Parallel Scheduling for Cyber-Physical Systems: Analysis and Case Study on a Self-Driving Car*

Junsung Kim, Hyoseung Kim, †Karthik Lakshmanan, Ragunathan (Raj) Rajkumar
Real-Time and Multimedia Systems Laboratory, Carnegie Mellon University, Pittsburgh, PA
†Google Inc., Mountain View, CA
{junsungk, hyoseunk, raj}@ece.cmu.edu, †karthiksingaram@gmail.com

ABSTRACT

As the complexity of software for Cyber-Physical Systems (CPS) rapidly increases, multi-core processors and parallel programming models such as `OpenMP` become appealing to CPS developers for guaranteeing timeliness. Hence, a *parallel* task on multi-core processors is expected to become a vital component in CPS such as a self-driving car, where tasks must be scheduled in *real-time*.

In this paper, we extend the fork-join parallel task model to be scheduled in real-time, where the number of parallel threads can vary depending on the physical attributes of the system. To efficiently schedule the proposed task model, we develop the task *stretch** transform. Using this transform for global Deadline Monotonic scheduling for fork-join real-time tasks, we achieve a resource augmentation bound of 3.73. In other words, any task set that is feasible on m unit-speed processors can be scheduled by the proposed algorithm on m processors that are 3.73 times faster. The proposed scheme is implemented on Linux/RK as a proof of concept, and ported to Boss, the self-driving vehicle that won the 2007 DARPA Urban Challenge. We evaluate our scheme on Boss by showing its driving quality, i.e., curvature and velocity profiles of the vehicle.

Categories and Subject Descriptors

D.4 [**Software**]: Operating Systems; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; F.1 [**Theory of Computation**]: Computation by Abstract Devices; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*; I.2.9 [**Artificial Intelligence**]: Robotics—*Autonomous vehicles*

*This work was supported in part by the National Science Foundation and in part by the US Department of Transportation.

†This author contributed to this work while he was at Carnegie Mellon University.

ICCPS'13 April 8-11, 2013, Philadelphia, PA, USA.

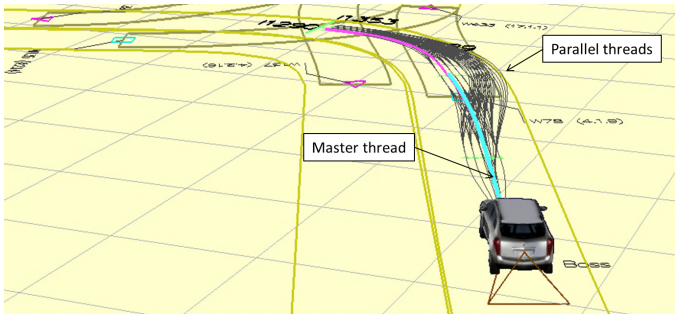

Figure 1: A motion planning algorithm for autonomous driving

1. INTRODUCTION

With cyber-physical systems (CPS), such as medical devices, aerospace systems, smart grids, nuclear power plants, robots and transportation vehicles, becoming more popular, demands for new functionality features multiply [28]. For example, active safety options such as adaptive cruise control, brake assist, collision avoidance, lane departure warning, sign detection and traction control are not rare anymore in recently built vehicles. We, in fact, expect these CPS functionalities to be readily available even in mid-range cars. With this trend, embedded real-time systems are indispensable in order to sense the physical environment, process data in real-time, control the actuators in a desirable manner and monitor the timing of the whole execution chain for ensuring safety.

Autonomous driving [31, 10, 26, 16, 30] is an appealing emerging CPS technology. In an autonomous car, motion planning, sensor fusion, computer vision and other artificial intelligence algorithms must run in real-time; however, the CPU-hogging nature of those algorithms poses challenges in guaranteeing their timeliness.

The timing challenge can be addressed by the fact that most algorithms for autonomous driving are *parallelizable*. A planning algorithm of a self-driving car can profit from parallelized tasks composed of numerous *threads*. The motion

planning algorithm calculates the best path for the vehicle to follow among a myriad of potential paths. This algorithm can be expedited by parallelizing the cost calculation for each path. The more paths the algorithm goes through, the better driving quality will be. Figure 1 is a screenshot of the operator interface for Boss, which won the 2007 DARPA Urban Challenge [31] showing a motion planning algorithm in operation. In the figure, the multiple lines coming out from Boss represent possible paths which Boss may follow, where each line is generated by a parallel thread of the motion planning algorithm. When all threads are completed, they merge into a *master thread* that selects the best path. It should be noted that the number of threads can vary depending on the physical conditions such as the shape of the road, the number of detected obstacles and the speed of the vehicle.

A perception subsystem of a self-driving car can also benefit from parallel tasks. In order for the vehicle to understand its surroundings, the perception subsystem should be able to process massive amounts of data from various types of sensors. Boss, for example, manages 36000 independent segments from its Velodyne HDL-64 LIDAR before fusing them with other sensor data. Then, the vehicle can classify and track the detected obstacles, whose number has a major impact on how many parallel threads are spawned by the perception subsystem.

The automotive industry has already started moving towards the multi-core processors for higher performance [22, 14]. AUTOSAR, a widely used automotive software infrastructure, supports multi-core processors [4]. In addition, parallel programming models like `OpenMP` [1] utilize multiple processing cores to guarantee concurrent execution. We believe that other CPS application domains will follow this trend sooner rather than later.

There has been relatively little research on tackling challenges in scheduling parallel real-time tasks. In [21], Lakshmanan et al. proposed a parallel task model and a partitioned fixed-priority scheduling algorithm on a multi-core processor, but the number of threads could not exceed the number of given processing cores. In [29], Saifullah et al. proposed a more generalized parallel real-time task model which allows different fork-join segments of a task to have a different number of threads.

Contributions: In this paper, we extend the fork-join real-time task model proposed in [21] so that an arbitrary number of threads can be scheduled, where the number of threads can vary depending on the physical attributes of the system. To efficiently schedule the proposed task model, we also propose a task *stretch** transform to schedule the task model on a given number of processing cores. Then, we prove that a resource augmentation bound of 3.73 is achieved when we use the task *stretch** transform for global Deadline Monotoic (DM) scheduling for fork-join real-time tasks. The proposed scheme is implemented on Linux/RK [25] and ported to the self-driving car Boss [31]. We evaluate our proposed scheme on Boss by showing its driving quality in terms of curvature and velocity profiles of the vehicle with an enhanced motion planning algorithm [17].

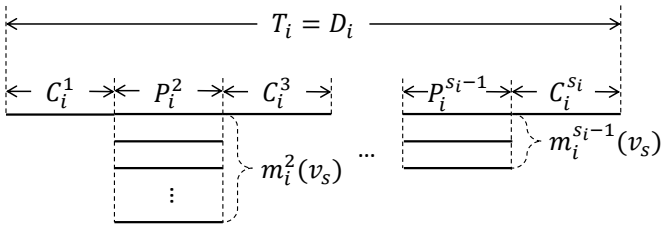

Figure 2: A fork-join real-time task model

Organization: The rest of this paper is organized as follows. In Section 2, we define our fork-join real-time task model and describe the system assumptions. We provide a scheduling algorithm to handle parallel real-time tasks in Section 3. The analysis using resource augmentation bound for global DM scheduling follows in Section 4. We, then, briefly explain in Section 5 the modifications made to Linux/RK to support the proposed scheme on a Linux-based system. Section 6 shows the curvature and velocity profiles of a self-driving car when our proposed scheme is used. We describe previous research relevant to our work in Section 7, and, in Section 8, we summarize our paper and discuss future work.

