1) + (m - 1) \cdot F \cdot c_i(1) \tag{3}
\]

## 4 Non-Preemptive EDF Schedulability

Preemptive earliest deadline first (EDF) schedulability analysis for sporadic task sets has been well studied [18, 5, 15]. In the fully preemptive setting for which the algorithm is optimal, the overhead of a large number of preemptions may be a detriment to schedulability. Baruah [6] addresses this concern with an algorithm for calculating the non-preemptive chunk size \( q_i \) of each task \( \tau_i \). The non-preemptive chunk size \( q_i \) guarantees that task \( \tau_i \) may execute up to \( q_i \) time units non-preemptively without introducing a deadline miss for any task in \( \tau \) scheduled by preemptive EDF.

Section 4.3 introduces the non-preemptive feasibility algorithm Thread Per Job (TPJ) based upon the non-preemptive chunks algorithm from [6]. TPJ differs from the non-preemptive chunks algorithm by requiring the non-preemptive chunk size \( q_i \) of each task \( \tau_i \) to be greater than or equal to its WCET: \( c_i(m_i) \leq q_i \). As such, all jobs can be scheduled non-preemptively without fear of a deadline miss. To clearly convey TPJ, a description of the non-preemptive chunks algorithm and its dependencies is provided in the immediate subsection. Subsection 4.2 describes, by example, the available improvements to the non-preemptive chunks algorithm [6]. Subsection 4.4 defines and proves TPJ’s optimality.

### 4.1 Non-Preemptive Chunks

The non-preemptive chunks algorithm depends on the demand bound function, EDF feasibility, ordering of absolute deadlines, and slack for the task set \( \tau \). Ordered absolute deadlines are given by \( \{D_1, D_2, \ldots\} \) with \( D_n < D_{n+1} \) for all \( n \), where each task \( \tau_i \in \tau \) contributes deadlines \( D = k \cdot p_i + d_i \) for \( k \in \mathbb{Z}^+ \).

For a sporadic task \( \tau_i \), the demand bound function for a task \( \text{DBF}(\tau_i, t) \) is an upper bound on the amount of execution requirement generated from jobs released by \( \tau_i \) over \( t \) units of time. The demand bound function is presented as Equation 4 as \( \text{DBF}(\tau_i, t) \) modified from [5] to suit the task set model used in this work.

▶ **Definition 5** (Demand Bound Function for a Task \( \tau_i \) and Interval \( t \)).

\[
\text{DBF}(\tau_i, t) = \max \left( 0, \left( \left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1 \right) \cdot c_i(m_i) \right) \tag{4}
\]

When necessary for brevity, Equation 5 will be used to represent the sum of demand of all tasks over an interval of length \( t \).

▶ **Definition 6** (Demand Bound Function for the Task Set \( \tau \) and Interval \( t \)).

\[
\text{DBF}(\tau, t) = \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t) \tag{5}
\]

Slack of the task set \( \tau \) at deadline \( D_k \) is given by Equation 6. Intuitively, slack is the minimum time the processor will be idle over an interval. It is the difference between the demand over the interval and the length of the interval.
Definition 7 (Slack at Deadline $D_k$).

$$\text{SLACK}(D_k) = \min_{j \leq k} \left( D_j - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k) \right) \quad (6)$$

For EDF, feasibility is determined by examining increasing time intervals and calculating the demand and supply. If demand exceeds supply, the system is infeasible. Equation 7 provides a formal definition of feasibility for the task set $\tau$.

Definition 8 (EDF Feasibility Demand Bound Test).

$$\forall t \geq 0, \left( \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t) \right) \leq t \quad (7)$$

In [6], the number of time instants tested by Equation 7 is limited to the values of the ordered set of absolute deadlines $\{D_1, D_2, \ldots\}$. The ordered set of absolute deadlines is an infinite set, impractical for feasibility test. There is an upper bound on the value of all time instants (absolute deadlines) that must be tested and is denoted $T^*(\tau)$. Taken from [15], $T^*(\tau)$ is given by Equation 8 below. Among all tasks the largest deadline is $d_{\text{max}} = \max_{\tau_i \in \tau} (d_i)$.

Utilization of $\tau_j$ is defined as $U_j = \frac{c_j \cdot m_j}{p_j}$. Among all tasks, the greatest difference of period and deadline is given by $\Delta_{\text{max}} = \max_{\tau_i \in \tau} (p_i - d_i)$. The hyper-period of all tasks (the least common multiple of all relative deadlines) is given by $P$.

Definition 9 (Feasibility Test Bound $t$ for $\tau$).

$$T^*(\tau) = \min \left( P, \max \left( d_{\text{max}}, \frac{1}{1 - U_j} \cdot \Delta_{\text{max}} \cdot \sum_{i=0}^{n-1} U_i \right) \right) \quad (8)$$

The non-preemptive chunks algorithm from [6] is presented (with additional details) as pseudocode in Algorithm 1 and named np-chunks. In addition to determining if the task set is schedulable under EDF, the algorithm produces a non-preemptive chunk size $q_j$ for each task $\tau_j \in \tau$. Jobs of $\tau_j$ may execute up to $q_j$ time units non-preemptively without negatively impacting schedulability. This setting, where a task $\tau_j$ may execute non-preemptively for some period of time $q_j$ is referred to as limited-preemption.

Algorithm 1 Non-Preemptive Chunks (np-chunks).

1: \text{SLACK}(D_1) \leftarrow D_1 - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_1)
2: \textbf{for} $\tau_j \in \{\tau_i \in \tau \mid (d_i = D_1)\}$ \textbf{do}
3: \hspace{1cm} $q_j \leftarrow c_j \cdot m_j$
4: \textbf{end for}
5: \textbf{for} $k \in \{D_2, D_3, \ldots\}$ \textbf{do}
6: \hspace{1cm} \textbf{if} $D_k > T^*(\tau)$ \textbf{then}
7: \hspace{2cm} \text{return feasible}
8: \hspace{1cm} \textbf{end if}
9: \hspace{1cm} \text{SLACK}(D_k) \leftarrow \min \{\text{SLACK}(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k)\}
10: \hspace{1cm} \textbf{if} \text{SLACK}(D_k) < 0 \textbf{then}
11: \hspace{2cm} \text{return infeasible}
12: \hspace{1cm} \textbf{end if}
13: \hspace{1cm} \textbf{for} $\tau_j \in \{\tau_i \in \tau \mid (d_i = D_k)\}$ \textbf{do}
14: \hspace{2cm} $q_j \leftarrow \text{SLACK}(D_k)$
15: \hspace{1cm} \textbf{end for}
16: \textbf{end for}
For a detailed description of NP-CHUNKS refer to [6]. To summarize, NP-CHUNKS begins by seeding the slack of the smallest interval $D_1$ and the non-preemptive chunk size of tasks with the smallest relative deadline equal to their WCET. During each iteration of $D_k \in \{D_2, D_3, \ldots \}$, the slack for the interval $D_k$ is calculated as the minimum of the current slack and the previous slack value. If there is less than zero slack, the system is infeasible. If the slack is zero or greater, each task with relative deadline equal to the current interval size is assigned the available slack as the task’s non-preemptive chunk size. A task $\tau_j$ is assigned a non-preemptive chunk once, before assignment $q_j = \emptyset$ afterwards $q_j \neq \emptyset$. If the interval being examined $D_k$ exceeds $T^*(\tau)$, the task set must be schedulable.

