به نام خدا

گزارش پروژه درس سیستم‌های بی‌درنگ

استاد درس:

دکتر سپیده صفری

دستیار آموزشی درس:

خانم جعفری

نویسندگان:

دنیا روشن‌ضمیر 99170467

ابولفضل سلطانی 99109217

خرداد 1402

فهرست مطالب

مقدمه

در این پروژه، هدف پیاده‌سازی الگوریتم زمان‌بندی Critically EDF و مدیریت منابع Ceiling Protocol است. این ترکیب به گونه‌ای طراحی شده است که وظایف با حساسیت بالا[1] و وظایف با حساسیت پایین[2] به صورت همزمان مدیریت شوند و سیستم قادر به پاسخگویی به تغییرات و اولویت‌بندی‌های مختلف باشد.

الگوریتم Criticality EDF وظایف را بر اساس زمان‌های مطلق (absolute times) و میزان حساسیت (criticality levels) آن‌ها زمان‌بندی می‌کند و در مواقع بحرانی، اولویت وظایف HC را نسبت به LC افزایش می‌دهد. از طرف دیگر، در الگوریتم تخصیص منابع تضمین می‌کند که زمان‌بندی وظایف بر اساس پروتکل‌های زمانی دقیق صورت گیرد و تداخل بین وظایف به حداقل برسد. این پروژه با استفاده از ترکیب این دو روش، یک سیستم زمان‌بندی بهینه و کارآمد برای مدیریت وظایف با حساسیت‌های مختلف ارائه می‌دهد.

در نهایت، این پروژه با استفاده از شبیه‌سازی‌ها و تحلیل‌های مختلف، کارایی و اثربخشی الگوریتم پیشنهادی را مورد بررسی قرار می‌دهد و نمودارها و گزارش‌هایی از عملکرد سیستم در شرایط مختلف ارائه می‌کند. این گزارش‌ها شامل بررسی وضعیت‌های مختلف زمانی و حساسیت وظایف، تأثیر پارامترهای مختلف بر عملکرد سیستم، و تحلیل پوشش زمانی و سرعت‌بخشی وظایف می‌باشد.

[1] High Critical

[2] Low Critical

تولید وظایف

در این بخش، وظایف به صورت تصادفی ایجاد می‌شوند و در قالب گراف مدل‌سازی می‌شوند. هر وظیفه دارای گره‌هایی است که نشان‌دهنده بخش‌هایی از وظیفه است که باید اجرا شوند و یال‌هایی که وابستگی بین این گره‌ها را نشان می‌دهند. برای تولید گره‌ها و وابستگی میان آن‌ها از الگوریتم FFT استفاده کرده‌ایم که جزئیات آن در زیر آورده شده است.

منابع

در سیستم تعدادی منبع واحد وجود دارد. هر منبع تنها یک شناسه دارد تا از دیگر منابع متمایز شود.

قطعه کد ۱

```
@dataclass
class Resource:
   id: str

   def map_to_color(self) -> str:
       if self.id == "R1":
           return 'red'
       if self.id == "R2":
           return 'blue'
       if self.id == "R3":
           return 'green'
       if self.id == "R4":
           return 'yellow'
       if self.id == "R5":
           return 'purple'
       return 'black'
```

- متد `map_to_color`: این متد برای کشیدن نمودارهای زمان‌بندی احتیاج می‌شود. به هر منبع یک رنگ اختصاص داده شده تا در نمودارهای زمان‌بندی راحت‌تر بتوان آن‌ها را مشاهده کرد.

زیروظیفه‌ها

هر وظیفه به تعدادی زیروظیفه شکسته شده است. میان برخی از زیروظیفه‌ها رابطه‌ی وابستگی وجود دارد، به این معنی که یک زیروظیفه باید قبل از زیروظیفه دیگر اجرا شود. این رابطه‌ها را با یک گراف به نام *گراف وظایف* نام‌گذاری می‌کنیم.

هر زیروظیفه می‌تواند بخش بحرانی باشد. در صورتی که یک زیروظیفه، بخش بحرانی از آن وظیفه باشد، به تعدادی منبع نیاز خواهد داشت.

- هر وظیفه به تعدادی منبع به صورت non-nested دسترسی خواهد داشت. بازه‌های زمانی در دو لیست `critical_st` و `critical_en` ذخیره می‌شوند.

هر زیروظیفه یک مدت زمان اجرای `wcet_lo` دارد. این مقدار برابر با بدترین زمان اجرای مورد انتظار برای این زیروظیفه است. تمامی زیروظایف یک وظیفه‌ی حساسیت بالا دارای مقدار `wcet_hi` نیز هستند، که در صورتی که وظیفه در زمان `wcet_lo` به اتمام نرسید، سیستم وارد over-run می‌شود.

هر زیروظیفه یک شناسه نیز دارد که با آن از دیگر زیروظایف متمایز می‌شود.