2. SYSTEM MODEL AND ASSUMPTIONS

Definition: We consider a set of tasks $\boldsymbol{\tau}$ composed of n multi-threaded real-time tasks, and the given set $\boldsymbol{\tau}$ runs on a system with m processing cores. $\boldsymbol{\tau}$ is represented as $\boldsymbol{\tau}$: $\{\tau_1, \tau_2, \ldots, \tau_n\}$, and the tasks in $\boldsymbol{\tau}$ are sorted in non-decreasing order of task periods (deadlines). Each task τ_i begins with a single thread spawning parallel threads, which join with another sequential thread of τ_i. τ_i interchanges this pattern between parallel and sequential segments. The number of parallel threads depends on the physical attributes of the given system $v_s \in \mathbb{R}^p$, where p is the number of dimensions that capture aspects of the operating environment. Then, as depicted in Figure 2, each task τ_i is represented as τ_i: $((C_i^1, (P_i^2, m_i^2(v_s)), C_i^3, \ldots, (P_i^{s_i-1}, m_i^{s_i-1}(v_s)), C_i^{s_i}), T_i, D_i)$, where

- s_i is the number of computation segments of τ_i. Since τ_i starts with a sequential segment and ends with a sequential segment while having parallel segments in the middle, s_i is a positive odd integer. For $1 \leq j \leq s_i$, the j^{th} element is a parallel segment if j is an even number. Similarly, the j^{th} element is a sequential segment if j is an odd number.
- $m_i^j(v_s)$ is the number of parallel threads for the j^{th} segment when $1 \leq j \leq s_i$. When j is an odd integer, $m_i^j(v_s)$ is 1 and omitted from the representation of τ_i above for ease of presentation. When j is an even integer, $m_i^j(v_s)$ is equal to or greater than 1 and represents the number of parallel threads spawned by the previous segment.
- C_i^j is the worst-case execution time of the j^{th} segment in task τ_i on a unit-speed processor when the j^{th} element is a sequential segment. Also, let $\tau_i^{j,1}$ denote the j^{th} sequential segment of τ_i.

- P_i^j is the worst-case execution time of each thread run in the j^{th} segment of task τ_i on a unit-speed processor when the j^{th} element is a parallel segment. For parallel segments of τ_i, each thread of parallel threads is represented as $\tau_i^{j,k}$, where k varies from 1 to $m_i^j(v_s)$.
- D_i is the relative deadline to its release time.
- T_i is the period of τ_i. An implicit deadline is assumed, i.e., $T_i = D_i$.

Application Examples to Autonomous Driving: The motion planning algorithm of Boss uses `OpenMP` to parallelize its cost calculations to find the best path. Since the algorithm takes its inputs: the road rules, the road shape, the vehicle speed, the list of static obstacles and the list of dynamic obstacles, we define v_s as $< RoadRule, RoadShape, VehicleInfo, StaticObstacles, DynamicObstacles >$. This vector v_s is then used to decide the number of parallel threads accordingly. The perception algorithm of Boss leverages `pthread` to expedite its executions of processing perceived objects. We therefore define v_s for the perception algorithm as $<SensorList, SensorPose, RawSensorDataList, VehiclePose>$. In this paper, we consider the number of threads within each parallel segment not to exceed the maximum value of $m_i^j(v_s)$ for $\forall v_s \in \mathbb{R}^p$. For ease of presentation, therefore, we use m_i^j instead of $m_i^j(v_s)$.

Assumptions: Each task τ_i is assumed to generate an infinite series of independent jobs. The release time of the j^{th} segment of each job of τ_i should be after the completion time of the $(j-1)^{th}$ segment[1]. Therefore, if the j^{th} element of τ_i is a sequential segment, all parallel threads of $(j-1)^{th}$ segment of τ_i should complete before the j^{th} element of τ_i starts. We assume that all jobs are preemptable with negligible cost. We also assume that there is negligible migration cost when a job is migrated from a core to another.

Terminology: Using this model, we define the *maximum number of threads* of τ_i, which is the maximum value among m_i^j of τ_i. Formally,

$$m_i = \max_{j=1}^{s_i} m_i^j$$

The *maximum execution length* of a task τ_i on a unit-speed processor is defined as:

$$C_i = \sum_{j=0}^{\frac{s_i-1}{2}} C_i^{2j+1} + \sum_{j=1}^{\frac{s_i-1}{2}} m_i^{2j} P_i^{2j}$$

where, C_i represents the response time on a unit-speed single core processor when run alone. The first term corresponds to sequential task segments and the second term corresponds to fork-join segments.

To define the *minimum execution length* of a task τ_i, we have to consider two different cases: (1) $m_i \leq m$ and (2) $m_i > m$. For the first case, the minimum execution length is defined as $\eta_i = \sum_{j=0}^{\frac{s_i-1}{2}} C_i^{2j+1} + \sum_{j=1}^{\frac{s_i-1}{2}} P_i^{2j}$, where η_i is the response time when each single thread of τ_i can use a core exclusively. When $m_i > m$, the definition above does not hold good because some threads must be serialized. When $m_i > m$, therefore, we define the minimum execution length η_i as:

$$\eta_i = \sum_{j=0}^{\frac{s_i-1}{2}} C_i^{2j+1} + \sum_{j=1}^{\frac{s_i-1}{2}} \left\lceil \frac{m_i^{2j}}{m} \right\rceil P_i^{2j} \quad (1)$$

The definition above can also be used when $m_i \leq m$ because $\lceil \frac{m_i^j}{m} \rceil = 1$ when $m_i \leq m$. Hence, it holds good for both cases. For ease of presentation, we also let $P_i = \sum_{j=1}^{\frac{s_i-1}{2}} \left\lceil \frac{m_i^{2j}}{m} \right\rceil P_i^{2j}$, which is the execution requirement of the parallel segments contributing to η_i.

The task model in this paper is extended from the fork-join task model[2] proposed in [21]. The two main differences between the previous one and this model are that (1) our model places no limitation on the number of threads, and (2) our model allows different number of threads per parallel segment. Hence, this model is more practical.

3. SCHEDULING FORK-JOIN REAL-TIME TASKS

It was shown in [21] that there are unschedulable task sets where the total utilization of the taskset is slightly greater and very close to 1 even though there are m processing cores. In other words, deadlines can be missed even though only $\frac{1}{m}$ of available cycles is used. Although m approaches infinity, the schedulability does not change [21]. This worst-case behavior continues to hold good for the proposed model in this paper because it is an extended form of the task model proposed in [21]. In this section, we first consider a scheduling method to handle fork-join real-time tasks on a processor with a given number of cores. Then, we propose the task *stretch** transform to deal with our enhanced task model.