### 4.2 Improving the Non-Preemptive Chunk Size

From the description of NP-CHUNKS in [6], there is an opportunity to improve the available slack for each of the $k$ deadlines considered. Algorithm 1 is pessimistic in the amount of available slack at any deadline $D_k$. To illustrate, consider the task set and intermediate values described by Table 1.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$p_i$</th>
<th>$d_i$</th>
<th>$m_i$</th>
<th>$c_i(m_i)$</th>
<th>$D_k$</th>
<th>$\tau_j : d_j = D_k$</th>
<th>DBF$(\tau, D_k)$</th>
<th>SLACK$(D_k)$</th>
<th>$q_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_0$</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>$D_1 = 2$</td>
<td>$\tau_0$</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>$D_2 = 3$</td>
<td>$\tau_1$</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td>$\tau_2$</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

There are three tasks in the task set of Table 1, with utilization of approximately 0.92. For $\tau_0$, initialization assigns a non-preemptive chunk of $q_0 = 1$ time units. By observation, after release $\tau_0$ may be delayed from execution by at most one time unit or it will miss its deadline. Consequently, the non-preemptive chunk size available to $\tau_1$ and $\tau_2$ is 1. As such NP-CHUNKS would be expected to find $q_0 = 1, q_1 = 1, q_2 = 1$.

Note, it is not possible for $\tau_0$ to be blocked for 1 or more time units if both $\tau_1$ and $\tau_2$ execute non-preemptively for 1 time unit each. If $\tau_0$ is blocked for less than 1 time unit by $\tau_1$, then $\tau_0$ will be the highest priority task when $\tau_1$ completes (similarly for $\tau_2$). It is impossible for $\tau_0$ to be blocked 1 time unit or more by $\tau_1$ or $\tau_2$. $\tau_0$ would have to be released at the same time instant as $\tau_1$ or $\tau_2$ and $\tau_1$ or $\tau_2$ would have to execute before $\tau_0$, since the relative deadline of $\tau_0$ is less than the other two, limited-preemption EDF executes $\tau_0$: the task with earliest absolute deadline.

For $\tau_0$, $q_0$ is calculated as expected $q_0 = c_0(m_0) = 1$, by Lines 2-4 of Algorithm 1. However, $\tau_1$ has a non-preemptive chunk size of $q_1 = 0$. The reason is Line 9, where $\text{SLACK}(D_2)$ is calculated which includes the execution demand of $\tau_1$ and $\tau_2$. Slack is an upper bound on the non-preemptive chunk size assigned to a task (in this case $\tau_1$). Giving a task the available slack permits the task to execute longer, delaying higher priority jobs from executing in the interval by delaying them for as much time as there is slack.

By example in Table 1, the available slack for $\tau_1$ is determined from the interval of length $D_2 = 3$. The execution requirement of $\tau_1$ and $\tau_2$ is included in $\text{DBF}(\tau_1, 3)$ because $d_1 = d_2 = 3$. Thus $\text{SLACK}(D_2)$ is zero. Since $\tau_1$’s execution requirement is already included, it cannot further interfere over the interval $D_2$. Furthermore, $\tau_1$ must have executed some portion without being preempted or the system would not be schedulable. Inclusion of $\tau_1$’s execution requirement within the interval over which slack is calculated for is pessimistic with respect to the non-preemptive chunk $q_1$ in this specific example, and $q_j$ in general.

In the pseudocode implementation of NP-CHUNKS adopted from [6], Line 9 calculates the non-preemptive chunk size according Equation 9 (Equation 7 of Theorem 1 in [6]).
Comparing Line 9 of Algorithm 1 to Equation 9 a mismatch between the algorithm and the infeasibility test is illuminated.

Definition 10 (Infeasibility Test, Equation 7, from [6]).

\[ \exists \tau_j, t \in [0,d_j) \mid t < q_j + \sum_{i=0, i \neq j}^{n-1} DBF(\tau_i, t) \tag{9} \]

If the condition of Equation 9 is satisfied for a task set \( \tau \), the task set is unschedulable given a limited-preemption task set with assigned non-preemptive chunks \( q \). The interval considered in the demand of Equation 9 is over \([0,d_j)\). The demand used in Algorithm 1 to calculate \( q_j \) is over the interval \([0,d_j]\). Extending the interval to include \( d_j \) introduces the pessimism identified by the example and is not required by Equation 9.

Table 1 illustrates the pessimism of np-chunks found in [6]. The example uses the notation of assigning non-preemptive chunks to individual tasks from [6]. A later work [7] uses a different notation, assigning non-preemptive chunks to interval lengths for the remaining execution of a job. The conceptual pessimism of including demand for tasks with deadline equal to the current interval (described by Table 1) is also found in [7].

4.3 Threads per Job (TPJ) Scheduling Algorithm

In this work, the np-chunks algorithm is modified for several purposes. First, the unnecessary pessimism is removed from chunk calculations. Second, the schedulability test is adapted to the model used herein. Lastly, when a given assignment of tasks and threads are infeasible, tasks are divided (when possible) to fit into their chunks. The division process is repeated until the task set is feasible, or no possible divisions remain and the task set is reported as infeasible. The algorithm is named the Threads Per Job (TPJ) scheduling algorithm.

A full description of TPJ is presented at the end of this subsection. To reach the complete description, an intermediate algorithm named Bigger Non-Preemptive Chunks (BNC) is presented as pseudocode in Algorithm 2. BNC removes the pessimism described in Section 4.2. The algorithm takes advantage of a property of the demand function DBF(\( \tau, t \)) noted in [6].

Property 2 (Demand Change). Demand for a task does not change for values of \( t \) that do not equal an absolute deadline. In terms of the set of ordered absolute deadlines, 
\[ DBF(\tau, D_{i-1}) = DBF(\tau, D_i - \epsilon), \text{ for } 0 < \epsilon \leq (D_i - D_{i-1}). \]

Algorithm 2 Bigger Non-Preemptive Chunks (BNC).