قطعه کد ۲: دیتاکلس یک زیروظیفه

```
@dataclass
class Node:
   id: str
   wcet_hi: int
   wcet_lo: int
   critical_st: list[int]
   critical_en: list[int]
   resources: list[Resource] = None

   def needed_resource(self, time: int) -> int:
       for i in range(len(self.resources)):
           if self.critical_st[i] <= time and self.critical_en[i] > time:
               return self.resources[i]
       return None

   def needed_resource_ends_at(self, time: int) -> bool:
       for i in self.critical_en:
           if i == time:
               return True
       return False

   def get_exec_time(self, overrun: bool) -> int:
       return self.wcet_hi if overrun else self.wcet_lo

   def __str__(self) -> str:
       return f"Node: {self.id}, WCET_HI: {self.wcet_hi}, WCET_LO: {self.wcet_lo},
Resource: {self.resources}"
```

- متد `get_exec_time` زمان مورد نیاز برای اجرای این زیروظیفه را با توجه به over-run بودن سیستم یا نبودن آن خروجی می‌دهد.
- متد `needed_resource`: یک زمان relative دریافت می‌کند و در صورت وجود، منبعی که در این زمان به آن احتیاج دارد را برمی‌گرداند.
- متد `needed_resource_ends_at`: یک زمان relative دریافت می‌کند و در صورتی که در این زمان دیگر به منبع احتیاج نداشته باشد، مقدار True را برمی‌گرداند.

با استفاده از تابع `generate_resources` که ورودی آن تعداد منابع سیستم است، می‌توان منابع را ساخت. در مثال‌های مورد استفاده ما همیشه تعدادی تصادفی بین ۱ تا ۵ منبع ایجاد می‌شود.

گراف وظایف

الگوریتم FFT (Fast Fourier Transform) یک خانوده از گراف را می‌سازد. در این پروژه نیز برای تولید گراف وظایف از این الگوریتم استفاده می‌شود. این گراف وظایف، ساختاری است که در آن هر گره نشان‌دهنده یک بخش از وظیفه و هر یال نشان‌دهنده وابستگی زمانی بین بخش‌های مختلف وظیفه است. در ادامه، نحوه عملکرد این الگوریتم توضیح داده می‌شود.

این الگوریتم از یک پارامتر `mlg` استفاده می‌کند. این پارامتر نشان دهنده عمق گراف حاصل می‌باشد. ابتدا یک درخت دودویی با ارتفاع `mlg` ساخته می‌شود، سپس `mlg` لایه دیگر راس اضافه می‌شود و در بین هر دولایه به شکلی خاص یال وجود دارد.

قطعه کد ۳: ایجاد گراف وظایف با استفاده از الگوریتم FFT

```python
def FFT(mlg: int) -> tuple[list[int], list[tuple[int, int]]]:
    edges: list[tuple[int, int]] = []
    for i in range(2, 2 ** (mlg + 1)):
        edges.append((i // 2, i))

    label = 2**mlg
    for j in range(mlg):
        for k in range(2 ** (mlg)):
            edges.append((label + k, (2**mlg) + label + k))
            edges.append((label + k, ((label + k) ^ (2**j)) + (2**mlg)))
        label += 2**mlg
    nodes = [i + 1 for i in range(2 ** (mlg + 1) - 1 + mlg * (2 ** mlg))]
    return nodes, edges
```

در قطعه کد ۱ ابتدا یال‌های درخت دودویی ایجاد می‌شوند، سپس دقیقا `mlg` لایه جدید، و در هر لایه 2^{mlg} راس جدید اضافه می‌شود و با رابطه‌ی توضیح داده شده به لایه قبلی متصل می‌شوند. در نهایت این قطعه کد لیست رئوس و لیست یال‌ها را برمی‌گرداند. برای مثال دو گراف زیر خروجی قطعه کد ۱ به ازای $mlg = 2$ و $mlg = 1$ هستند.

مسیر بحرانی گراف - تابع `get_critical_path`

این تابع برای پیدا کردن مسیر بحرانی یا بلندترین مسیر وزن‌دار در گراف‌های جهت‌دار بدون‌دور[3] پیاده‌سازی شده است. ورودی‌های آن لیستی از رئوس و لیستی از یال‌های متصل به آن‌ها است.

- توضیح الگوریتم: هر گراف جهت‌دار بدون‌دور را می‌توان در یک خط نوشت به طوری که جهت همه یال‌ها رو به جلو باشد. با استفاده از این ترتیب و برنامه‌ریزی پویا[4] بلندترین مسیر مختوم به هر راس از گراف محاسبه می‌شود؛ در نهایت بیشترین این مقدارها برگردانده می‌شود.

قطعه کد ۴: پیدا کردن بلندترین مسیر - مسیر بحرانی

```
def get_critical_path(nodes: list[Node], edges: list[Edge]) -> list[Node]:
    dp: dict[str, int] = {}
    degree_in: dict[str, int] = {}
    for edge in edges:
        degree_in[edge.sink.id] = degree_in.get(edge.sink.id, 0) + 1
    sources = [node for node in nodes if degree_in.get(node.id, 0) == 0]
    for node in sources:
        dp[node.id] = node.wcet_hi
    while sources:
        node = sources.pop()
        for edge in edges:
            if edge.src.id == node.id:
                degree_in[edge.sink.id] -= 1
                dp[edge.sink.id] = max(dp.get(edge.sink.id, 0), dp[node.id] +
edge.sink.wcet_hi)
                if degree_in.get(edge.sink.id, 0) == 0:
                    sources.append(edge.sink)
    return max(dp.values())
```

[3] Directed Acycling Graphs

[4] Dynamic Programming

ایجاد وظیفه - تابع `generate_task`

این تابع برای تولید یک وظیفه استفاده می‌شود. در این تابع، ابتدا با استفاده از الگوریتم FFT یک گراف شامل زیروظیفه‌ها و ارتباط میان آن‌ها ساخته می‌شود. سپس برای هر زیروظیفه ابتدا عددی تصادفی بین ۵ تا ۱۰ برای مقدار `wcet_hi` آن انتخاب می‌شود. اگر وظیفه از نوع HC باشد، مقدار `wcet_lo` برابر با ۰.۶ برابر `wcet_hi` است، و در غیر این صورت مقداری برابر دارند.