3.1 Running Fork-Join Real-Time Tasks on m CPU Cores

Consider a task $\tau_i \in \boldsymbol{\tau}$ running on m processing cores. If the maximum number m_i of parallel threads among all parallel segments in τ_i is less than the number of processing cores m, we can directly apply the task transformation algorithm described in [21]. If m_i exceeds the number of processing cores m, then the serialization of some parallel threads must happen as depicted in Figure 3, where a task meets its deadline on a quad-core processor, but not on a dual-core processor.

PROPOSITION 1. *A fork-join real-time task τ_i requires at least the minimum execution length η_i units of time on m CPU cores to meet its deadline.*

We obtain the minimum execution length η_i of τ_i depicted in Figure 3 as 10 on a quad-core processor and 16 on a dual-core processor from Equation 1. From Proposition 1, we can show that the given task is infeasible on a dual-core processor because η_i on a dual-core processor is greater than its deadline.

[1]We will use the terms 'jobs' and 'tasks' interchangeably where the distinction is not of importance.

[2]We also call our proposed model a fork-join task model unless stated otherwise.

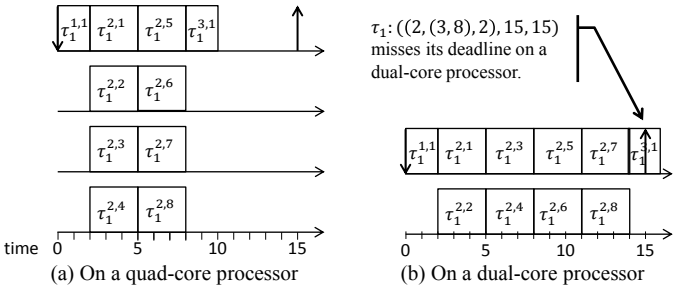

Figure 3: $\tau_1 : ((2, (3, 8), 2), 15, 15)$ **misses its deadline on a dual-core processor, but not on a quad-core processor.**

3.2 The Task Stretch* Transform

We propose a task transformation algorithm *stretch** in Algorithm 1. It breaks down a fork-join real-time task into a set of tasks. This set is composed of a long task called a *master string* and a bunch of constrained-deadline tasks with $D < T$. This set can be scheduled using any scheduling algorithm supporting conventional single-threaded tasks such as global DM, global EDF [12] and FBB-FFD [13].

In Algorithm 1, when a new constrained-deadline task is created, it is represented as $\tau : (C, D, \phi)$, where C is the worst-case execution time, D is the relative deadline, and ϕ is the release offset. When a parallel thread is merged into an existing task, we use $\oplus$ as a symbol and $\tau : (C)$ as the thread added to the existing task. Merging a thread does not change either the deadline or the offset of the existing task. In this algorithm, we made a small change on how to use the modulo function. $k \mod q_i$ returns q_i if $k \mod q_i = 0$.

We use two parameters f_i and q_i in Algorithm 1. f_i is the ratio of the parallel execution requirements P_i to the slack of the task $T_i - \eta_i$. We use this value to evenly distribute the slack to each parallel segment. q_i is the number of parallel threads after a task is processed by Algorithm 1. In other words, at any point of time t, $\tau_i^{stretch^*}$ will have at most q_i concurrent running threads on m cores. It should be noted that the deadline assignment for the q_i^{th} thread is different from others because we split the thread so that we can avoid the worst-case scenario explained in [21].

The algorithm is an extension of the task *stretch* transformation proposed in [21]. The *stretch** transformation can handle more general cases: (1) when the number of parallel threads exceeds the number of cores, and (2) when the number of parallel threads of each segment is different. The improvements can be described as follows:

- If the number of parallel threads within a fork-join segment exceeds the number of CPU cores m, all parallel threads $\tau_i^{2j,k}$ with the same value of $(k \mod q_i)$, where $1 \le k \le m_i^{2j}$, coalesce into the thread $\tau_i^{2j,k \mod q_i}$. This step guarantees that the number of parallel threads does not exceed the number of processing cores after the task transformation.
- Based on the new worst-case execution time of the merged threads of each parallel segment, a constrained deadline proportional to $(1+f_i)$ is assigned to each parallel segment by the algorithm. Accordingly, an offset is also determined so that parallel threads are released at the right time instants.

Algorithm 1 Stretch*(τ)

Input: τ: a fork-join real-time task
Output: $\tau^{stretch^*}$: a *stretch**ed task set

$\tau_i^{master} \leftarrow ()$
$\{\tau_i^{cd}\} \leftarrow \{\}$
if $C_i \le T_i$ **then**
  ▷ *The task can run on a single core*
  for $j = 1$ to $\frac{s_i - 1}{2}$ **do**
    $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2j-1,1} : (C_i^{2j-1})$
    for $k = 1$ to m_i^{2j} **do**
      $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2j,k} : (P_i^{2j})$
  $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{s_i,1} : (C_i^{s_i})$
else
  ▷ *Stretch* the task to its deadline*
  $f_i \leftarrow \frac{T_i - \eta_i}{\sum_{j=1}^{\frac{s_i-1}{2}} \lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}}$
  $q_i \leftarrow \min(m, m_i) - \lfloor f_i \rfloor$
  for $j = 1$ to $\frac{s_i - 1}{2}$ **do**
    $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2j-1,1} : (C_i^{2j-1})$
    ▷ *1) Coalesce threads so that the total number of parallel threads is less than q_i*
    for $k = 1$ to m_i^{2j} **do**
      if $k \mod q_i = 1$ **then**
        ▷ *Part of the master string*
        $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{2j,k} : (P_i^{2j})$
      else if $\tau_i^{2j,k \mod q_i} \notin \{\tau_i^{cd}\}$ **then**
        ▷ *Create a new parallel thread*
        $D_i^{2j} \leftarrow (1 + f_i)\lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}$
        $\phi_i^{2j} \leftarrow \sum_{l=0}^{j-1} C_i^{2l+1} + \sum_{l=1}^{j-1} D_i^{2l}$
        $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i^{2j,k \mod q_i} : (P_i^{2j}, D_i^{2j}, \phi_i^{2j})$
      else if $\tau_i^{2j,k \mod q_i} \in \{\tau_i^{cd}\}$ **then**
        ▷ *Part of the existing threads*
        $\tau_i^{2j,k \mod q_i} \leftarrow \tau_i^{2j,k \mod q_i} \oplus \tau_i^{2j,k} : (P_i^{2j})$
    ▷ *2) Split among the q_i-th thread and the master string*
    if $\tau_i^{2j,q_i} \in \{\tau_i^{cd}\}$ **then**
      $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} - \tau_i^{2j,q_i}$
      $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i'^{2j,q_i} : ((f_i - \lfloor f_i \rfloor)\lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j})$
      ▷ *Create a new parallel thread*
      $D_i^{2j,q_i} \leftarrow (1 + \lfloor f_i \rfloor)\lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}$
      $\phi_i^{2j} \leftarrow \sum_{l=0}^{j-1} C_i^{2l+1} + \sum_{l=1}^{j-1} D_i^{2l}$
      $\{\tau_i^{cd}\} \leftarrow \{\tau_i^{cd}\} \cup \tau_i''^{2j,k} : ((1 + \lfloor f_i \rfloor - f_i)\lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}, D_i^{2j,q_i}, \phi_i^{2j})$
  $\tau_i^{master} \leftarrow \tau_i^{master} \oplus \tau_i^{s_i,1} : (C_i^{s_i})$
return $\tau_i^{stretch^*} := (\tau_i^{master}, \{\tau_i^{cd}\})$

Figure 4 shows an example of the task *stretch** transformation with a task τ_1: $((2, (3, 8), 2), 15, 15)$. The task has 8 parallel threads, and it has a slack of 5 because the minimum execution length η_1 is 10. Using the slack, a portion of $\tau_1^{2,4}$ and $\tau_1^{2,8}$ are scheduled with the master string.