1: \text{SLACK}(D_0) \leftarrow \infty
2: \text{for } k \in \{D_1, D_2, D_3, \ldots \} \text{ do}
3: \quad \text{if } D_k > T^*(\tau) \text{ then}
4: \quad \quad \text{return feasible}
5: \quad \text{end if}
6: \quad \text{SLACK}(D_k) \leftarrow \min\left\{ \text{SLACK}(D_{k-1}), D_k - \sum_{\tau_i \in \tau} DBF(\tau_i, D_k) \right\}
7: \quad \text{if } \text{SLACK}(D_k) < 0 \text{ then}
8: \quad \quad \text{return infeasible}
9: \quad \text{end if}
10: \quad \text{for } \tau_j \in \{\tau_i \in \tau \mid (d_i = D_k)\} \text{ do}
11: \quad \quad q_j \leftarrow \min(c_j(m_j), \text{SLACK}(D_{k-1}))
12: \quad \text{end for}
13: \text{end for}
Line 11 of Algorithm 2 implements the improvement of \texttt{bnc} over \texttt{np-chunks}. The non-preemptive chunk $q_j$ of task $\tau_j$ is taken from the slack of the previous interval $D_{k-1}$ or the task’s WCET $c_j(m_j)$, whichever is smaller. The algorithm verifies the condition set by Equation 9, selecting the correct interval length by Property 2, which precludes the inclusion of $\tau_j$’s execution requirement in the interval (and other tasks with deadline $D_k$).

Algorithm 3 (Threads-Per-Job (TPJ)).

1: \texttt{SLACK}(D_0) \leftarrow \infty
2: \texttt{for} $k \in \{D_1, D_2, D_3, \ldots\}$ \texttt{do}
3: \hspace{1em} \texttt{if} $D_k > T^*(\tau)$ \texttt{then}
4: \hspace{2em} \texttt{return \texttt{feasible}}
5: \hspace{1em} \texttt{end if}
6: \hspace{1em} \texttt{for} $\hat{\tau}_j \in \{\tau_i \in \tau \mid (d_i = D_k)\}$ \texttt{do}
7: \hspace{2em} \texttt{if} \texttt{SLACK}(D_{k-1}) < \hat{c}_{\hat{\tau}_j}(1) \texttt{then}
8: \hspace{3em} \texttt{return \texttt{infeasible}}
9: \hspace{2em} \texttt{end if}
10: \hspace{1em} \Phi_j \leftarrow \{\hat{\tau}_j\}
11: \hspace{1em} \texttt{if} \texttt{SLACK}(D_{k-1}) < \hat{c}_{\hat{\tau}_j}(\hat{m}_j) \texttt{then} \quad \triangleright \text{Jobs must be divided}
12: \hspace{2em} \Phi_j \leftarrow \texttt{DIVIDE}(\hat{\tau}_j, \texttt{SLACK}(D_{k-1}))
13: \hspace{1em} \tau \leftarrow \tau \setminus \hat{\tau}_j \quad \triangleright \text{Anterior task } \hat{\tau}_j \text{ is represented by } \Phi_j
14: \hspace{1em} \tau \leftarrow \tau \cup \Phi_j \quad \triangleright \text{Partial tasks include all threads of } \hat{\tau}_j
15: \hspace{1em} \texttt{end if}
16: \hspace{1em} \texttt{for} $\tau_j \in \Phi_j$ \texttt{do}
17: \hspace{2em} $q_j \leftarrow c_j(m_j)$
18: \hspace{1em} \texttt{end for}
19: \hspace{1em} \texttt{end for}
20: \texttt{SLACK}(D_k) \leftarrow \min(\texttt{SLACK}(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \texttt{DBF}(\tau_i, D_k))
21: \hspace{1em} \texttt{if} \texttt{SLACK}(D_k) < 0 \texttt{then}
22: \hspace{2em} \texttt{return \texttt{infeasible}}
23: \hspace{1em} \texttt{end if}
24: \texttt{end for}

The Threads per Job scheduling Algorithm 3, is a modification of \texttt{bnc} from limited-preemption EDF (EDF-LP) scheduling to non-preemptive EDF (EDF-NP). Input to the schedulability test is a task set specification $\tau$, if TPJ returns a \texttt{feasible} result there exists a posterior task set which can be scheduled by non-preemptive EDF and the posterior task set is returned as $\tau$. An \texttt{infeasible} result indicates that TPJ could not guarantee $\tau$ would be schedulable by EDF-NP for any posterior task set. Since non-preemptive EDF is not optimal with respect to feasibility [12], TPJ is a sufficient test but cannot be necessary.

Algorithm 3 (TPJ) modifies \texttt{bnc}, the modifications are limited to Lines 6-19. An additional benefit of \texttt{bnc} removing the pessimism of each $q_j$, is that each $q_j$ can be calculated without consideration of the current task $\tau_j$ and the demand at $D_k$. Chunk values depend on the demand of $D_{k-1}$ instead. This permits an efficient implementation of TPJ by moving the slack calculation of the current interval to the end of each iteration. Otherwise, if slack were calculated earlier in each iteration, the changes to demand resulting from Lines 6-19 would force the demand and slack of $D_k$ to be recalculated.

The first notable change to \texttt{bnc} is introduced on Line 7, comparing the available slack to the WCET of a single thread of $\hat{\tau}_j$. If there is insufficient slack to execute just one thread of $\hat{\tau}_j$ to completion, the task cannot be executed non-preemptively for any number of threads and the task set is infeasible non-preemptively.
Lines 11-15 introduce several subtle changes. For clarity, it is simpler to discuss the negative case ($\text{SLACK}(D_{k-1}) \geq \hat{c}_j(\hat{m}_j)$) before the positive. When there is sufficient slack for $\hat{m}_j$ threads to execute without preemption, $\hat{\tau}_j$ is given its full WCET ($\hat{c}_j(\hat{m}_j)$) as its non-preemptive chunk. In other words, no division of $\hat{\tau}_j$ is required and the posterior task set $\tau$ is unchanged (with respect to $\hat{\tau}_j$). Lines 11-15 are avoided, the algorithm progresses to the next task such that $d_j = D_k$.

However, in the positive case on Line 11 (when $\text{SLACK}(D_{k-1}) < \hat{c}_j(\hat{m}_j)$), $\hat{m}_j$ threads of $\hat{\tau}_j$ cannot feasibly execute without being preempted. Therefore, $\hat{\tau}_j$ must be divided. The \textsc{divide} procedure creates a partial task $\Phi_j$ set of $\hat{\tau}_j$, such that all tasks $\tau_p \in \Phi_j$ will complete within the available slack $c_p(m_p) \leq D_{k-1}$. The posterior task set $\tau$ has $\hat{\tau}_j$ removed, and is replaced by the partial set $\Phi_j$ maintaining the specified number of threads for $\hat{\tau}_j$.