برای مقداردهی کردن دوره تناوب وظیفه، ابتدا مسیر بحرانی در گراف زیروظیفه‌ها محاسبه می‌شود و در متغیر `critical_path` ریخته می‌شود. سپس کوچک‌ترین 2^x بزرگ‌تر از آن محاسبه می‌شود. مقدار دوره تناوب وظیفه به صورتی تصادفی از بین دو عدد 2^x و 2^{x+1} انتخاب می‌شود. همچنین `wcet` کل وظیفه برابر با مجموع `wcet_hi` زیروظیفه‌ها انتخاب می‌شود. در این پروژه، مقدار ددلاین نسبی هر نمونه از آن برابر با مقدار دوره تناوب است و نیازی به انتخاب کردن آن نیست.

سپس برای وظیفه بین ۱ تا ۱۰ بخش بحرانی به صورت تصادفی در نظر گرفته می‌شود. هر بخش بحرانی دقیقا یک زیروظیفه است و تنها به یک منبع نیاز پیدا می‌کند.

قطعه کد ۵: تابع تولید یک وظیفه

```
def generate_task(task_id: int, task_type: TaskType, resources: list[Resource]) ->
Task:
    graph_nodes, graph_edges = FFT(1)
    nodes: list[Node] = []

    critical_nodes_count = rand.randint(1, min(10, len(graph_nodes)))
    critical_nodes = rand.sample(graph_nodes, k=critical_nodes_count)

    for node_id in graph_nodes:
        wcet_hi = rand.randint(5, 10)
        wcet_lo = wcet_hi if task_type == TaskType.LC else int(0.6 * wcet_hi)

        node_resources = []
        critical_st = []
        critical_en = []

        if node_id in critical_nodes:
            m = rand.randint(1, wcet_lo // 2)
            node_resources = rand.choices(resources, k=m)
            random_list = rand.sample(range(0, wcet_lo + 1), 2 * m)
            random_list = sorted(random_list)
```

```
            critical_st = [i for idx, i in enumerate(random_list) if idx % 2 == 0]
            critical_en = [i for idx, i in enumerate(random_list) if idx % 2 == 1]
        nodes.append(Node(id=f"T{task_id}-J{node_id}", wcet_hi=wcet_hi,
wcet_lo=wcet_lo, resources=node_resources, critical_st=critical_st,
critical_en=critical_en))

    edges: list[Edge] = []
    for edge in graph_edges:
        src = nodes[edge[0] - 1]
        sink = nodes[edge[1] - 1]
        edges.append(Edge(src=src, sink=sink))

    critical_path = get_critical_path(nodes, edges)
    x = 1
    while x < critical_path:
        x *= 2

    period = rand.choice([x, 2 * x])
    wcet = sum([node.wcet_hi for node in nodes])

    return Task(id=f"T{task_id}", period=period, wcet=wcet, nodes=nodes, edges=edges,
task_type=task_type)
```

ایجاد مجموعه وظایف - تابع `generate_tasks`

این تابع برای تولید یک مجموعه از وظایف بر اساس پارامترهای مختلف استفاده می‌شود. وظایف تولید شده به گونه‌ای تنظیم می‌شوند که نسبت وظایف HC به LC و حد بالای استفاده از منابع رعایت شود. پارامترهای آن به صورت زیر است:

- `resources`: لیست منابع مورد نیاز برای مجموعه وظایف.
- `task_count`: تعداد وظیفه‌هایی که باید ایجاد شود.
- `ratio`: نسبت تعداد وظایف HC به LC
- `utilization_ub`: حداکثر مقدار مجموع بهره‌وری[5] وظیفه‌ها.

قطعه کد ۶: تابع تولید مجموعه وظایف

```
def generate_tasks(resources: list[Resource], task_count: int, ratio: float = 0.5,
```

[5] Utilization

```
utilization_ub: int = 1) -> list[Task]:
    tasks = []
    for i in range(task_count):
        task_type = TaskType.HC if i < ratio * task_count else TaskType.LC
        tasks.append(generate_task(i, task_type, resources))

    # remove tasks with utilization > 1.0
    trash_tasks = [task for task in tasks if task.utilization() > 1.0]
    for trash_task in trash_tasks:
        tasks.remove(trash_task)

    while sum([task.utilization() for task in tasks]) > utilization_ub:
        trash_task = rand.choice(tasks)
        tasks.remove(trash_task)

    return tasks
```

توضیحات تکمیلی: ابتدا با استفاده از تابع `generate_task` که توضیح داده شد، وظایف تولید می‌شوند. سپس وظایفی که بهره‌وری آن‌ها بزرگتر از یک است، حذف می‌شوند. این اتفاق مطابق با توضیحات پروژه است. سپس تا زمانی که مجموع بهره‌وری وظایف از حد بالای تعریف شده برای آن بیش‌تر است، به صورت تصادفی یکی از آن‌ها حذف می‌شوند.

محاسبه بهره‌وری: برای هر وظیفه، بهره‌وری با استفاده از فرمول زیر محاسبه می‌شود:

$$U = \frac{WCET}{PERIOD}$$

به کلاس داده[6] `Task` بپردازیم. هر نمونه از این کلاس یک وظیفه است. هر وظیفه یک شناسه، دوره تناوب، بدترین زمان اجرا، زیروظیفه‌ها و یال‌های وابستگی میان زیروظیفه‌ها و در نهایت نوع حساسیت این وظیفه را داراست.