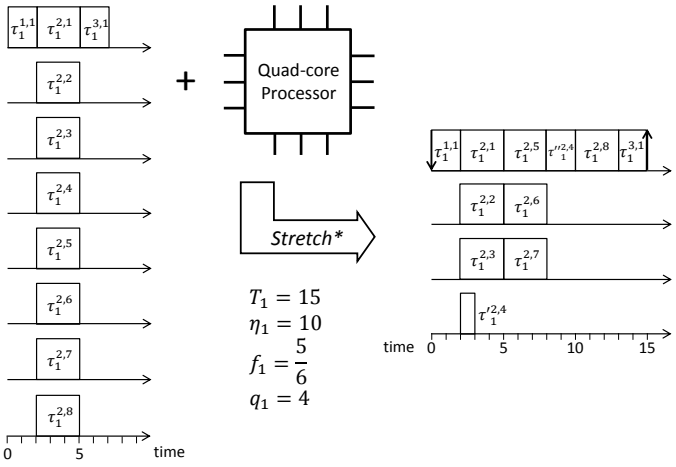

Figure 4: The task *stretch transformation example with** $\tau_1 : ((2,(3,8),2),15,15)$

4. RESOURCE AUGMENTATION BOUND ANALYSIS FOR GLOBAL DEADLINE MONOTONIC SCHEDULING

In this section, we derive the resource augmentation bound of global DM scheduling for the task model described in Section 3. To the best of our knowledge, this is the first result of resource augmentation bound of global DM scheduling for parallel real-time tasks. For this approach, we use a density-based schedulability test proposed in [8] given below.

THEOREM 1 (FROM [8]). *A set of periodic or sporadic tasks with constrained deadlines is schedulable with Deadline-Monotonic priority assignment on $m \geq 2$ processors if:*

$$\lambda_{sum} \leq \frac{m}{2}(1 - \lambda_{max}) + \lambda_{max} \quad (2)$$

where, λ_{sum} is the sum of the density of each task in the taskset, λ_{max} is the maximum value of task densities, and a density λ is a ratio of the deadline of a task to its worst-case execution time.

Let $\lambda_i^{stretch^*}$ denote the sum of the density of each task in the stretched taskset $\tau_i^{stretch^*}$. As specified in Algorithm 1, two cases, (1) $C_i \leq T_i$ and (2) $C_i > T_i$ should be considered to understand the properties of $\lambda_i^{stretch^*}$. Two corresponding lemmas are presented next.

LEMMA 1. *For a fork-join real-time task τ_i, the density of the resulting stretched task $\tau_i^{stretch^*}$ is bounded by $\frac{C_i}{T_i}$ if $C_i \leq T_i$.*

PROOF. For the case of $C_i \leq T_i$, we use the fact that the execution requirement and $T_i(= D_i)$ of both τ_i and $\tau_i^{stretch^*}$ are equal. Then, from the definition of density, $\frac{C_i}{T_i}$. □

Before investigating a fork-join real-time task τ_i with $C_i > T_i$, we assume that τ_i is provided with a level of parallelism so that $\frac{C_i}{min(m,m_i)} \geq P_i$ is satisfied. In the ideal case, based on Amdahl's law [2], $\frac{C_i}{min(m,m_i)} = P_i$ holds good because all the segments are running in parallel. Since we assume a fork-join model that has non-zero serial segments, the ideal case cannot be achieved. However, approaching P_i to $\frac{C_i}{min(m,m_i)}$ is desirable to fully utilize parallelism.

LEMMA 2. *For a fork-join real-time task τ_i, the sum of the density of the resulting stretched $\tau_i^{stretch^*}$ is bounded by $\frac{C_i}{T_i - \eta_i}$ if $C_i > T_i$.*

PROOF. For the case of $C_i > T_i$, it should be noted that the output of the algorithm is a set of tasks composed of a master thread τ_i^{master} and several constrained deadline tasks $\{\tau_i^{cd}\}$. Hence, the following inequality holds good:

$$\lambda_i^{stretch^*} \leq \lambda_i^{master} + \sum_{\tau_i \in \{\tau_i^{cd}\}} \lambda_i$$

Since the worst-case execution time of τ_i^{master} is less than T_i, $\lambda_i^{master} \leq 1$ from the implicit deadline assumption. It is known that there will be at most q_i concurrent running threads including the master thread at any point of time t. We ensure this by assigning an offset whenever a new parallel thread is created in Algorithm 1. The offset also guarantees that only one segment is active at a time. Thus, the density of τ_i can be substituted with the density of a segment that has the largest value among the densities of the segments of τ_i.

Let $P_i^{max} = \max_{j=1}^{\frac{s_i-1}{2}} \lceil \frac{m_i^{2j}}{m} \rceil P_i^{2j}$. We first consider the case of $q_i > 2$. When q_i threads are simultaneously running, for the $q_i - 1$ constrained tasks, there will be $q_i - 2$ parallel threads with the execution time of P_i^{max} and the relative deadline of $(1 + f_i)P_i^{max}$. There will also be a parallel thread with the execution time of $(1 + \lfloor f_i \rfloor - f_i)P_i^{max}$ and the relative deadline of $(1 + \lfloor f_i \rfloor)P_i^{max}$. Therefore, if we let $P_i = \sum_{j=1}^{\frac{s_i-1}{2}} \left\lceil \frac{m_i^{2j}}{m} \right\rceil P_i^{2j}$, the following inequalities are satisfied:

$$\sum_{\tau_i \in \{\tau_i^{cd}\}} \lambda_i \leq \frac{(q_i - 2)P_i^{max}}{(1 + f_i)P_i^{max}} + \frac{(1 + \lfloor f_i \rfloor - f_i)P_i^{max}}{(1 + \lfloor f_i \rfloor)P_i^{max}}$$
$$\leq \frac{(q_i - 1)}{(1 + f_i)} = \frac{(q_i - 1)P_i}{(P_i + T_i - \eta_i)}$$

We then consider the case of $0 < q_i \leq 2$. When q_i is 1, it means that τ_i can run on a single core. Therefore, we focus on the case of $q_i = 2$, which means that $\sum_{\tau_i \in \{\tau_i^{cd}\}} \lambda_i$ will have only the task which is split. Therefore,

$$\sum_{\tau_i \in \{\tau_i^{cd}\}} \lambda_i \leq \frac{(1 + \lfloor f_i \rfloor - f_i)P_i^{max}}{(1 + \lfloor f_i \rfloor)P_i^{max}} \leq \frac{1}{(1 + f_i)}$$
$$= \frac{P_i}{(P_i + T_i - \eta_i)} = \frac{(q_i - 1)P_i}{(P_i + T_i - \eta_i)}$$

