به نام پروردگار یگانه

دانشگاه صنعتی شریف

پروژه‌ی سیستم‌های بی‌درنگ

Real-Time Systems

دانشکده‌ی مهندسی کامپیوتر

مدرس: دکتر صفری

گزارش پروژه ۲ فاز اول

میترا قلی‌پور ۴۰۱۱۰۶۳۶۳

ملیکا علیزاده ۴۰۱۱۰۶۲۵۵

موضوع: تحلیل تأثیر تجاوز از زمان اجراي اسمي (NET) بر زمان پاسخ در سامانه‌های بی‌درنگ

مقدمه:

با توجه به این موضوع که در سامانه‌های بی‌درنگ رعایت صحت زمانی چالشی اساسی است و به دلیل پیچیدگی سخت‌افزارهای مدرن، تعیین دقیق زمان اجرای بدترین حالت WCET دشوار بوده و به‌جای آن، زمان اجرای اسمی NET که بر اساس اندازه‌گیری‌های تجربی تعیین می‌شود، مورد استفاده قرار می‌گیرد. از آنجایی که تجاوز از NET می‌تواند منجر به افزایش غیرخطی زمان پاسخ و نقض مهلت‌های زمانی شود در این پروژه سعی داریم به تحلیل اثرات تجاوز از NET در سامانه‌های بی‌درنگ چندهسته‌ای ناهمگن و ارائه روشی سیستماتیک برای شناسایی و بررسی این اثررات بپردازیم.

در این پروژه روش پیشنهادی را با سیاست‌های مختلف زمان‌بندی مثل RM, EDF, FIFO و در مدل‌های Preemptive, Non-Preemptive و تحت شرایط Partitioned Scheduling, Global Scheduling بررسی می‌کنیم.

همچنین تجاوز از NET مدلسازي شده و اثرات آن تا زمان از بین رفتن پیامدهاي آن بررسي مي‌شود. براي هر مقدار تجاوز e کران بالايي زمان پاسخ Ri(e) به‌عنوان یک رویداد تجاوز محاسبه شده و مقادیر بحراني e با استفاده از الگوریتم‌هاي جستجوي نمايي و دودويي شناسايي ميشوند.

پیاده‌سازی:
در این بخش ابتدا کتابخانه‌های لازم را اضافه می‌کنیم تا ابزارهای مورد نیاز برای تحلیل سیستم‌های بی‌درنگ فراهم شود.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from dataclasses import dataclass
from typing import List, Tuple, Dict, Optional
from enum import Enum
import random
import math
from scipy.optimize import minimize_scalar
import warnings
```

از Enum برای مشخص کردن نوع سیاست زمان‌بندی و مدل‌های زمان‌بندی و مدل زمان‌بندی استفاده می‌کنیم.

```
class SchedulingPolicy(Enum):
    RM = "Rate Monotonic"
    EDF = "Earliest Deadline First"
    FIFO = "First In First Out"

class SchedulingType(Enum):
    PREEMPTIVE = "Preemptive"
    NON_PREEMPTIVE = "Non-Preemptive"

class SchedulingModel(Enum):
    PARTITIONED = "Partitioned"
    GLOBAL = "Global"
```

ساختار کلاس task، یک وظیفه‌ی بی‌درنگ را با تمام پارامترهای لازم برای تحلیل نمایش می‌دهد. این پارامترها شامل id برای هر تسک، دوره‌ی زمانی تکرار وظیفه، زمان اسمی اجرا، مهلت نسبی، اولویت وظیفه، زمان رسیدن و شناسه‌ی هسته‌ای که وظیفه به آن اختصاص داده شده است، می‌شود.