For any task $\hat{\tau}_j$, the task is transformed into a partial task set $\Phi_j$ and assigned a non-preemptive chunk only once in the iteration where the absolute deadline $D_k$ is equal to the relative deadline of the task: $D_k = \hat{d}_j$. Since tasks of $\tau$ are evaluated in strictly increasing absolute deadline order, the impact on demand and non-preemptive chunk sizes of processing $\hat{\tau}_j$ exclusively impacts demand for larger intervals $D_k > D_k$ non-preemptive chunk sizes for tasks $\tau_l \in \tau$ with greater relative deadlines $d_l > d_k$.

\textbf{Property 3 (Divisions of $\hat{\tau}_j$ Exclusively Impacts Interval of Length $t \geq \hat{d}_j$).} Division of $\hat{\tau}_j$ into the partial set $\Phi_j$, and replacing $\hat{\tau}_j$ in $\tau$ with $\Phi_j$ will impact demand exclusively for intervals of length $D_k \geq \hat{d}_j$, slack of absolute deadlines $D_k > \hat{d}_j$, and therefore non-preemptive chunk values $q_e$ for tasks $\tau_e \in \tau$ with relative deadlines $d_l \geq D_k$.

By definition of $\text{DBF}(\hat{\tau}_j,t)$, no task of $\Phi_j$ or $\hat{\tau}_j$ can impact the task set $\tau$ demand $\text{DBF}(\tau,t)$ when $t < \hat{d}_j$. Thus replacing $\hat{\tau}_j$ in $\tau$, only affects the demand of intervals with length $\hat{d}_j$ or greater. Slack over the interval $D_k$ is calculated from exclusively shorter intervals. Since the demand of the current interval $D_k$ does not influence the slack at $D_k$, replacing $\hat{\tau}_j$ in $\tau$ only affects the slack of intervals with length greater than $D_k$. Non-preemptive chunk sizes are assigned based on the available slack, and only those assigned for an interval of length greater than $D_k$ can be affected by replacing $\hat{\tau}_j$ in $\tau$.

\begin{algorithm}
\caption{\textsc{divide}.}
\begin{algorithmic}[1]
\Procedure{\textsc{divide}}{\hat{\tau}_j, q}
\State $\Phi_j \leftarrow \{\}$
\State $m \leftarrow \arg\max_{m \in \mathbb{Z}^+} (\hat{c}_i(m) \leq q)$
\State $r \leftarrow \hat{m}_j$
\While{$r > 0$}
\State $m_p \leftarrow \min(r,m)$
\State $\tau_p \leftarrow (\hat{p}_j, \hat{d}_j, m_p, \hat{c}_j)$ \Comment{Posterior task, same period, deadline, WCET function.}
\State $\Phi_j \leftarrow \Phi_j \cup \tau_p$
\State $r \leftarrow r - m_p$
\EndWhile
\State return $\Phi_j$
\EndProcedure
\end{algorithmic}
\end{algorithm}

On Line 12 of the \textsc{tpj} Algorithm 3, the task $\hat{\tau}_j$ is divided into $\Phi_j$ by the \textsc{divide} procedure. Pseudocode of \textsc{divide} is given by Algorithm 4. The number of tasks in $\Phi_j$ are determined by the maximum number of threads $m$ of $\hat{\tau}_j$ that can execute non-preemptively within $q$ time units. Each task $\tau_k \in \Phi_j$ is assigned $m$ threads of $\hat{\tau}_j$ or however many remain, whichever is less. The result is that each task set has the following properties.
**Property 4** (Partial Task Sets Returned from **DIVIDE**). The partial task set $\Phi_j$ of an anterior task $\tau_j$ for a specific $q$ value (and related maximum threads assigned per job $m$ such that $c_j(m) \leq q$) contains posterior tasks where:

1. The exact number of posterior tasks is $|\Phi_j| = \lceil \frac{\hat{m}_j}{m} \rceil$
2. Exactly $\lceil \frac{\hat{m}_j}{m} \rceil$ tasks of $\Phi_j$ are assigned $m$ threads per job.
3. There is at most one task $\tau_g \in \Phi_j$ with exactly $m_g = \hat{m}_j \mod m$ threads.

### 4.4 Non-Preemptive Feasibility of TPJ and **DIVIDE**

The **DIVIDE** Algorithm 4 creates a partial task set $\Phi_j$ for an anterior task $\hat{\tau}_j$, assigning as many threads to each task in $\Phi_j$ as possible. Upon returning $\Phi_j$ to TPJ, $\hat{\tau}_j$ is replaced in the task set $\tau$. Algorithm 4 is one method of dividing of $\hat{\tau}_j$ which TPJ could employ when creating the posterior task set $\tau$. This section justifies **DIVIDE**’s method by demonstrating the effect on schedulability and optimality of TPJ.

This section’s ultimate objective is to clearly convey Theorem 5; concluding that TPJ is optimal with respect to task-level non-preemptive multi-threaded feasibility. The theorems that precede Theorem 5 establish minimal demand and WCET sums for partial sets created by **DIVIDE** necessary to illustrate TPJ’s optimality.

Non-preemptive EDF scheduling of jobs of multiple threads ordered by a thread-level scheduler (such as **BUNDLE** or **BUNDLEP**) allows preemptions between threads of the same job but precludes preemptions between jobs. Each task benefits from the advantages of thread-level scheduling by the exclusive use of the processor and shared resources. Since task set specifications may be divided, a specification is feasible when threads of the specification $\hat{\tau}$ may be assigned to tasks such that the posterior task set $\tau$ is feasible by EDF-NP.

**Definition 11** (npm-feasible). A task set specification $\hat{\tau}$ is task-level non-preemptive multi-threaded feasible (npm-feasible) if there exists a posterior task set $\tau$ of $\hat{\tau}$ such that all multi-threaded jobs scheduled by EDF-NP will always meet their deadlines.

For the theorems that follow, unless necessary to discriminate between anterior and posterior tasks, the anterior task $\hat{\tau}_i$ will be written $\tau_i$. The sum of the demand of the partial tasks of $\tau_i$ for an interval of length $t$ is $\sum_{\tau_k \in \Phi_i} DBF(\tau_k, t)$.