قطعه کد ۷: دیتاکلس یک وظیفه

```
@dataclass
class Task:
    id: int
```

[6] Dataclass

```
    period: int
    wcet : int
    nodes: list[Node]
    edges: list[Edge]
    task_type: TaskType

    def get_wcet(self) -> int:
        return sum([node.wcet_hi for node in self.nodes])

    def utilization(self) -> float:
        return self.get_wcet() / self.period

    def do_need_resource(self, resource: Resource) -> bool:
        return any([node.resource == resource for node in self.nodes])

    def nearest_deadline(self, time: int) -> int:
        return self.period - (time % self.period)
```

- متد `get_wcet`: این متد مجموع `wcet_hi` زیروظیفه‌ها را برمی‌گرداند.
- متد `utilization`: این متد بهره‌وری کل وظیفه را که توضیح داده شد، برمی‌گرداند.
- متد `do_need_resource`: این متد بررسی می‌کند که یک وظیفه به یک منبع احتیاج دارد یا نه. پیاده‌سازی آن با استفاده از بررسی تمامی زیروظیفه‌ها انجام شده است.
- متد `nearest_deadline`: این متد نیز اولین ددلاین نسبی یکی از نمونه‌های این وظیفه بعد از زمان `time` را برمی‌گرداند. برای محاسبه سقف منبع به این متد نیاز می‌شود.

زمان‌بندی با الگوریتم Critically EDF

در پیاده‌سازی، یک کلاس `CriticallyEDF` وجود دارد. برای نمونه‌گیری از این کلاس، باید لیست وظایف، لیست منابع، ضریب سرعت‌بخشی[7]، توضیحات بیشتر برای لاگ بیشتر و شانس over-run در انجام یک وظیفه HC داده شود.

در هنگام ساختن یک نمونه از کلاس، مقدار ضریب سرعت‌بخشی هر چقدر باشد، زمان مورد نیاز برای اجرای وظایف آپدیت می‌شوند. متغیرهای پرکاربردی که در این کلاس مقداردهی می‌شوند:

- `current_time`: لحظه کنونی در هنگام زمان‌بندی کردن وظایف.

[7] Speedup Factor

- `allocated_by`: دیکشنری از منابع به وظایف. اگر یک منبع توسط یک وظیفه در حال استفاده باشد، از این دیکشنری می‌توان آن وظیفه را پیدا کرد.
- `execution_time`: دیکشنری از کارها به مدت زمانی که تابه‌حال آن کار اجرا شده است.
- `jobs`: لیست نمونه‌هایی[8] از وظایف که شروع آن‌ها رسیده و هنوز اجرا نشده‌اند.
- `done_jobs`: لیست نمونه‌هایی از وظایف که به طور کامل اجرا شده‌اند.
- `in_degree`: برای پیدا کردن زیروظایفی که وابستگی ندارند، استفاده می‌شود.
- `hyperperiod`: مقدار هایپرپریود وظایف

پروتکل سقف[9]

مطابق با تعریف پروژه، برای هر منبع R_r سقف آن برابر است با:

$$\psi_r(t) = t + \psi_{r,i} \; if \; R_r \; is \; currently \; held \; by \; \tau_i \; at \; t \; \infty \; otherwise$$

برای هر جفت وظیفه و منبع مرتبط با آن $\psi_{r,i}$ برابر است با کم‌ترین relative deadline وظایفی که به آن منبع دسترسی دارند. توجه داشته باشید که این محاسبه شامل خود وظیفه τ_i نمی‌شود.

سقف سیستم در زمان t برابر با کمینه مقدار سقف منابع است؛

$$\gamma(t) = \min_r \psi_r(t)$$

متد `psi` و `resource_ceiling` و `system_ceiling` در کلاس `CriticallyEDF` به صورت زیر پیاده‌سازی شده است:

قطعه کد ۸: پیدا کردن ψ یک منبع و وظیفه

```
def psi(self, resource: Resource, task: Task) -> int:
    result = math.inf
    for i, job in enumerate(self.jobs):
        if job.task == task:
            continue
        if job.task.do_need_resource(resource):
            result = min(result, job.task.nearest_deadline(self.current_time))
    return result
```

[8] Instances

[9] Ceiling

قطعه کد ۹: متد محاسبه سقف منبع

```
def __resource_ceiling(self, resource: Resource) -> int:
    if not self.allocated_by.get(resource.id, None):
        return math.inf
    return self.current_time + self.psi(resource, self.allocated_by[resource.id])
```

قطعه کد ۱۰: متد محاسبه سقف سیستم

```
def __system_ceiling(self) -> int:
    result = math.inf
    for resource in self.resources:
        result = min(result, self.__resource_ceiling(resource))
    return result
```

تعریف Πr(t):

در هر زمان t ، برای هر منبع R_r، یک request deadline وجود دارد که به صورت زیر محاسبه می‌شود:

- اگر منبع R_r در حال حاضر در حال استفاده باشد، Πr(t) برابر با زودترین deadline وظیفه‌ای است که در حال استفاده از منبعR_r است.
- اگر منبع R_r آزاد باشد، Πr(t) برابر با بی‌نهایت است.

الگوریتم زمان‌بندی Criticality EDF

متد `schedule` زمان‌بندی را از زمان صفر تا هایپرپریود اجرا می‌کند. برای بهبود نمودارها، در صورتی که از هایپرپریود عبور کنیم و وظیفه‌ای ناتمام مانده باشد، تا زمان پایان آن‌ها پیش می‌رویم.
ابتدا وظایفی که نمونه‌ای از آن‌ها در زمان `current_time` به سیستم اضافه می‌شود، مدیریت می‌شود. این کار با استفاده از متد `create_periodic_jobs` پیاده‌سازی می‌شود.