Now, we consider both τ_i^{master} and $\{\tau_i^{cd}\}$.

$$\lambda_i^{stretch^*} \leq 1 + \frac{(q_i-1)P_i}{(P_i+T_i-\eta_i)} = \frac{P_i+T_i-\eta_i+(q_i-1)P_i}{(P_i+T_i-\eta_i)}$$
$$= \frac{(f_i+q_i)P_i}{(P_i+T_i-\eta_i)} = \frac{(f_i+min(m,m_i)-\lfloor f_i \rfloor)P_i}{(P_i+T_i-\eta_i)}$$
$$\leq \frac{min(m,m_i)P_i}{(P_i+T_i-\eta_i)} \leq \frac{C_i}{T_i-\eta_i}$$

From the inequality above, the lemma is proved. □

We define a task called a *heavy task* that has a density greater than or equal to $\frac{1}{\nu}$ on a ν-speed processing core.

THEOREM 2. *Global Deadline Monotonic scheduling of the fork-join real-time task model has a resource augmentation bound of 3.73 when each heavy task is assigned to its own processing core.*

PROOF. Consider a set of n fork-join real-time tasks $\boldsymbol{\tau}$. We assume that the given taskset is feasible on m identical unit-speed processors, which implies $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m$. Otherwise, the given taskset is not feasible.

Let there be k heavy tasks on a ν-speed processor. Under the task *stretch** transform described in Algorithm 1, these are either *fully stretched* tasks ($C_i \leq T_i$) or master threads ($C_i > T_i$). Both types of tasks have a deadline equal to their period, and their density is at least 1 on a unit-speed processor by the definition of a heavy task.

Therefore, for the remaining n' tasks:

$$\sum_{i=1}^{n'} \frac{C_i}{T_i} = \sum_{i=1}^{n'} \frac{C_i}{D_i} = \sum_{i=1}^{n'} \lambda_i = \lambda_{sum} \leq (m-k) \tag{3}$$

We need to show that these remaining tasks are schedulable on $m'(=m-k)$ processors of speed ν, where $\nu \geq 3.73$.

On a processor that is ν times faster, the minimum execution length η_i^{ν} on a ν-speed processor is given by

$$\eta_i^{\nu} = \sum_{j=0}^{\frac{s_i-1}{2}} \frac{C_i^{2j+1}}{\nu} + \sum_{j=1}^{\frac{s_i-1}{2}} \left\lceil \frac{m_i^{2j}}{m} \right\rceil \frac{P_i^{2j}}{\nu} \leq \frac{\eta_i}{\nu} \leq \frac{T_i}{\nu} \tag{4}$$

where, $\forall 1 \leq i \leq n$. Also, the maximum execution length of τ_i on a ν-speed processor is

$$C_i^{\nu} = \sum_{j=0}^{\frac{s_i-1}{2}} \frac{C_i^{2j+1}}{\nu} + \sum_{j=1}^{\frac{s_i-1}{2}} m_i^{2j} \frac{P_i^{2j}}{\nu} = \frac{C_i}{\nu} \tag{5}$$

where, $\forall 1 \leq i \leq n$.

Case (1): For each fully stretched task τ_i that is non-heavy on ν-speed processors, the density is $\frac{\frac{C_i}{\nu}}{T_i} \leq \frac{1}{\nu}\frac{C_i}{T_i} \leq \frac{1}{\nu-1}\frac{C_i}{T_i}$ from Lemma 1 and Equation 5.

Case (2): Consider the constrained-deadline taskset generated by *stretch** on ν-speed processors for task τ_i. From the perspective of load, the total density on ν-speed processors is bounded by $\frac{C_i^{\nu}}{T_i-\eta_i^{\nu}} \leq \frac{C_i/\nu}{T_i-\frac{T_i}{\nu}} = \frac{1}{\nu-1}\frac{C_i}{T_i}$ from Lemma 2, Inequality 4 and Equation 5.

λ_{sum} on ν-speed processors, therefore, is bounded by $\frac{m'}{\nu-1}$ because $\lambda_{sum} \leq \sum_{i=1}^{n'} \frac{1}{\nu-1}\frac{C_i}{T_i} = \frac{1}{\nu-1}\sum_{i=1}^{n'} \frac{C_i}{T_i} \leq \frac{m'}{\nu-1}$ from Inequality 3. The master threads for tasks that cannot be fully stretched are always heavy tasks since they use up the entire T_i on the ν-speed processor. By the definition of heavy tasks, λ_{max} is always upper bounded by $\frac{1}{\nu}$ on ν-speed processors. Then, for $m' \geq 2$ using Inequalities 2 and 3 and the cases considered above,

$$\frac{m'}{2}\left(1-\frac{1}{\nu}\right) + \frac{1}{\nu} \geq \frac{m'}{\nu-1}$$
$$\Leftrightarrow \frac{m'}{2} - 1 \leq \nu\left(\frac{m'}{2} - \frac{m'}{\nu-1}\right)$$
$$\Leftrightarrow m' \frac{4\nu-\nu^2-1}{2\nu(\nu-1)} \leq \frac{1}{\nu}$$
$$\Leftrightarrow \frac{4\nu-\nu^2-1}{2(\nu-1)} \leq \frac{1}{m'}$$

As $m' \to \infty$, we get,

$$\frac{4\nu-\nu^2-1}{2(\nu-1)} \leq \frac{1}{m'} \Leftarrow \nu \geq 2+\sqrt{3}$$

This holds good for all $m' \geq 2$ processors using $\nu \geq 2+\sqrt{3} \approx 3.73$. □

5. GLOBAL SCHEDULING ON LINUX/RK

We have designed an operating system abstraction for managing our parallel real-time task model using the resource-reservation paradigm. A parallel task in our model is composed of multiple threads. A thread called *master string* executes all sequential segments and a portion[3] of parallel segments. Parallel threads are spawned by the master thread and execute the remaining portion of parallel segments. In order to represent the multiple threads and their precedence constraints, our abstraction employs the resource management entities, *resource set* and *reserve*, introduced in resource kernels [27], where

- *Resource set*: A resource set corresponds to a parallel task. It is a container of multiple reserves.
- *Reserve*: A reserve represents the amount of CPU budget to be reserved on a single core or multiple cores. A reserve is specified with (C, T, D, ϕ): C is a worst-case execution time; T is a period; D is a relative deadline; ϕ is a release offset.

Figure 5 shows the scheduling of a parallel real-time task on four cores with the *stretch** transformation. The parallel task τ_1 has one parallel segment comprising four threads.

[3]This portion is obtained by running Algorithm 1.