**Theorem 1** (Minimal Demand of Partial Task Sets Over All Intervals). For a partial task set $\Phi_i$ of an anterior task $\tau_i$ with $m_i$ threads, minimizing $\sum_{\tau_k \in \Phi_i} DBF(\tau_k, d_i)$ minimizes $\sum_{\tau_k \in \Phi_i} DBF(\tau_k, t)$ for all $t \geq 0$.

**Proof.** Provided into two parts, when $t < d_i$ and $t \geq d_i$. The first portion is a simple direct argument. The second portion is by contradiction.

**Part 1:** When $t < d_i$, $0 = \sum_{\tau_k \in \Phi_i} DBF(\tau_k, t)$. By definition of the demand bound function (Equation 4) the execution requirement of a task is zero before the first possible deadline. All tasks $\tau_k \in \Phi_i$ share the same relative deadlines $d_k = d_i$ and absolute deadlines because $p_k = p_i$. These follow from the definition of division (Definition 1) and partial tasks (Definition 2). Since $t < d_i$, $DBF(\tau_k, t) = 0$ for all $\tau_k \in \Phi_i$. Therefore, $\sum_{\tau_k \in \Phi_i} DBF(\tau_k, t)$ will be minimal (exactly zero) when $t < d_i$, regardless of $\sum_{\tau_k \in \Phi_i} DBF(\tau_k, d_i)$.

**Part 2:** When $t \geq d_i$, assume $\sum_{\tau_k \in \Phi_i} DBF(\tau_k, d_i)$ is minimal and $\sum_{\tau_k \in \Phi_i} DBF(\tau_k, t)$ is not minimal. Since all partial tasks $\tau_k \in \Phi_i$ share absolute deadlines (as described in Part 1), demand for each task $DBF(\tau_k, t)$ increases only for values of $t$ that equal absolute deadlines. Furthermore, the execution requirement of every $\tau_k$ increases exactly by $c_k(m_k)$ for each
absolute deadline of \( \tau_i = \{D_1, D_2, \ldots\} \):

\[
\text{DBF}(\tau_k, D_1) = 1 \cdot c_k(m_k) \\
\text{DBF}(\tau_k, D_2) = 2 \cdot c_k(m_k) \\
\ldots \\
\text{DBF}(\tau_k, D_z) = z \cdot c_k(m_k)
\]

Utilizing Property 2, for \( t \geq d_i \) and \( D_z \), where \( D_z \) is the greatest absolute deadline of \( \tau_i \) less than or equal to \( t \) \((D_z \leq t)\): 

\[
\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d) = \sum_{\tau_k \in \Phi_i} z \cdot \text{DBF}(\tau_k, d_i) = z \cdot \sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d_i)
\]

Because \( z \) depends on \( t \) (and is completely independent of the division of the partial task set), if \( \sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, t) \) were not minimal then \( \sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d_i) \) could not be minimal, contradicting the assumption.

Combining Parts 1 and 2, when the demand for the partial tasks of \( \tau_i \) is minimized for the interval \( d_i \), the demand of partial tasks of \( \tau_i \) is minimized for all intervals of length \( t \geq 0 \).

\[
\textbf{Corollary 2} \ (\text{Minimal WCET Sum of } \Phi_i \text{ Minimizes Demand Over the Interval } d_i). \text{ The demand of } \Phi_i \text{ over the interval } d_i \text{ is minimized when the sum of WCET of } \Phi_i \text{ is minimized.}
\]

\textbf{Proof.} Following directly from Theorem 1, where the demand over the interval \( d_i \) of each task \( \tau_k \in \Phi_i \) is given by \( \text{DBF}(\tau_k, d_i) = 1 \cdot c_k(m_k) = c_k(m_k) \). Then,

\[
\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d_i) = \sum_{\tau_k \in \Phi_i} c_k(m_k)
\]

Thus, minimizing \( \sum_{\tau_k \in \Phi_i} c_k(m_k) \) minimizes \( \sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d_i) \).

\[
\textbf{Corollary 3} \ (\text{Minimal WCET Sum of } \Phi_i \text{ Minimizes Demand Over all Intervals } t \geq 0). \text{ The demand of } \Phi_i \text{ over all interval } t \geq 0 \text{ is minimized when the sum of WCET of } \Phi_i \text{ is minimized.}
\]

\textbf{Proof.} Following directly from Theorem 1 and Corollary 2.

\[
\textbf{Definition 12} \ (\text{Assumptions of Theorem 4}). \text{ For the following theorem, there are several assumptions that must be upheld for the result to be valid. These assumptions are consequences of the non-preemptive setting and requirements of the task set specification.}
\]

1. All tasks \( \tau_i \) must be characterized by strictly increasing discrete concave WCET function \( c_i(m_i) \).
2. Any task \( \tau_i \in \tau \) where \( c_i(m_i) > q_i \) is not schedulable non-preemptively. Consequently, no assignment of \( m_i \) may cause \( c_i(m_i) > q_i \) or the task set is infeasible.
3. The greatest number of threads assigned to a task \( \tau_i \) such that \( c_i(m_i) \leq q_i \) is named \( m = \text{argmax} (c_i(m) \leq q_i). \)

\[
\textbf{Theorem 4} \ (\text{Minimal Sum of WCET of } \Phi_i \text{ for any } q \text{ by } \text{DIVIDE}). \text{ For an anterior task } \hat{\tau}_i \text{ and non-preemptive chunk size } q, \text{ DIVIDE will produce a partial task set } \Phi_i \text{ with minimum WCET sum among all possible partial task sets of } \hat{\tau}_i.
\]
While there are two (or more) tasks of \( tp_j \) returns feasible if and only if there exists an npm-feasible posterior task set \( \hat{\tau}_i \) that differs, ie. \( \Phi_i \neq \Phi_k \) and

\[
\sum_{\tau_k \in \Phi_k} c_k(m_k) < \sum_{\tau_j \in \Phi_i} c_j(m_j)
\]

By Property 4 of partial tasks created by \( \text{DIVIDE} \), \( \Phi_i \) will have at most one task with less than \( m \) threads assigned to it. For \( \Phi_k \) to differ, it must have at least two tasks with less than \( m \) threads assigned to them. Call these two tasks with less than \( m \) threads \( \tau_x, \tau_y \in \Phi_k \).

Select \( \tau_x \) as the task with the greater number of threads \( m_x \geq m_y \).