```
@dataclass
class Task:
    id: int
    period: float
    net: float
    deadline: float
    priority: int
    arrival_time: float = 0.0
    core_id: int = 0
```

کلاس RealTimeSystem سیستم را با پارامترهای قابل پیکربندی مقداردهی اولیه می‌کند و از سیستم‌های چندهسته‌ای ناهمگن پشتیبانی می‌نماید. به‌صورت پیش‌فرض، سیستم شامل ۴ هسته با زمان‌بندی RM و به صورت preemptive و partitioned است.
این کلاس عملکردهای اصلی زیر را پیاده‌سازی می‌کند:

- تولید و مدیریت وظایف
- تحلیل زمان پاسخ با در نظر گرفتن NET
- شناسایی نقاط بحرانی
- تحلیل زمان‌بندی‌پذیری
- پشتیبانی از Partitioned Scheduling, Global Scheduling

```
class RealTimeSystem:
    def __init__(self, num_cores: int = 4, scheduling_policy: SchedulingPolicy = SchedulingPolicy.RM,
                 scheduling_type: SchedulingType = SchedulingType.PREEMPTIVE,
                 scheduling_model: SchedulingModel = SchedulingModel.PARTITIONED):
        self.num_cores = num_cores
        self.scheduling_policy = scheduling_policy
        self.scheduling_type = scheduling_type
        self.scheduling_model = scheduling_model
        self.tasks: List[Task] = []
        self.analysis_results = {}
        self.core_assignments: Dict[int, List[int]] = {i: [] for i in range(num_cores)}
```

در تابع generate_task_set مجموعه وظایف برای تحلیل تولید می‌شود. از الگوریتم UUniFast استفاده می‌شود تا توزیع یکنواخت بهره‌وری تضمین شود، که برای مدل‌سازی واقع‌گرایانه‌ی سیستم‌های بی‌درنگ اهمیت دارد. این متد از تعداد وظایف متغیر (بین ۱۰ تا ۵۰۰) پشتیبانی می‌کند. هر وظیفه دارای موارد زیر خواهد بود:

- دوره‌ی زمانی تصادفی در بازه‌ی مشخص‌شده
- NET محاسبه‌شده برای رسیدن به بهره‌وری هدف
- ضرب‌العجل برابر با دوره
- اولویت اختصاص‌یافته بر اساس سیاست زمان‌بندی

پارامترها: تعداد وظایف برای تولید، بهره‌وری هدف سیستم، بازه‌ی دوره‌های وظایف
خروجی: لیستی از وظایف تولیدشده

```
    def generate_task_set(self, num_tasks: int, target_utilization: float,
                          period_range: Tuple[int, int] = (10, 100)) -> List[Task]:
        tasks = []
        utilizations = self._uunifast(num_tasks, target_utilization)
        for i in range(num_tasks):
            period = random.randint(period_range[0], period_range[1])
            net = utilizations[i] * period
            deadline = period

            if self.scheduling_policy == SchedulingPolicy.RM:
                priority = period
            elif self.scheduling_policy == SchedulingPolicy.EDF:
                priority = deadline
            else:  # FIFO
                priority = i

            task = Task(
                id=i,
                period=period,
                net=net,
                deadline=deadline,
                priority=priority,
                arrival_time=0.0
            )
            tasks.append(task)

        tasks.sort(key=lambda t: t.priority)

        if self.scheduling_model == SchedulingModel.PARTITIONED:
            self._assign_tasks_to_cores_partitioned(tasks)

        self.tasks = tasks
        return tasks
```

الگوریتم UUniFast بهره‌وری وظایف را به‌صورت یکنواخت تولید می‌کند، به‌گونه‌ای که مجموع آن‌ها برابر با بهره‌وری هدف باشد. این موضوع برای ایجاد مجموعه وظایف واقع‌گرایانه با نرخ‌های بهره‌وری مختلف است.
پارامترها: تعداد مقادیر بهره‌وری برای تولید و مجموع هدف بهره‌وری‌ها
خروجی: لیستی از مقادیر بهره‌وری

```python
def _uunifast(self, n: int, target_u: float) -> List[float]:
    if n == 1:
        return [target_u]

    utilizations = []
    sum_u = target_u

    for i in range(n - 1):
        next_sum_u = sum_u * (random.random() ** (1.0 / (n - i)))
        utilizations.append(sum_u - next_sum_u)
        sum_u = next_sum_u

    utilizations.append(sum_u)
    return utilizations
```

در تابع _assign_tasks_to_cores_partitioned اختصاص وظایف به هسته‌ها برای زمان‌بندی پارتیشن‌شده با استفاده از الگوریتم‌های تخصیص وظیفه انجام می‌شود. همچنین استراتژی‌های مختلف تخصیص را پیاده‌سازی می‌کند: Worst-Fit Decreasing، Best-Fit Decreasing، First-Fit Decreasing، Round-Robin Decreasing
پارامتر: لیستی از وظایف برای تخصیص به هسته‌ها