قطعه کد ۱۱: متد ایجاد یک نمونه از یک وظیفه پریودیک در زمان کنونی

```
def __create_periodic_jobs(self):
   for task in self.tasks:
       if self.current_time % task.period == 0:
           if task.task_type == TaskType.LC and self.overrun:
               print(f"Overrun - Task {task.id} is dropped")
               continue
           do_overrun = False if task.task_type == TaskType.LC or rand.randint(0,
100) >= self.overrun_chance else True
           job = Job(id=self.counter, task=task, arrival=self.current_time,
deadline=self.current_time + task.period, active=False, overrun=do_overrun)
           self.jobs.append(job)
           self.counter += 1
           for node in task.nodes:
               self.execution_time[f"{node.id}-{job.id}"] = 0
           for edge in task.edges:
               self.in_degree[f"{edge.sink.id}-{job.id}"] =
self.in_degree.get(f"{edge.sink.id}-{job.id}", 0) + 1
```

- با حرکت روی تمامی وظیفه‌ها، وظایفی که باقی‌مانده تقسیم زمان بر دوره تناوب آن‌ها صفر باشد، در این لحظه نمونه‌ای به سیستم اضافه می‌کنند.
- اگر وظیفه از نوع HC باشد، با احتمال `overrun_chance` که در هنگام ساخت نمونه زمان‌بندی تعیین شده است، دچار overrun می‌شود. دقت کنید در بخش اول پروژه این احتمال ۰ و در بخش دوم این احتمال ۰.۳ است.
- سپس یک کار ایجاد می‌شود و به لیست کارها اضافه می‌شود. همچنین دو متغیر `execution_time` و `in_degree` که برای نمونه‌های مختلف یک وظیفه یکسان هستند، مقداردهی اولیه می‌شوند.

سپس اگر کاری در سیستم نباشد، سیستم در حالت idle خواهد بود، و یک واحد زمانی بدون اجرای وظیفه‌ای جلو می‌رود. در غیر این صورت، به دنبال وظیفه‌ی Critically EDF خواهیم بود. کارهای

موجود را بر اساس حساسیت آن‌ها و سپس ددلاینشان مرتب می‌شوند. در این ترتیب، اولین وظیفه‌ای که از ددلاین آن از سقف سیستم کمتر باشد، یا خودش active باشد انتخاب می‌شود. در صورتی که هیچ وظیفه‌ی active وجود نداشته باشد، یعنی هیچ وظیفه‌ای هنوز شروع نشده؛ پس سقف سیستم برابر بی‌نهایت خواهد بود و وظیفه‌ی اول در ترتیب انتخاب می‌شود.

قطعه کد ۱۲: پیدا کردن وظیفه با بیشترین اولویت در سیستم

```
job = None
self.jobs.sort(key=lambda job: (job.task.task_type.value, job.deadline))
for j in self.jobs:
   if j.deadline < self.__system_ceiling() or j.active:
       job = j
       break
```

سپس با استفاده از متد `execute_job` یک واحد زمانی این وظیفه اجرا می‌شود. در این متد، ابتدا زیروظیفه‌ای که وابستگی به زیروظیفه‌ی دیگری ندارد انتخاب می‌شود. اگر این زیروظیفه نیاز به منبعی دارد، به آن اختصاص داده می‌شود. سپس مقدار زمان اجرا شده آن یکی بیش‌تر می‌شود. در صورتی که اجرای آن تمام شده باشد، منبعی که اختیار کرده (در صورت وجود) آزاد می‌شود و وابستگی زیروظیفه‌های دیگر به این زیروظیفه از بین می‌رود. اگر ددلاین وظیفه miss شده باشد، در صورتی که وظیفه HC باشد، ارور زمان‌ناپذیری خواهد داد، در غیر صورت تنها از کیفیت سرویس کم می‌شود. در صورتی که تمامی زیروظیفه‌های این وظیفه تمام شده باشند، خروجی متد `True` است تا در متد اصلی آن را از لیست کارها حذف کنیم.

قطعه کد ۱۳: اجرا کردن یک واحد زمانی

```
def __execute_job(self, job: Job) -> bool:
   job.active = True
   selected_node = self.__get_not_dependent_job(job=job)
   if selected_node != None and self.verbose:
       print(f"Current Time: {self.current_time}, Executing Job: {job.id}, Task:
{job.task.id}, Node: {selected_node.id}")
   assert selected_node != None

   needed_resource =
selected_node.needed_resource(self.execution_time.get(f"{selected_node.id}-{job.id}"
, 0))
   assert not needed_resource or self.allocated_by.get(needed_resource.id, None) in
```

```
[None, job.task]

   if needed_resource:
       self.allocated_by[needed_resource.id] = job.task
   self.execution_time[f"{selected_node.id}-{job.id}"] =
self.execution_time.get(f"{selected_node.id}-{job.id}", 0) + 1

   color = 'gray'
   if needed_resource:
       color = needed_resource.map_to_color()
   self.ax.broken_barh([(self.current_time, 1)], (self.row[selected_node.id], 0.5),
facecolors=color)

   if needed_resource and
selected_node.needed_resource_ends_at(self.execution_time.get(f"{selected_node.id}-{
job.id}", 0)):
       self.allocated_by[needed_resource.id] = None

   if self.execution_time[f"{selected_node.id}-{job.id}"] ==
selected_node.get_exec_time(job.overrun):
       if self.verbose:
           print(f"Node {selected_node.id} is Done")
       for edge in job.task.edges:
           if edge.src.id == selected_node.id:
               self.in_degree[f"{edge.sink.id}-{job.id}"] -= 1
   elif self.execution_time[f"{selected_node.id}-{job.id}"] > selected_node.wcet_lo:
       self.enable_overrun()
   if self.current_time > job.deadline:
       if job.task.task_type == TaskType.HC:
           raise Exception("Not Schedulable - Deadline Missed")
       job.quality = 100 - (self.current_time - job.deadline)
   for node in job.task.nodes:
       if self.execution_time.get(f"{node.id}-{job.id}", 0) <
node.get_exec_time(job.overrun):
           return False
   return True
```