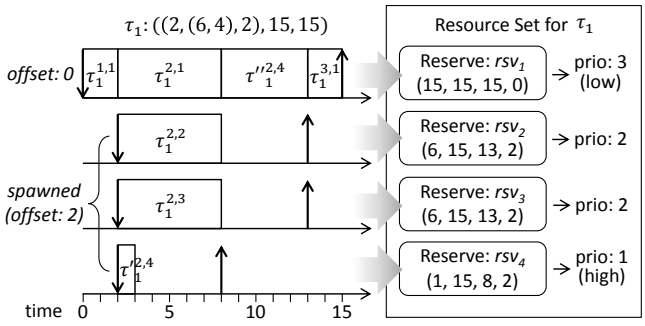

Figure 5: CPU resource abstraction for a parallel task with global DM scheduling

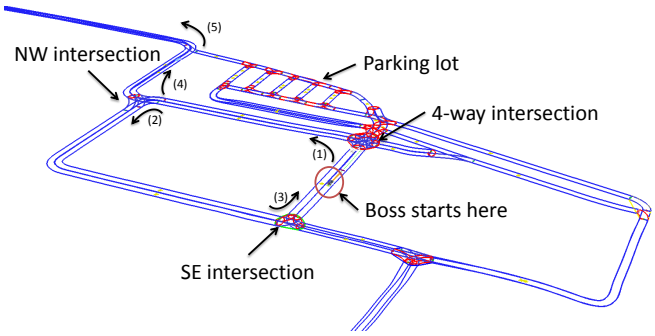

Figure 6: The map followed by Boss

The *stretch** transformation splits the last thread of the parallel segment, $\tau_1^{2,4}$, into $\tau'^{2,4}_1$ and $\tau''^{2,4}_1$. Hence, $\tau'^{2,4}_1$ is assigned a relative deadline of 8 that is equal to the release offset of $\tau''^{2,4}_1$. The CPU usage and its offset on each core can be represented as a reserve. Since a reserve is equivalent to an individual sequential periodic task, the global DM scheduling algorithm can determine the scheduling priorities for reserves. Then, we assign reserves to threads so that each thread is scheduled with the priority and the release offset of the assigned reserve and consumes the reserve's CPU budget. The master string thread, $(\tau_1^{1,1} \rightarrow \tau_1^{2,1} \rightarrow \tau_1^{3,1})$, is assigned a reserve (rsv_1). The second and the third thread in the parallel segment, $\tau_1^{2,2}$ and $\tau_1^{2,3}$, are assigned (rsv_2) and (rsv_3), respectively. The last thread $\tau_1^{2,4}$ is assigned an ordered list of reserves, $(rsv_4 \rightarrow rsv_1)$. This means that $\tau_1^{2,4}$ first uses rsv_4's priority and CPU budget, and when it uses up rsv_4's budget, it continues its execution with rsv_1's priority and remaining CPU budget.

We implemented the abstraction for parallel tasks on Linux/RK [25], which is based on the Linux 2.6.38.8 kernel. We used `hrtimers` to release threads at specified offsets and to account the CPU usage of threads. When a thread uses up all reserves assigned to it, the abstraction enforces the CPU usage of the thread by suspending it. The accounting and the enforcement of our abstraction can also be used for the measurement-based worst-case-execution-time estimation of threads in a parallel task, by checking an occurrence of the enforcement with a tentative execution time.

6. CASE STUDY ON SELF-DRIVING CAR

We studied the efficacy of our proposed scheme using a self-driving car platform Boss. The latest motion planning algorithm running on Boss [17] is used for our evaluation. The algorithm considers the distance to the next destination, the lateral offset of the car to the center of the lane, the longitudinal velocity, the longitudinal acceleration, the lateral acceleration and a list of static/dynamic obstacles on the road where the vehicle is driving. With the given information based on which the number of parallel threads varies, the algorithm generates curvature and velocity profiles for the path which the vehicle should follow. The planning algorithm is implemented using `OpenMP`, and we evaluate the quality of autonomous driving by analyzing curvature and velocity profiles of Boss (1) when the conventional reservation approach with Linux/RK [25] is used, (2) when the previous task model [21] is used, and (3) when our proposed task model and algorithm are used.

We ran the planning algorithm on a simulation cluster [23, 19] equipped with an Intel Core i7 quad-core processor. Although we run the exact same algorithm on the vehicle, we measure the results on the simulation cluster due to testing, convenience and safety considerations. We ran a scenario with the layout of our test track located at Robot City in Hazelwood, Pittsburgh, PA, where we test the vehicle at straight multi-lane roads, curvy roads, intersections, U-turns and parking lots. The exact same scenario file is also used during the field test, but the tasks for receiving raw sensor data are replaced with simulation tasks. In Figure 6, the test track for the scenario is illustrated. Boss will depart at the point circled in the middle of Figure 6. Boss will follow the road, (1) cross a 4-way intersection governed by stop signs, follow the straight road and (2) make a left turn at NW intersection. Then, Boss will (3) make a left turn at SE intersection, proceed to NW intersection and (4) turn right towards the curve marked with (5) connecting to the long straight road.

The scenario is composed of eight tasks: `BehaviorTask`, `MissionPlannerTask`, `OnRoadMotionPlannerTask`, `PrePlannerTask`, `RoadBlockageDetector`, `RobotClient`, `ServerTask` and `SimpleControllerTask`. The `BehaviorTask` decides what to do such as turning, intersection handling and lane changing. The `MissionPlannerTask` interacts with the stored map to decide where to go. The `OnRoadMotionPlannerTask` and the `PrePlannerTask` send trajectories to the vehicle controller. The `RoadBlockageDetector` works with the `BehaviorTask` so that the vehicle detects the blocked road and finds an alternate route when needed. The `SimpleControllerTask` receives the actuator commands and directly interfaces with the vehicle hardware such as the accelerator, the brake and the steering wheel. On the simulation cluster, this task operates in simulation mode, and the `ServerTask` and the `RobotClient` behave as the vehicle hardware. In this paper, our focus is on the `OnRoadPlannerTask` running the motion planning algorithm [17] with `OpenMP` enabled. The task generates curvature and velocity profiles for the vehicle hardware, so the lack of resources will affect the control algorithm, making the car drive in an unstable manner. If the planning algorithm does not meet the deadline, the steering wheel, for example, jerks and the car goes to an unexpected place, which can cause an accident.

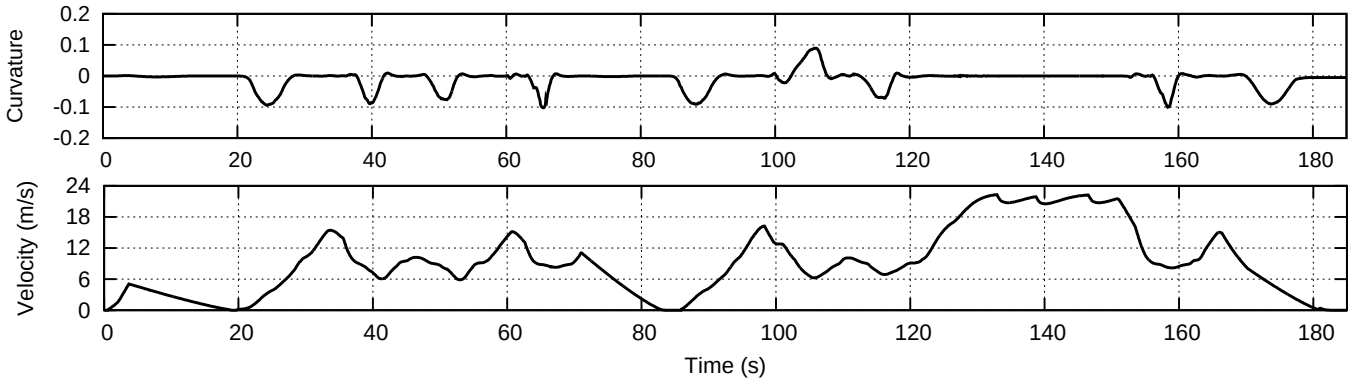

Figure 7: Curvature and velocity profiles during the entire journey of Boss illustrated in Figure 6.