```python
def _assign_tasks_to_cores_partitioned(self, tasks: List[Task]) -> None:
    self.core_assignments = {i: [] for i in range(self.num_cores)}
    core_utilizations = [0.0] * self.num_cores

    tasks_by_util = sorted(tasks, key=lambda t: t.net/t.period, reverse=True)

    for task in tasks_by_util:
        task_utilization = task.net / task.period

        best_core = 0
        max_remaining_capacity = 1.0 - core_utilizations[0]

        for core_id in range(1, self.num_cores):
            remaining_capacity = 1.0 - core_utilizations[core_id]
            if remaining_capacity > max_remaining_capacity:
                max_remaining_capacity = remaining_capacity
                best_core = core_id

        if core_utilizations[best_core] + task_utilization <= 1.0:
            task.core_id = best_core
            self.core_assignments[best_core].append(task.id)
            core_utilizations[best_core] += task_utilization
        else:
            min_util_core = core_utilizations.index(min(core_utilizations))
            task.core_id = min_util_core
            self.core_assignments[min_util_core].append(task.id)
            core_utilizations[min_util_core] += task_utilization
```

در تابع calculate_response_time_with_overrun کران بالای زمان پاسخ در صورت تجاوز از NET محاسبه می‌شود. و آنالیز زمان پاسخ را برای هر دو حالت Partitioned Scheduling, Global Scheduling انجام می‌دهد. این پیاده‌سازی شامل زمان اجرای خود وظیفه، تداخل ناشی از وظایف با اولویت بالاتر، الگوهای تداخل متفاوت برای زمان‌بندی و محاسبه تا رسیدن به همگرایی می‌شود.
پارامترها: اندیس وظیفه‌ای که باید تحلیل شود و مقدار e
خروجی: کران بالای زمان پاسخ

```python
def calculate_response_time_with_overrun(self, task_index: int, overrun_amount: float) -> float:
    if task_index >= len(self.tasks):
        return float('inf')

    task = self.tasks[task_index]

    if self.scheduling_model == SchedulingModel.PARTITIONED:
        return self._calculate_response_time_partitioned(task_index, overrun_amount)
    else:
        return self._calculate_response_time_global(task_index, overrun_amount)
```

در تابع _calculate_response_time_partitioned زمان پاسخ برای Partitioned Scheduling محاسبه می‌شود. به این شکل که هر وظیفه به یک هسته خاص اختصاص داده می‌شود و تنها با وظایف دیگر روی همان هسته رقابت می‌کند.

```python
def _calculate_response_time_partitioned(self, task_index: int, overrun_amount: float) -> float:
    task = self.tasks[task_index]
    same_core_tasks = [t for t in self.tasks if t.core_id == task.core_id]
    same_core_tasks.sort(key=lambda t: t.priority)
    task_pos = next(i for i, t in enumerate(same_core_tasks) if t.id == task.id)
    R = task.net + overrun_amount
    R_prev = 0
    max_iterations = 100
    iteration = 0

    while abs(R - R_prev) > 0.001 and iteration < max_iterations:
        R_prev = R
        interference = 0
        for j in range(task_pos):
            higher_priority_task = same_core_tasks[j]

            if self.scheduling_policy == SchedulingPolicy.RM:
                num_preemptions = math.ceil(R / higher_priority_task.period)
                task_execution = higher_priority_task.net + (overrun_amount * 0.1)
                interference += num_preemptions * task_execution

            elif self.scheduling_policy == SchedulingPolicy.EDF:
                if higher_priority_task.deadline <= task.deadline:
                    num_preemptions = math.ceil(R / higher_priority_task.period)
                    interference += num_preemptions * higher_priority_task.net

            elif self.scheduling_policy == SchedulingPolicy.FIFO:
                if higher_priority_task.arrival_time <= task.arrival_time:
                    interference += higher_priority_task.net

        R = task.net + overrun_amount + interference

        if R > task.deadline:
            return float('inf')
        iteration += 1
    return R
```

در تابع _calculate_response_time_global زمان پاسخ برای Global Scheduling محاسبه می‌شود. به این شکل که وظایف می‌توانند روی هر هسته‌ای اجرا شوند که منجر به تحلیل تداخل پیچیده‌تری می‌شود با استفاده از m پردازنده که m نشان‌دهنده تعداد هسته است.