توضیح مفهوم Speedup Factor

در سیستم‌های زمان واقعی، یکی از راه‌های بهبود زمانبندی وظایف افزایش سرعت اجرای وظایف است. مفهوم ضریب سرعت‌بخش به معنای افزایش سرعت اجرای وظایف برای بهبود زمان‌بندی و جلوگیری از شکست در رعایت مهلت‌های زمانی است. با افزایش مقدار Speedup Factor، زمان اجرای وظایف کاهش می‌یابد و ممکن است مجموعه وظایف زمانبندی‌پذیر شود.

توضیح Overrun

اگر یک وظیفه HC در زمان wcet_lo به پایان نرسد، سیستم وارد فضای overrun می‌شود. در این حالت ابتدا همه کارهای LC دراپ می‌شوند و تا زمانی که به هایپرپریود نرسیم، هیچ کار جدیدی از LC پذیرفته نمی‌شود و همه کارهای HC با فرض wcet_hi زمان‌بندی می‌شوند.

```
def schedule(self):
   while self.current_time < 2 * self.hyperperiod:
       if self.current_time % self.hyperperiod == 0:
           self.overrun = False

       self.__create_periodic_jobs()
       if not self.jobs:
           self.current_time += 1
           continue

       job = None
       self.jobs.sort(key=lambda job: (job.task.task_type.value, job.deadline))
       for j in self.jobs:
           if j.deadline < self.__system_ceiling() or j.active:
               job = j
               break

       if not job:
           print("Not Schedulable - Impossible")
           return False
       try:
```

```
            if self.__execute_job(job):
                self.jobs.remove(job)
                job.active = False
                self.done_jobs.append(job)
        except Exception as e:
            if self.verbose:
                print(e)
            return False

        self.current_time += 1

    if len(self.jobs) > 0:
        return False
    return True
```

نمودار کیفیت خدمات (QoS)

برای رسم نمودار کیفیت خدمات (QoS) برای حالتی که نسبت وظایف High-Criticality به Low-Criticality برابر یک باشد، مراحل زیر را دنبال می‌کنیم:

```
def quality_of_service(self) -> float:
    return sum([job.quality for job in self.done_jobs]) / len(self.done_jobs)
```

نمودار تعداد ددلاین‌های میس شده

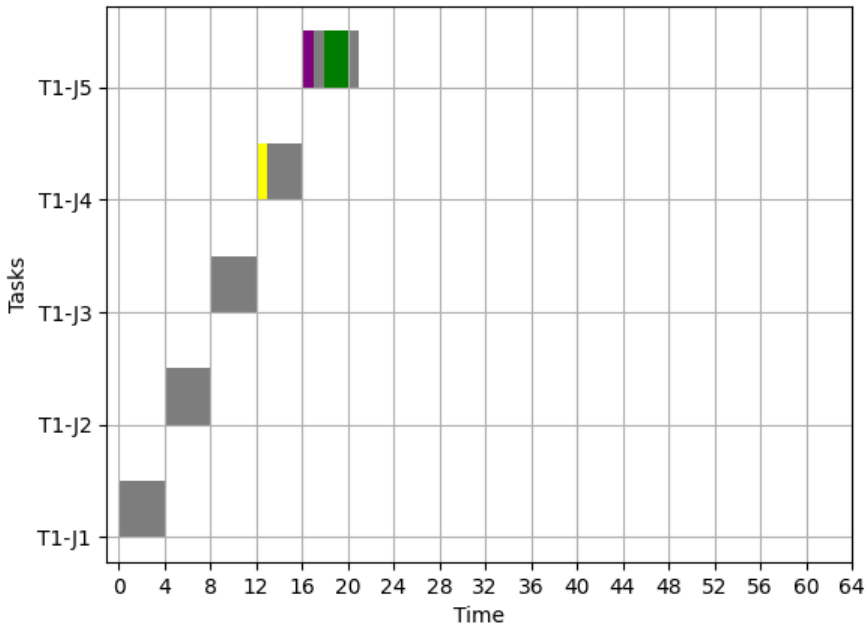
```
def missed_deadline_count(self) -> int:
    count = 0
    for job in self.done_jobs:
```

```
        if job.quality < 100:
            count += 1
    return count
```