Consider the impact on \( \sum_{\tau_k \in \Phi_k} c_k(m_k) \) of moving one thread of \( \tau_y \) to \( \tau_x \), as the operation of adding the difference of WCET values for \( c_x(m_x + 1) \) and \( c_y(m_y - 1) \) to the sum.

\[
\left( \sum_{\tau_k \in \Phi_k} c_k(m_k) \right) - c_x(m_x + 1) - c_y(m_y - 1)
\]

By the concave growth Property 1 and virtue of \( \Phi_i \) to \( \Phi_k \), ie. \( \Phi_i \neq \Phi_k \), moving one thread from \( \tau_y \) to \( \tau_x \) will reduce the WCET sum. By repeatedly moving tasks to reduce the WCET sum, \( \Phi_k \) will satisfy all aspects of Property 4 of partial task sets created by \( \text{DIVIDE} \), ie. \( \Phi_i = \Phi_k \) after all moves have been completed. This contradicts the assumption that \( \Phi_i \neq \Phi_k \) and the relationship of their WCET sums, therefor \( \Phi_i \) is minimal.

\[\blacktriangleright \text{Theorem 5 (TPJ is Optimal with Respect to npm-feasibility).} \]

For a task set specification \( \hat{\tau} \), \( TPJ \) returns feasible if and only if there exists an npm-feasible posterior task set \( \tau \) of \( \hat{\tau} \).

\[\blacktriangleright \text{Proof.} \]

\[\text{Forward Direction} \ (TPJ \text{ returns feasible for } \hat{\tau} \implies \exists \text{ a posterior task set } \tau \mid \tau \text{ is npm-feasible).} \]

The \( TPJ \) algorithm returned a posterior task set \( \tau \) where the infeasibility condition (Equation 9) is never satisfied across intervals of length \( 0 \leq t \leq T^*(\tau) \) and every job of \( \tau \) executes non-preemptively for \( c_i(m_i) \leq q_i \) time units. Therefore, \( \tau \) is npm-feasible.

\[\text{Reverse Direction} \ (\exists \text{ a posterior task set } \tau \mid \tau \text{ is npm-feasible} \implies TPJ \text{ returns feasible for } \hat{\tau}). \]

For the purpose of demonstrating a contradiction, assume \( TPJ \) returns infeasible for an npm-feasible task set \( \hat{\tau} \). Name the absolute deadline which \( TPJ \) returned infeasibility for \( D_x \) from the set ordered deadlines \( \{D_1, D_2, \ldots\} \) and the task which generated \( D_x, \hat{x} \). Name the set of tasks with relative deadlines smaller than \( \hat{\tau} \).

For any task \( \tau_k \in \hat{\tau} \) and partial task set \( \Phi_k \) of \( \tau_k \) included in the posterior set \( \tau \), the number of tasks and threads assigned to each \( \Phi_k \) cannot be affected by \( \hat{\tau} \) due to \( d_x \) and Property 3. The combined set of posterior tasks of \( \hat{\tau} \) in \( \tau \) is denoted \( \hat{\tau} = \cup_{\tau_k \in \hat{\tau}} \Phi_k \).

There are two cases where \( TPJ \) will return infeasible for \( \hat{\tau} \), on Line 8 and Line 22. Both illustrate a contradiction with the respect to demand.
Line 8: If TPJ returns infeasible for on Line 8 there is insufficient slack \( q_x \) to execute any one-thread job of \( \hat{r} \), non-preemptively. Since slack is inversely related to demand, the demand of \( \hat{r} \) is too great to allow any thread of \( \tau_x \) as part of a feasible task set.

Line 22: If TPJ returns infeasible for \( \hat{r} \) on Line 22, there is insufficient supply for \( \Phi_x \) (the set of partial tasks of \( \hat{r} \)). By Corollary 2 and Theorem 4 the demand of \( \Phi_x \) is minimal over all intervals for the available slack \( q_x \). Due to Property 3 only tasks with shorter relative deadlines i.e. \( \hat{r} \), can impact the demand of \( \Phi_x \) by affecting \( q_x \). In this case, the demand of \( \hat{r} \) is too great for the demand of \( \Phi_x \) to be included as part of a feasible task set.

By assumption \( \hat{r} \) is nmp-feasible, the infeasibility conditions on Lines 8 and 22 of TPJ indicate the demand of \( \hat{r} \) is too great. However, TPJ adds each partial set \( \Phi_k \) to \( \hat{r} \) in increasing deadline order. By Property 3, every \( \Phi_k \) added to \( \hat{r} \) exclusively impacts the demand of larger deadlines. Every \( \Phi_k \) increases the demand of \( \hat{r} \) minimally starting with \( D_1 \), maximizing the slack available for partial task sets with greater deadlines; thus the demand of \( \hat{r} \) is minimal and cannot be reduced. For \( \hat{r} \) to be nmp-feasible, there must be another partial task set that reduces \( \hat{r} \)’s demand, which is a direct contradiction. Therefore, TPJ must return feasible.

5 Evaluation

Evaluation [28] of TPJ and the non-preemptive multi-threaded task model presented in this work focuses on the schedulability ratio of synthetic task sets and a case study based upon the evaluation of BUNDLEP [29]. The ratio of task set specifications deemed schedulable by TPJ for EDF-NP will be compared to NP-CHUNKS in both limited and fully preemptive settings for EDF. What follows is a description of the parameters to task set specification generation, the prescribed evaluation metrics, and analysis of the results.

5.1 Generating Task Sets

A specified task set \( \tau \) is generated with four parameters, \( M \) the total number of threads of execution, \( U \) the target utilization, a maximum growth factor \( \mathbb{F} \), and \( m \) the maximum number of threads per task. The number of threads \( M \) may be one of \( \{3, 5, 7, 10, 25, 50, 100\} \) with dependent \( m \) values of \( \{2, 2, 3, 4, 8, 16, 32\} \). Utilization varies from \( [0.1, 0.9] \) and the growth factor varies from \( [0.1, 0.9] \) independently by increments of 0.1.

Each task \( \tau_i \in \tau \) is assigned \( m_i \) threads from a random uniform integer distribution over \([1, m_i]\), such that the sum of all threads is equal to \( M = \sum_{\tau_i \in \tau} m_i \). A task’s period \( p_i \) is from a uniform integer distribution over \([10, 1000]\). Utilization \( u_i \) of each task \( \tau_i \) is calculated using the UUniFast\((n, U)\) [9] algorithm, where \( n = |\tau| \).

A task’s WCET is assigned for \( m_i \) threads, \( c_i(m_i) = [p_i \cdot U_i] \). Tasks are given a growth factor \( \mathbb{F}_i \) in a uniform real distribution over \([0.1, \mathbb{F}]\). The remaining \( m_i - 1 \) WCET values are determined by substituting \( \mathbb{F}_i \) into Equation 3. The relative deadline of \( \tau_i \), \( d_i \) is taken from a uniform integer distribution over \([\text{max}(c_i(m_i), p_i/2), 1000]\).