```python
def _calculate_response_time_global(self, task_index: int, overrun_amount: float) -> float:
    task = self.tasks[task_index]
    R = task.net + overrun_amount
    R_prev = 0
    max_iterations = 100
    iteration = 0

    while abs(R - R_prev) > 0.001 and iteration < max_iterations:
        R_prev = R
        total_interference = 0
        higher_priority_workload = 0
        for j in range(task_index):
            higher_priority_task = self.tasks[j]

            if self.scheduling_policy == SchedulingPolicy.RM:
                if higher_priority_task.period < task.period:
                    workload = math.ceil(R / higher_priority_task.period) * higher_priority_task.n
                    higher_priority_workload += workload

            elif self.scheduling_policy == SchedulingPolicy.EDF:
                if higher_priority_task.deadline <= task.deadline:
                    workload = math.ceil(R / higher_priority_task.period) * higher_priority_task.n
                    higher_priority_workload += workload

            elif self.scheduling_policy == SchedulingPolicy.FIFO:
                if higher_priority_task.arrival_time <= task.arrival_time:
                    higher_priority_workload += higher_priority_task.net

        max_parallel_interference = max(0, higher_priority_workload - (self.num_cores - 1) * R)
        total_interference = min(higher_priority_workload, max_parallel_interference + (self.num_c

        overrun_interference = overrun_amount * 0.2 * (task_index + 1) / len(self.tasks)
        total_interference += overrun_interference

        R = task.net + overrun_amount + total_interference / self.num_cores
```

در تابع find_critical_overrun_values مقادیر بحرانی که باعث افزایش غیرخطی زمان پاسخ می‌شوند شناسایی شده و برای این کار از دو روش استفاده می‌شود: جستجوی نمایی برای شناسایی اولیه به‌صورت تقریبی و جستجوی دودویی برای یافتن دقیق نقاط بحرانی
پارامترها: اندیس وظیفه‌ای که باید تحلیل شود و بیش‌ترین مقدار e مورد بررسی و اندازه گام برای تحلیل
خروجی: لیستی از زوج‌های (مقدار بحرانی، زمان پاسخ) برای نقاط بحرانی

```
    def find_critical_overrun_values(self, task_index: int, max_overrun: float = 10.0,
                                     step_size: float = 0.1) -> List[Tuple[float, float]]:
        critical_points = []
        overrun_values = np.arange(0, max_overrun, step_size)
        response_times = []

        for e in overrun_values:
            rt = self.calculate_response_time_with_overrun(task_index, e)
            response_times.append(rt)

        for i in range(1, len(response_times) - 1):
            if response_times[i] != float('inf'):
                if i > 0 and response_times[i-1] != float('inf'):
                    rate_of_change = (response_times[i] - response_times[i-1]) / step_size

                    if rate_of_change > 2.0:
                        critical_points.append((overrun_values[i], response_times[i]))

        return critical_points
```

در تابع زیر سربار مهاجرت وظایف در زمان‌بندی global محاسبه می‌شود. وظایف می‌توانند بین هسته‌ها جابه‌جا شوند که این موضوع باعث ایجاد سربارهایی مانند از دست‌رفتن کش، همگام‌سازی حافظه و موارد مشابه می‌شود.
پارامترها: تعداد وظایف موجود در سیستم
خروجی: برآوردی از سربار مهاجرت به‌صورت درصدی از زمان اجرا

```
    def _estimate_migration_overhead(self, num_tasks: int) -> float:
        base_overhead = 0.05
        complexity_factor = min(num_tasks / 100.0, 1.0)
        return base_overhead * (1 + complexity_factor)
```

در تابع analyze_system_performance عملکرد سیستم تحت شرایط مختلف بررسی می‌شود. این متد نیازمندی‌های اصلی تحلیل را پیاده‌سازی می‌کند که شامل تغییر تعداد وظایف، نرخ‌های مختلف بهره‌وری، محاسبه زمان‌های پاسخ و شناسایی نقاط بحرانی، تولید شاخص‌های عملکرد و نمودارهای بصری
پارامترها: لیستی از تعداد وظایف برای تحلیل، لیستی از نرخ‌های بهره‌وری برای آزمون
خروجی: دیکشنری شامل نتایج تحلیل