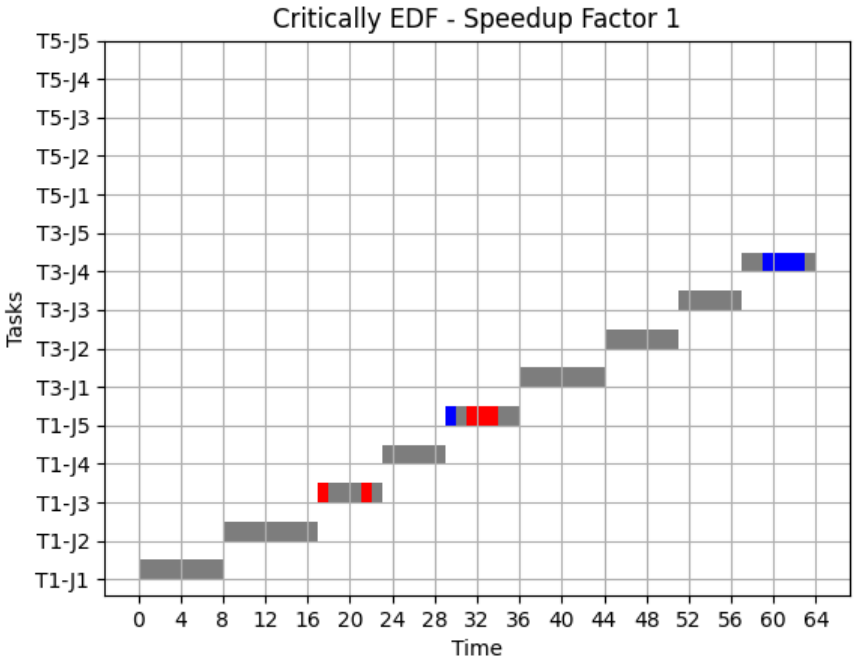
نتایج

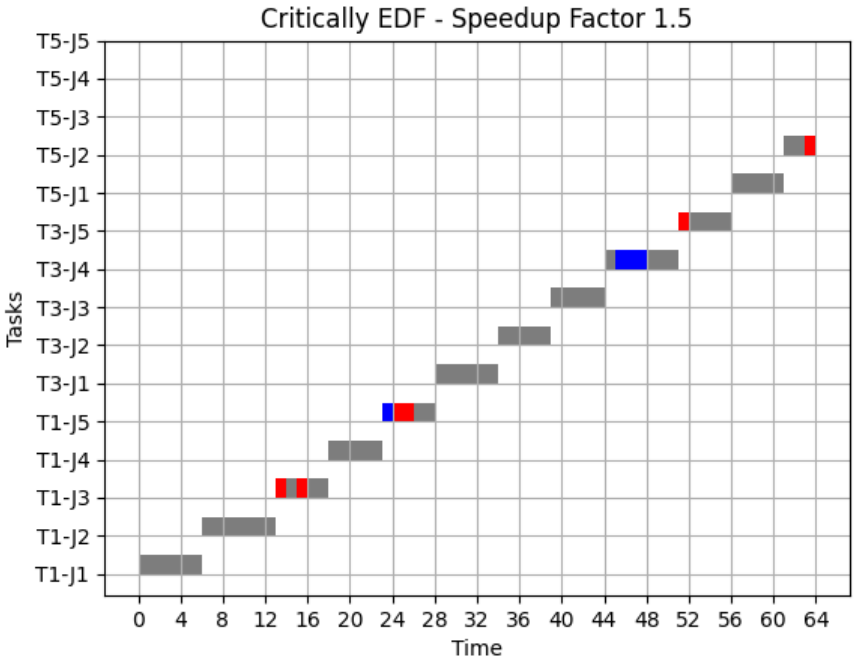
رسم نمودار برای حالتی که همه وظایف در حالت نرمال باشند:

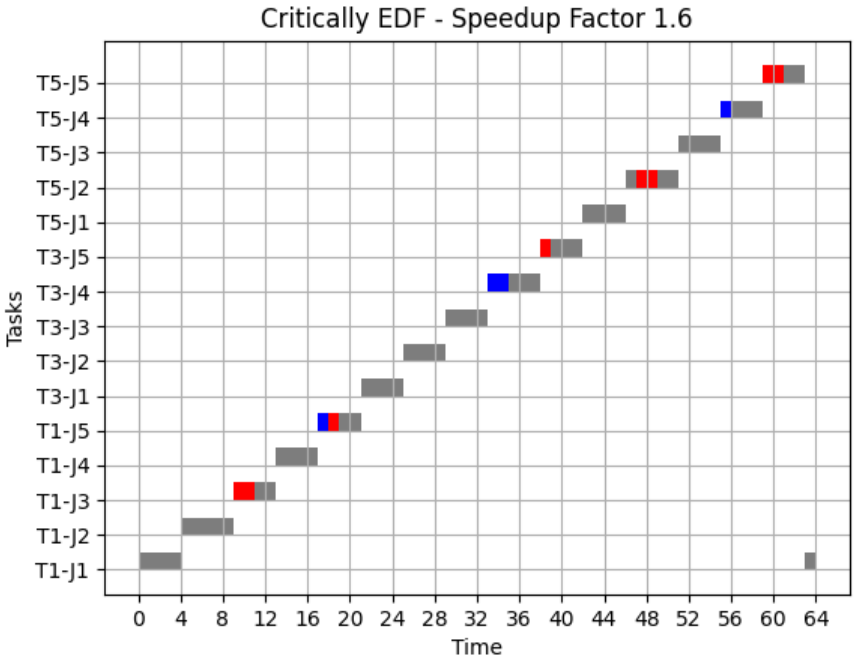
۱.۱. نمودار زمان‌بندپذیری وظایف با speedup factor = 1 برای حالتی که نسبت وظایف HC به LC برابر یک باشد.