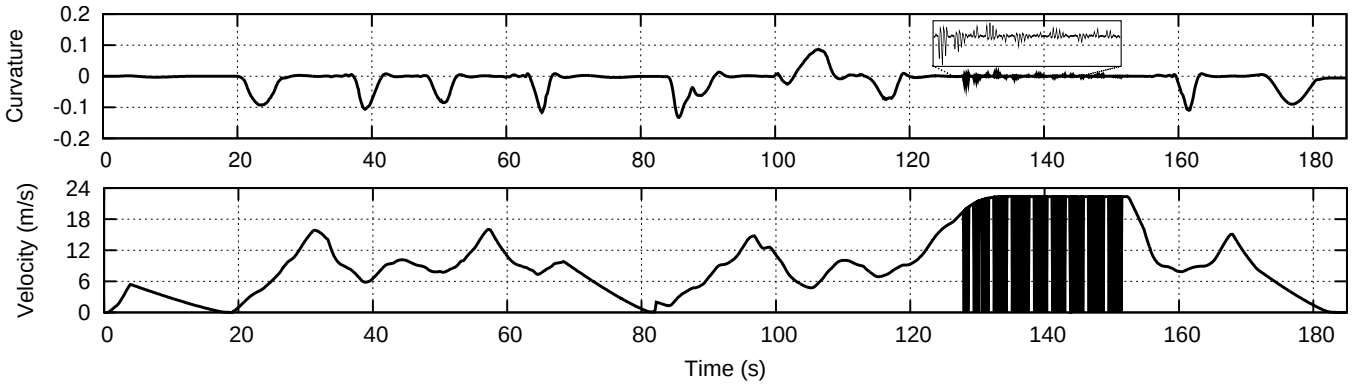

Figure 8: Curvature and velocity profiles of Boss when conventional resource reservation is used.

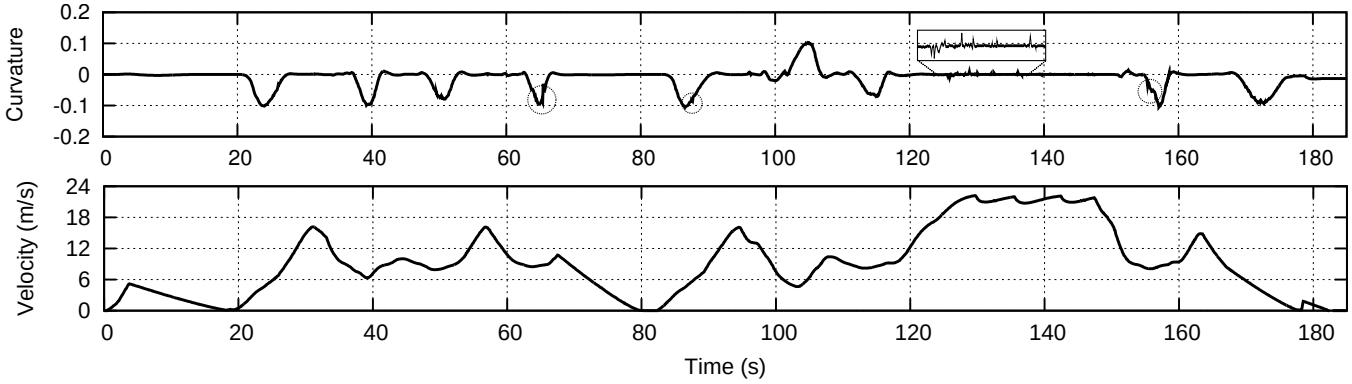

Figure 9: Curvature and velocity profiles of Boss when previously known techniques [21] are used.

Figure 7 shows the autonomous driving performance, i.e., the curvature and velocity profiles collected from the output of `OnRoadMotionPlannerTask` when the proposed task model and algorithm are used with a varying number of threads. We limit the maximum number of threads to 50. The curvature graph shows when Boss makes turns; a negative value means a left rotation of steering wheel, and vice versa. For example, Boss arrives at the SE intersection in Figure 7 around $t = 65s$, and that is the fourth valley in the curvature graph. Accordingly, we can see the velocity of Boss decreases to turn left. The bigger the absolute value of curvature, the steeper will be the turn made by Boss. From the perspective of autonomous driving quality, a sudden control change on an actuator is not desirable.

Figure 8 shows an undesirable case when the *conventional*

resource reservation approach with Linux/RK is used. Since the traditional Linux/RK does not consider a parallel task model, it assigns all child threads into a reserve allocated to a processing core. Since this may prevent the planning algorithm from running in parallel, the planner may not be able to meet its deadline, which is shown from $130s$ to $150s$ in Figure 8. The planning algorithm requires more threads when a car is moving faster and/or when a car is making a sharp turn. The results shown, therefore, are consistent with the property of the planning algorithm. Figure 9 also shows the result when the model of [21] is used, where only four threads can run in parallel because the simulation cluster has a quad-core processor. For this case, the velocity profiles are fine, but the curvatures show some jitters that can make the vehicle unstable and also uncomfortable for passengers. The results shown in Figure 8 and 9 could be potentially dangerous on the real vehicle because the vehicle in the real-world may slip, drift and crash.

7. RELATED WORK

Since Dhall and Liu [12] showed that RM and EDF scheduling could utilize only one processor regardless of how many processors a system had, there has been extensive research on global real-time scheduling [3, 15, 6, 5, 7, 8, 9, 11], where a comprehensive survey can be found in [11]. It is well-known that the anomaly of global scheduling happens when a set of tasks has two types of tasks: tasks with a low ratio of the worst-case execution time to relative deadline and tasks with a high ratio of the worst-case execution time to relative deadline. Many algorithms have been invented to avoid such cases, and corresponding schedulability tests have been proposed. Using our proposed task transformation, any existing global scheduling algorithm can be applied to schedule parallel real-time tasks. In this paper, we have used the schedulability bounds for global DM proposed in [6, 9].

There has not been much research on scheduling parallel real-time tasks [18, 21, 29, 24]. Lakshmanan et al. [21] proposed a fork-join real-time task model composed of alternating sequential and parallel segments. They also provided the analysis and resource augmentation bound for the partitioned DM scheduling [13] of parallel real-time tasks using the task *stretch* transformation. The proposed multiprocessor scheduling algorithm is shown to have a resource augmentation bound of 3.42, which implies that any task set that is feasible on m unit-speed processors can be scheduled by the proposed algorithm on m processors that are 3.42 times faster. Our work is a generalization of this model and provides a resource augmentation bound when global scheduling is used.