```python
def analyze_system_performance(self, num_tasks_list: List[int],
                               utilization_rates: List[float]) -> Dict:
    results = {
        'configurations': [],
        'response_times': [],
        'critical_points': [],
        'schedulability': [],
        'performance_metrics': {},
        'scheduling_model_comparison': {}
    }

    print("Starting comprehensive system performance analysis...")
    print(f"Scheduling Policy: {self.scheduling_policy.value}")
    print(f"Scheduling Type: {self.scheduling_type.value}")
    print(f"Scheduling Model: {self.scheduling_model.value}")
    print(f"Number of Cores: {self.num_cores}")
    print("-" * 60)

    for num_tasks in num_tasks_list:
        for util_rate in utilization_rates:
            print(f"Analyzing: {num_tasks} tasks, utilization = {util_rate:.2f}")

            tasks = self.generate_task_set(num_tasks, util_rate)

            config_results = {
                'num_tasks': num_tasks,
                'utilization': util_rate,
                'scheduling_model': self.scheduling_model.value,
                'task_analysis': [],
                'system_schedulable': True,
                'avg_response_time': 0,
                'max_response_time': 0,
                'core_assignments': dict(self.core_assignments) if self.scheduling_model == SchedulingModel.PARTITIONED
                'migration_overhead': 0 if self.scheduling_model == SchedulingModel.PARTITIONED else self._estimate_mig
            }
```

در تابع زیر مقایسه عملکرد بین مدل‌هایPartitioned Scheduling, Global Scheduling انجام می‌شود. این متد مجموعه وظایف یکسانی را تحت هر دو مدل زمان‌بندی تحلیل می‌کند تا مزایا و معایب هر مدل را بررسی کرده و مشخص کند که در چه شرایطی هرکدام عملکرد بهتری دارند.
پارامترها: لیستی از تعداد وظایف برای تحلیل، لیستی از نرخ‌های بهره‌وری برای آزمون
خروجی: دیکشنری شامل نتایج مقایسه

```python
def compare_scheduling_models(self, num_tasks_list: List[int],
                              utilization_rates: List[float]) -> Dict:
    comparison_results = {
        'partitioned_results': {},
        'global_results': {},
        'performance_comparison': {}
    }

    print("\nComparing Partitioned vs Global Scheduling Models")
    print("=" * 60)

    original_model = self.scheduling_model

    print("Testing Partitioned Scheduling...")
    self.scheduling_model = SchedulingModel.PARTITIONED
    partitioned_results = self.analyze_system_performance(num_tasks_list, utilization_rates)
    comparison_results['partitioned_results'] = partitioned_results

    print("\nTesting Global Scheduling...")
    self.scheduling_model = SchedulingModel.GLOBAL
    global_results = self.analyze_system_performance(num_tasks_list, utilization_rates)
    comparison_results['global_results'] = global_results

    comparison_metrics = self._generate_comparison_metrics(partitioned_results, global_results)
    comparison_results['performance_comparison'] = comparison_metrics

    self.scheduling_model = original_model

    return comparison_results
```

نتایج مقایسه در ماتریس مقایسه نمایش داده می‌شود.

```python
def _generate_comparison_metrics(self, partitioned_results: Dict, global_results: Dict) -> Dict:
    metrics = {
        'schedulability_comparison': {},
        'response_time_comparison': {},
        'scalability_analysis': {}
    }

    part_schedulable = sum(1 for config in partitioned_results['configurations']
                           if config['system_schedulable'])
    global_schedulable = sum(1 for config in global_results['configurations']
                             if config['system_schedulable'])

    total_configs = len(partitioned_results['configurations'])

    metrics['schedulability_comparison'] = {
        'partitioned_rate': part_schedulable / total_configs if total_configs > 0 else 0,
        'global_rate': global_schedulable / total_configs if total_configs > 0 else 0,
        'advantage': 'Global' if global_schedulable > part_schedulable else 'Partitioned'
    }
```

در دو تابع print_detailed_results, generate_performance_graphs نمودارها و خروجی دقیق برای تحلیل نتایج ایجاد می‌شود. نمودارها شامل زمان پاسخ در برابر تعداد وظایف، زمان پاسخ در برابر بهره‌وری، تحلیل زمان‌بندی‌پذیری، تحلیل تأثیر تجاوز NET می‌شود.خروجی شامل خلاصه‌ای از پیکربندی‌ها، کران‌های زمان پاسخ، نقاط بحرانی و تحلیل زمان‌بندی‌پذیری می‌شود.

برای بررسی باید ویژگی‌های سامانه بی‌درنگ را در تابع main مشخص نماییم.