For each combination of \((M, m, U, \mathbb{F})\), 1000 task sets specifications are generated. Table 2 summarizes the parameters of task set generation. The smaller values of \( M \) are taken from \([7]\) and the dependent \( m \) values were selected to avoid one task consuming more than half of the threads in the task set specification (where possible).

| Table 2 Task Set Generation Parameters. |
| --- | --- |
| \( U \) | \([0.1, 0.9]\) |
| \( \mathbb{F} \) | \([0.1, 0.9]\) |
| \( M \) | \([3, 5, 7, 10, 25, 50, 100]\) |
| \( m \) | \([2, 2, 3, 4, 8, 16, 32]\) |
5.1.1 Applicability of Parameters

To avoid favoring TPJ, the task set generation parameters $m$ and $F$ were carefully selected. For the threads per task $m$, a large $m$ favors TPJ. Therefore, no single task may be assigned more than half the total threads: $m \leq \lfloor \frac{M}{2} \rfloor$ (except for $M = 3$).

The growth factor $F$ is informed by previous results for BUNDLEP [29]. In [29], multi-threaded tasks are constructed from the Mälardalen WCET benchmarks [16]. Task analysis in [29] yields growth factors below 0.1 for several benchmarks. A lower bound (0.1) on $F$ greater than observed values is pessimistic, resulting in less favorable results for TPJ.

5.2 Case Study

BUNDLEP’s evaluation covers 18 benchmarks for distinct architecture configurations. An architecture configuration includes the block reload time (BRT), cycles per instruction (CPI), and number of cache lines. One of the least favorable in terms of the analytical benefit of BUNDLEP is a BRT of 100, CPI of one, and 32 cache lines. From this configuration, the WCET values and growth factors were extracted, growth factors ranging in the range $[0.08, 3.02]$.

From these results of BUNDLEP 1000 task sets with 18 tasks (one per benchmark) and a total 100 threads were generated per utilization target. The utilization target ranged from 0.1 to 1.0 increments of 0.1. Threads were assigned to each task $\tau_i$ from a distribution over $m_i \in [2, 8]$. Each tasks utilization, period, and deadline, $c_i(m_i)$ were assigned using the same method as synthetic tasks. The WCET values for fewer threads $1 \leq k < m_i$, were scaled such that the value of $c_i(k)/c_i(m_i)$ remained constant after the $c_i(m_i) = \lceil p_i \cdot U_i \rceil$ assignment.

5.3 Evaluation Metrics

TPJ is compared with the NP-CHUNKS schedulability test in non-preemptive (EDF-NP) and preemptive (EDF-P) settings. The focus of the evaluation is on the non-preemptive setting. The preemptive setting serves as a comparison to alternative scheduling strategies and the theoretical best case. For EDF-P, preemptions incur no penalty, CRPD or otherwise. In this highly advantageous setting for EDF-P, TPJ can still produce feasible non-preemptive task sets NP-CHUNKS deems infeasible in a preemptive setting!

To compare schedulability tests, each task set specification $\hat{\tau}$ is provided to TPJ without modification under EDF-NP scheduling. TPJ will transform the task set producing a posterior task set $\tau$ if a feasible one exists. A task set specification $\hat{\tau}$ cannot be provided directly to NP-CHUNKS, since NP-CHUNKS has no concept of threads per job.

To be suitable for analysis by NP-CHUNKS, a task set specification $\hat{\tau}$ is transformed into two posterior task sets. The first task set, $\tau^1$ represents single-threaded tasks by including all threads of $\hat{\tau}$ as individual tasks. The second task set, $\tau^m$ represents the tasks of $\hat{\tau}$ as indivisible, executing all specified threads without preemption per job. Each task in $\tau^m$ benefits from the thread-level scheduler but does not expose the threaded nature of the task to the scheduling algorithm. This is achieved by modifying an anterior task $\hat{\tau}_j$ with $\hat{m}_j > 1$ and $\hat{c}_j(\hat{m}_j)$ to a posterior task $\tau_j$ with $m_j = 1$ and $c_j(1) = \hat{c}_j(\hat{m}_j)$.

The NP-CHUNKS schedulability test will produce results for $\tau^1$ and $\tau^m$ in both preemptive and non-preemptive settings. For non-preemptive schedulability analysis, each task $\tau_i \in \tau^1$ or $\tau^m$ must have a non-preemptive chunk size $q_i \geq c_i(m_i)$. When evaluating preemptive EDF schedulability for $\tau^1$ and $\tau^m$, the results are labeled EDF-P:1 and EDF-P:M respectively. When evaluating non-preemptive EDF schedulability, the results are labeled EDF-NP:1 and

---

1 Due to length restrictions the full listing of WECT and growth factors are omitted.
Table 3 Schedulability Test Combinations.

<table>
<thead>
<tr>
<th>Test</th>
<th>Task Set</th>
<th>EDF-NP</th>
<th>EDF-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPJ</td>
<td>(\hat{\tau})</td>
<td>EDF-TPJ</td>
<td>-</td>
</tr>
<tr>
<td>NP-CHUNKS</td>
<td>(\tau^1)</td>
<td>EDF-NP:1</td>
<td>EDF-P:1</td>
</tr>
<tr>
<td></td>
<td>(\tau^m)</td>
<td>EDF-NP:M</td>
<td>EDF-P:M</td>
</tr>
</tbody>
</table>

EDF-NP:M. Schedulability results for \(\hat{\tau}\) under EDF-NP scheduling are labeled EDF-TPJ. Table 3 gives a synopsis of the schedulability tests. Schedulability ratios for each of the combinations are calculated for every \((M, m, U, F)\) configuration.

It must be noted that EDF-P:M is an unrealistic schedulability test. It serves only as a theoretical limit to the benefits of concave growth. Concave growth is a result of scheduling threads of the same job without preemption by another job with a BUNDLE-based thread-level scheduler. However, current BUNDLE implementations require that an executing task cannot be preempted by a different task. Such a preemption would destroy the cache benefits and analysis of BUNDLE scheduling. Analysis of EDF-P:M assumes preemptions between jobs are allowed and have zero cost. It is included as a reference for \(\hat{\tau}\)’s performance, as a ceiling for what is theoretically possible given ideal (but likely impossible) conditions.

As a consequence of transforming multi-threaded task set specifications \(\hat{\tau}\) to single-threaded task sets \(\tau^1\), some single threaded task sets may not be feasible. One reason for a task set \(\tau^1\) to become infeasible is the utilization exceeding one, while \(\tau^m\) and \(\hat{\tau}\) have utilization less than one. In this setting, EDF-TPJ is capable of scheduling task sets that preemptive EDF cannot.