Saifullah et al. [29] also proposed a parallel synchronization model that is also generalized from the fork-join task model in [21] so that a task can have an arbitrary number of threads per segment. Based on the proposed model, a task decomposition method is used to decompose each parallel task into a set of sequential tasks. The task decomposition achieves a resource augmentation bound of 4 and 5 when the decomposed tasks are scheduled using global EDF and partitioned DM scheduling, respectively. Our work focuses more on global fixed-priority scheduling and shows the evaluation results measured from a real-world implementation.

More recently, Nelissen et al. [24] presented both offline and online algorithms to minimize the number of cores to be used to schedule multi-threaded tasks using a similar model to the model proposed in [29]. By using scheduling algorithms which can guarantee the schedulability of the given tasks as long as the sum of densities of all the given tasks is less than or equal to the number of processing cores, they obtained a resource augmentation bound of 2. Our perspective is different from theirs in a sense that we schedule a set of tasks under a given hardware constraint (the number of processing cores) rather than finding hardware for the given tasks. We also use global DM scheduling algorithm more commonly used in practice and show the evaluation results obtained from a working system.

Apart from work using the *Thread* model mentioned above, there has also been research based on *gang scheduling*, where all parallel components of the same task should arrive and complete at the same time. Gang EDF [18] was proposed to address gang scheduling in the real-time context. Our work is different from this in two ways: (1) our model allows the parallel segments to be preempted during the parallel execution, and (2) a different number of parallel threads can be used.

8. SUMMARY AND FUTURE WORK

To meet rapidly increasing demands for complex cyber-physical systems, we motivated the necessity of using multi-core processors and corresponding parallel programming models such as `OpenMP` [1]. In particular, emerging CPS such as a self-driving vehicle can benefit significantly from parallel real-time tasks allowing multiple compute-intensive real-time tasks to support demanding requirements. Thus, a self-driving vehicle can model its physical surroundings in parallel and react to them in real-time. In this paper, we proposed a fork-join parallel real-time task model, where the amount of parallel executions can vary depending on the physical attributes of the system. The proposed task model is transformed using our *stretch** algorithm. With global deadline-monotonic scheduling, we obtained a resource augmentation bound of 3.73, which means that any task set that is feasible on m unit-speed processors can be scheduled by the proposed algorithm on m processors that are 3.73 times faster. The proposed scheme was implemented on Linux/RK [25] as a proof of concept, and ported to Boss, the self-driving car that won the 2007 DARPA Urban Challenge [31]. On Boss, we evaluated our proposed scheme that improved its autonomous driving quality. Future work to be done includes supporting dynamic changes of periods and execution times of parallel real-time tasks. We already have early work on varying periods [20], and the dynamic nature of CPS will be addressed using this model combined with parallel tasks.

Acknowledgments

The authors would like to thank Tianyu Gu and Junqing Wei for their valuable comments on the case study.

9. REFERENCES

[1] OpenMP. `http://openmp.org`.
[2] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In

Proceedings of the April 18-20, 1967, spring joint computer conference, pages 483–485. ACM, 1967.

[3] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *22nd IEEE Real-Time Systems Symposium*, 2001.

[4] AUTOSAR Administration. Specification of Operating System V5.0.0 R4.0 Rev 3, 2011.

[5] T. Baker. An analysis of deadline-monotonic schedulability on a multiprocessor. Technical report, TR-030201, Department of Computer Science, Florida State University, 2003.

[6] T. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 120 – 129, dec. 2003.

[7] T. Baker. An analysis of edf schedulability on a multiprocessor. *Parallel and Distributed Systems, IEEE Transactions on*, 16(8):760–768, 2005.

[8] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, 2006.

[9] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 2009.

[10] BMW ConnectedDrive. Take Over Please! `http://www.bmw.com/com/en/insights/technology/connecteddrive/2010/future_lab/index.html#/1/4` as of Jan 31, 2013.

[11] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, Oct. 2011.

[12] S. Dhall and C. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[13] N. Fisher, S. Baruah, and T. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *18th Euromicro Conference on Real-Time Systems*, 2006.

[14] Freescale Semiconductor, Inc. MPC8640: MPC8640D Integrated Dual-Core Processor. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8640` as of Jan 31, 2013.

[15] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. In *22nd IEEE Real-Time Systems Symposium*, pages 183 – 192, dec. 2001.

[16] GM Press. Self-Driving Car in Cadillac's Future. `http://media.gm.com/media/us/en/cadillac/news.detail.html/content/Pages/news/us/en/2012/Apr/0420_cadillac.html` as of Jan 31, 2013.

[17] T. Gu and J. Dolan. On-road motion planning for autonomous vehicles. *Intelligent Robotics and Applications*, 2012.

[18] S. Kato and Y. Ishikawa. Gang edf scheduling of parallel task systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 459 –468, dec. 2009.

[19] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim. SAFER: System-level Architecture for Failure Evasion in Real-time Applications. In *Proc. of the 33rd IEEE Real-Time Systems Symposium*, 2012.

[20] J. Kim, K. Lakshmanan, and R. Rajkumar. Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems. In *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pages 55 –64, april 2012.

[21] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010.

[22] Luca de Ambroggi. Multicore Trends in Automotive Offer Cost Savings, Higher Performance. iSuppli, 2011.

[23] M. McNaughton, et al. Software infrastructure for an autonomous ground vehicle. *Journal of Aerospace Computing, Information, and Communication*, 5(1):491 – 505, 2008.

[24] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *the 24th Euromicro Conference on Real-Time Systems*, 2012.

[25] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in linux. In *IEEE Real-Time Systems Sumposium (RTSS) Work-In-Progress*, 1998.

[26] S. Pinkow. Continental Tests Highly-Automated Driving. `http://www.conti-online.com/generator/www/com/en/continental/pressportal/themes/press_releases/3_automotive_group/chassis_safety/press_releases/pr_2012_03_23_automated_driving_en.html` as of Jan 31, 2013.

[27] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[28] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 731–736, New York, NY, USA, 2010. ACM.

[29] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *the 32nd IEEE Real-Time Systems Symposium*, 2011.

[30] S. Thrun. What we're driving at. `http://googleblog.blogspot.com/2010/10/what-were-driving-at.html` as of Jan 31, 2013.

[31] C. Urmson, J. Anhalt, H. Bae, D. Bagnell, C. Baker, R. Bittner, T. Brown, M. Clark, M. Darms, D. Demitrish, J. Dolan, D. Duggins, D. Ferguson, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, S. Kolski, M. Likhachev, B. Litkouhi, A. Kelly, M. McNaughton, N. Miller, J. Nickolaou, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, V. Sadekar, B. Salesky, Y.-W. Seo, S. Singh, J. Snider, J. Struble, A. Stentz, M. Taylor, W. Whittaker, Z. Wolkowicki, W. Zhang, and J. Ziglar. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, 25(1):425–466, June 2008.