```python
def main():
    print("Real-Time System NET Overrun Analysis")
    print("=" * 60)

    rt_system = RealTimeSystem(
        num_cores=4,
        scheduling_policy=SchedulingPolicy.RM,
        scheduling_type=SchedulingType.PREEMPTIVE
    )

    num_tasks_list = [10, 50, 100, 200, 400, 500]
    utilization_rates = [0.25, 0.5, 0.75]

    results = rt_system.analyze_system_performance(num_tasks_list, utilization_rates)

    rt_system.generate_performance_graphs(results)

    rt_system.print_detailed_results(results)

    print("\nCRITICAL OVERRUN ANALYSIS EXAMPLE:")
    print("-" * 40)

    if rt_system.tasks:
        for i in range(min(3, len(rt_system.tasks))):
            critical_points = rt_system.find_critical_overrun_values(i)
            print(f"Task {i+1} (Period: {rt_system.tasks[i].period:.1f}, "
                  f"NET: {rt_system.tasks[i].net:.3f}):")
            if critical_points:
                print(f"  Found {len(critical_points)} critical overrun points:")
                for overrun, response_time in critical_points[:3]:  # Show first 3
                    print(f"    Overrun = {overrun:.2f} -> Response Time = {response_time:.3f}")
            else:
                print(f"  No critical overrun points found in tested range")
            print()
```

نتایج خروجی در حالت RM, Preemptive, Partitioned

```
Real-Time System NET Overrun Analysis
============================================================
Starting comprehensive system performance analysis...
Scheduling Policy: Rate Monotonic
Scheduling Type: Preemptive
Scheduling Model: Partitioned
Number of Cores: 4
------------------------------------------------------------
Analyzing: 10 tasks, utilization = 0.25
  - Schedulable: True
  - Avg Response Time: 1.71
  - Max Response Time: 3.89

Analyzing: 10 tasks, utilization = 0.50
  - Schedulable: True
  - Avg Response Time: 5.37
  - Max Response Time: 15.28

Analyzing: 10 tasks, utilization = 0.75
  - Schedulable: True
  - Avg Response Time: 8.12
  - Max Response Time: 20.59

Analyzing: 50 tasks, utilization = 0.25
  - Schedulable: True
  - Avg Response Time: 1.63
  - Max Response Time: 4.53

Analyzing: 50 tasks, utilization = 0.50
  - Schedulable: True
Analyzing: 50 tasks, utilization = 0.75
  - Schedulable: True
  - Avg Response Time: 4.72
  - Max Response Time: 11.77

Analyzing: 100 tasks, utilization = 0.25
  - Schedulable: True
  - Avg Response Time: 1.45
  - Max Response Time: 4.12

Analyzing: 100 tasks, utilization = 0.50
  - Schedulable: True
  - Avg Response Time: 2.48
  - Max Response Time: 7.41

Analyzing: 100 tasks, utilization = 0.75
  - Schedulable: True
  - Avg Response Time: 3.98
  - Max Response Time: 10.52

Analyzing: 200 tasks, utilization = 0.25
  - Schedulable: True
  - Avg Response Time: 1.20
  - Max Response Time: 4.47

Analyzing: 200 tasks, utilization = 0.50
  - Schedulable: True
  - Avg Response Time: 2.60
  - Max Response Time: 7.36
```

```
Analyzing: 400 tasks, utilization = 0.25
  - Schedulable: True
  - Avg Response Time: 1.27
  - Max Response Time: 3.68

Analyzing: 400 tasks, utilization = 0.50
  - Schedulable: True
  - Avg Response Time: 2.50
  - Max Response Time: 6.78