For a task set specification configuration \((M, m, U, F)\), call \(S\) the set of all task set specifications \(\hat{\tau}\) generated for the configuration. Call \(s\) the set of \(\tau^1\) task sets transformed from \(\hat{\tau} \in S\) such that \(\tau^1\) has utilization greater than one. The set \(s^{TPJ}\) is the subset of \(s\) deemed feasible by the TPJ schedulability test. That is, \(s^{TPJ}\) is the set of all tasks TPJ could schedule, yet EDF-P:1 could not (even) when CRPD values are zero.

5.4 Results

(a) BUNDLEP Case Study.  
(b) EDF-TPJ Summary.

Figure 2 Case Study and EDF-TPJ Summary Results.

Schedulability ratios from the BUNDLEP case study are given in Figure 2a. For the target architecture and 18 benchmarks, EDF-TPJ consistently outperforms the other non-preemptive algorithms. For preemptive EDF-P:1 (with zero cost preemptions), EDF-TPJ has higher schedulability ratios for the majority of target utilization values. EDF-TPJ’s
comparative performance increases with the target utilization. This case study demonstrates the benefit of TPJ to non-preemptive and (potentially) preemptive approaches.

Figures 2b, 3a, and 3b, summarize the results for the synthetic task sets varied by the utilization and growth factor. Within each graph, the schedulability ratios provided by EDF-P:1 and EDF-P:M serve as references. The difference between EDF-P:1 and the subject of the graph illustrate the benefit of preemptive scheduling. Inclusion of EDF-PM highlights the theoretical limit of concave growth to schedulability.

**Figure 3** EDF-NP:1 and EDF-NP:M Summary.

Including EDF-P:1 and EDF-P:M in each of the summary graphs eases the comparison between EDF-NP:1, EDF-NP:M, and EDF-TPJ. Comparing EDF-NP:1 (3a) to EDF-NP:M (3b), illustrates the benefits of the model and scheduling mechanism. EDF-NP:M has a consistently higher schedulability ratio for all utilizations and growth factors. EDF-TPJ (2b) outperforms EDF-NP:M, with higher schedulability ratios for all utilizations and growth factors due to the ability to transform task sets. EDF-TPJ performs best among the non-preemptive tests across all configurations. Additionally, EDF-TPJ is able to schedule task sets deemed infeasible for EDF-P:1.

Table 4 summarizes the infeasible utilization findings for the synthetic tasks. For moderate and larger values of $M$($\geq 25$), the number of infeasible by utilization task sets dominate the specifications. For 25, 50, and 100 total threads, the infeasible by utilization comprise 44, 59, and 74 percent of the task sets respectively, with EDF-TPJ finding 25, 34, and 45 percent feasible. This illustrates the large potential of the proposed model, in conjunction with concave growth WCET functions of thread-level schedulers (e.g. BUNDLE and BUNDLEP).

**Table 4** $U > 1$ Feasibility.

<table>
<thead>
<tr>
<th>$(M, m)$</th>
<th>$(3, 2)$</th>
<th>$(5, 2)$</th>
<th>$(7, 3)$</th>
<th>$(10, 4)$</th>
<th>$(25, 8)$</th>
<th>$(50, 16)$</th>
<th>$(100, 32)$</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>S</td>
<td>$</td>
<td>81000</td>
<td>81000</td>
<td>81000</td>
<td>81000</td>
<td>81000</td>
<td>81000</td>
</tr>
<tr>
<td>$</td>
<td>s</td>
<td>$</td>
<td>3131</td>
<td>4973</td>
<td>11744</td>
<td>18689</td>
<td>36565</td>
<td>49147</td>
</tr>
<tr>
<td>$</td>
<td>s^{TPJ}</td>
<td>$</td>
<td>465</td>
<td>291</td>
<td>1437</td>
<td>3065</td>
<td>9426</td>
<td>16912</td>
</tr>
</tbody>
</table>

There are two noteworthy trends within the schedulability results. The simpler of the two is the relationship between utilization and schedulability ratio for a fixed growth factor. Figure 4a illustrates the trend common among $M \leq 10$ total threads. The trend for preemptive and non-preemptive schedulability tests when utilization increases is for the schedulability ratio to decrease. However, EDF-TPJ always outperforms the other non-preemptive tests.
The second trend is slightly more complex. Figure 4b was selected for the smallest $M$ and $U$ values with visually distinct plots per schedulability test. The growth factor and the schedulability ratio are correlated. As the growth factor increases, so does the schedulability ratio. This is due to the utilization being held constant. When the growth factor is small, the WCET of the first thread of each task is larger. Larger WCET values are harder to schedule non-preemptively.

As $M$ increases beyond 10 total threads, the number of infeasible by utilization task sets $s$ grows. This contributes to the schedulability ratio of EDF-TPJ surpassing EDF-P:1 for threshold utilization and growth factor values. For $M = 25$, the threshold of utilization is between $[0.6, 0.7]$ shown in Figure 5.
For $M = 100$ and $F \leq 0.4$, EDF-TPJ outperforms EDF-P:1. Figure 6 highlights the advantage of EDF-TPJ compared to EDF-P:1 by virtue of concave growth. It also highlights the benefit of dividing tasks, as the performance of EDF-NP:M is always below EDF-TPJ.

The comparative performance of EDF-TPJ is at its lowest for $M < 10$ threads and $U > 4$ utilization. In these ranges EDF-TPJ maintains the highest schedulability ratio among the non-preemptive methods, but the ratio is closer to EDF-NP:M or EDF-NP:1 than EDF-P:1. This suggests, the decrease in EDF-TPJ’s performance is more likely due to the non-preemptive setting combined with larger WCET values for individual threads.

6 Conclusion

Motivation for this work stemmed from BUNDLE-based thread-level schedulers limitation of a single task and single job. The primary goal was to create a multi-task scheduling technique and schedulability test for those BUNDLE-based thread-level schedulers which leverages without decreasing the inter-thread cache benefit.

In addition to achieving the primary goal, the scheduling technique and schedulability test developed for the multi-task BUNDLE-based scheduler can be applied to any thread level scheduler with strictly increasing discrete concave WCET functions. This allows any compatible thread-level scheduling technique to benefit from the TPJ approach developed in this work. As a non-preemptive multi-threaded schedulability test TPJ is optimal with respect to npm-feasibility, always producing a feasible task set if one is schedulable by EDF-NP.

For future work, the primary focus is upon a fully or limited preemption scheduling algorithm that permits the inter-thread cache benefit of BUNDLE-based schedulers and other schedulers characterized by concave growth to retain their thread-level scheduling benefits.

References

15:22  NPM-BUNDLE: Non-Preemptive Multitask BUNDLE