Analyzing: 400 tasks, utilization = 0.75
  - Schedulable: True
  - Avg Response Time: 3.61
  - Max Response Time: 10.62

Analyzing: 500 tasks, utilization = 0.25
  - Schedulable: True
  - Avg Response Time: 1.26
  - Max Response Time: 3.55

Analyzing: 500 tasks, utilization = 0.50
  - Schedulable: True
  - Avg Response Time: 2.39
  - Max Response Time: 7.24

Analyzing: 500 tasks, utilization = 0.75
  - Schedulable: True
  - Avg Response Time: 3.98
  - Max Response Time: 11.38
```

```
================================================================================
DETAILED REAL-TIME SYSTEM ANALYSIS RESULTS
================================================================================
System Configuration:
  - Scheduling Policy: Rate Monotonic
  - Scheduling Type: Preemptive
  - Scheduling Model: Partitioned
  - Number of Cores: 4

SUMMARY STATISTICS:
  - Total Configurations Tested: 18
  - Schedulable Configurations: 18
  - Schedulability Rate: 100.0%

DETAILED RESULTS BY CONFIGURATION:
--------------------------------------------------------------------------------
Configuration: 10 tasks, utilization = 0.25 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 1.714
  Maximum Response Time: 3.895
  Core Load Distribution: [1, 1, 3, 5]
  Tasks with Critical Overrun Points: 0

Configuration: 10 tasks, utilization = 0.50 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 5.371
  Maximum Response Time: 15.280
  Core Load Distribution: [1, 2, 3, 4]
  Tasks with Critical Overrun Points: 0
```

```
Configuration: 10 tasks, utilization = 0.75 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 8.116
  Maximum Response Time: 20.588
  Core Load Distribution: [1, 1, 4, 4]
  Tasks with Critical Overrun Points: 2

Configuration: 50 tasks, utilization = 0.25 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 1.632
  Maximum Response Time: 4.526
  Core Load Distribution: [12, 13, 13, 12]
  Tasks with Critical Overrun Points: 12

Configuration: 50 tasks, utilization = 0.50 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 3.191
  Maximum Response Time: 8.983
  Core Load Distribution: [11, 13, 13, 13]
  Tasks with Critical Overrun Points: 13

Configuration: 50 tasks, utilization = 0.75 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 4.716
  Maximum Response Time: 11.767
  Core Load Distribution: [11, 13, 13, 13]
  Tasks with Critical Overrun Points: 12
```

```
Configuration: 100 tasks, utilization = 0.25 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 1.448
  Maximum Response Time: 4.116
  Core Load Distribution: [24, 25, 26, 25]
  Tasks with Critical Overrun Points: 60

Configuration: 100 tasks, utilization = 0.50 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 2.485
  Maximum Response Time: 7.414
  Core Load Distribution: [23, 25, 26, 26]
  Tasks with Critical Overrun Points: 60

Configuration: 100 tasks, utilization = 0.75 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 3.976
  Maximum Response Time: 10.519
  Core Load Distribution: [24, 26, 25, 25]
  Tasks with Critical Overrun Points: 62

Configuration: 200 tasks, utilization = 0.25 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 1.198
  Maximum Response Time: 4.469
  Core Load Distribution: [50, 49, 51, 50]
  Tasks with Critical Overrun Points: 160
```

```
Configuration: 400 tasks, utilization = 0.75 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 3.612
  Maximum Response Time: 10.624
  Core Load Distribution: [99, 100, 101, 100]
  Tasks with Critical Overrun Points: 361

Configuration: 500 tasks, utilization = 0.25 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 1.256
  Maximum Response Time: 3.548
  Core Load Distribution: [125, 125, 125, 125]
  Tasks with Critical Overrun Points: 461

Configuration: 500 tasks, utilization = 0.50 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 2.387
  Maximum Response Time: 7.236
  Core Load Distribution: [124, 125, 125, 126]
  Tasks with Critical Overrun Points: 461

Configuration: 500 tasks, utilization = 0.75 (Partitioned Scheduling)
  Schedulable: True
  Average Response Time: 3.978
  Maximum Response Time: 11.375
  Core Load Distribution: [125, 125, 125, 125]
  Tasks with Critical Overrun Points: 460
```