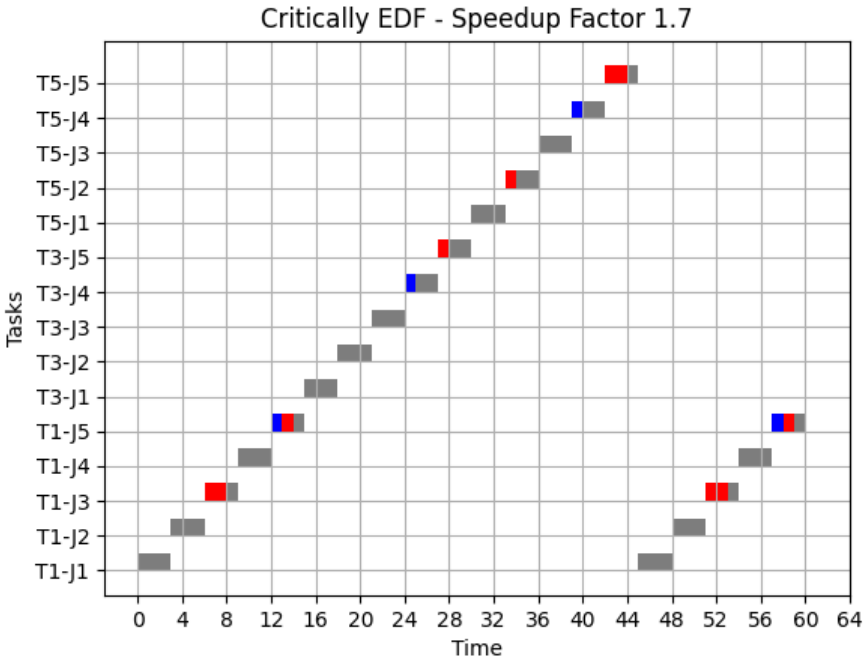


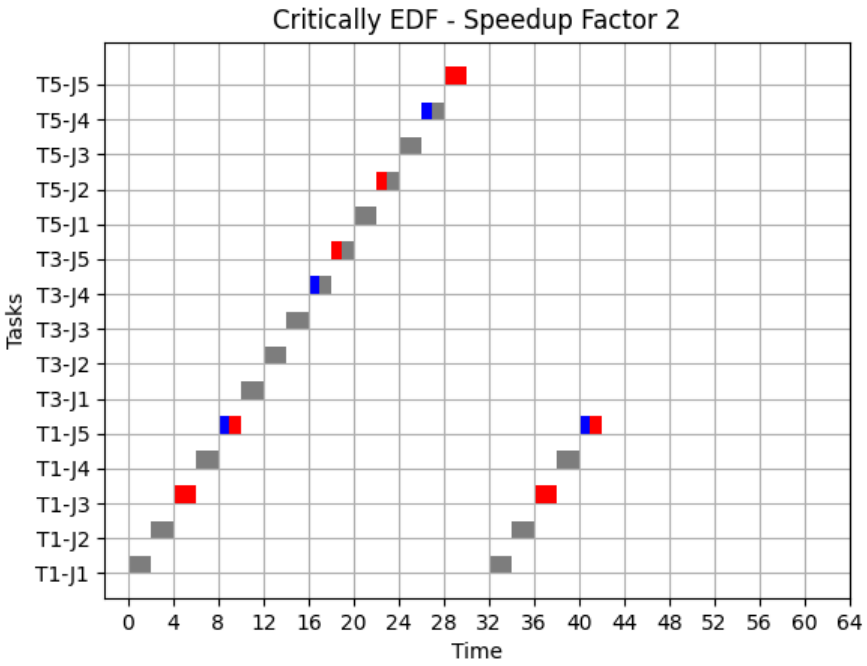
۱.۲. و ۱.۳. نمودار کیفیت خدمات برای حالتی که نسبت وظایف HC به LC برابر یک باشد به ازای speedup factorهای مختلف

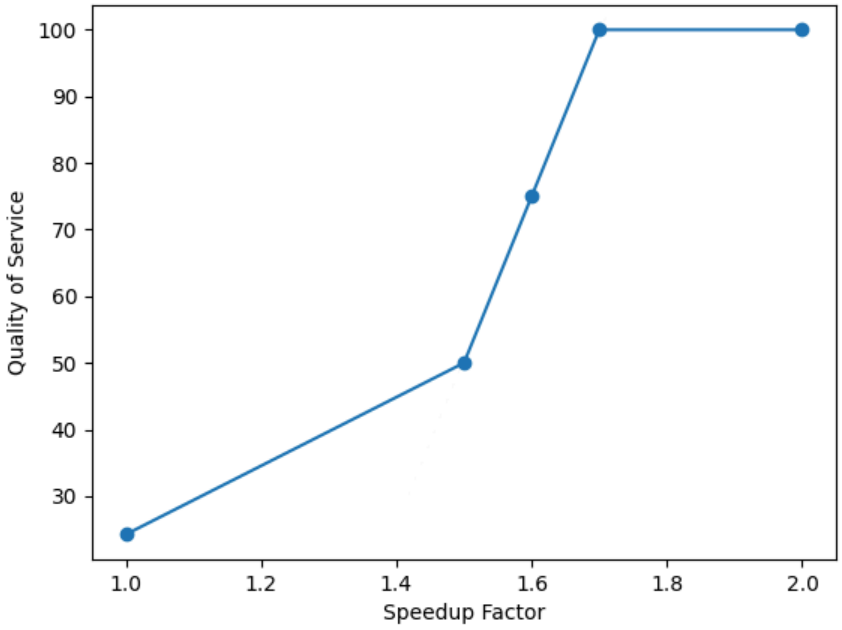


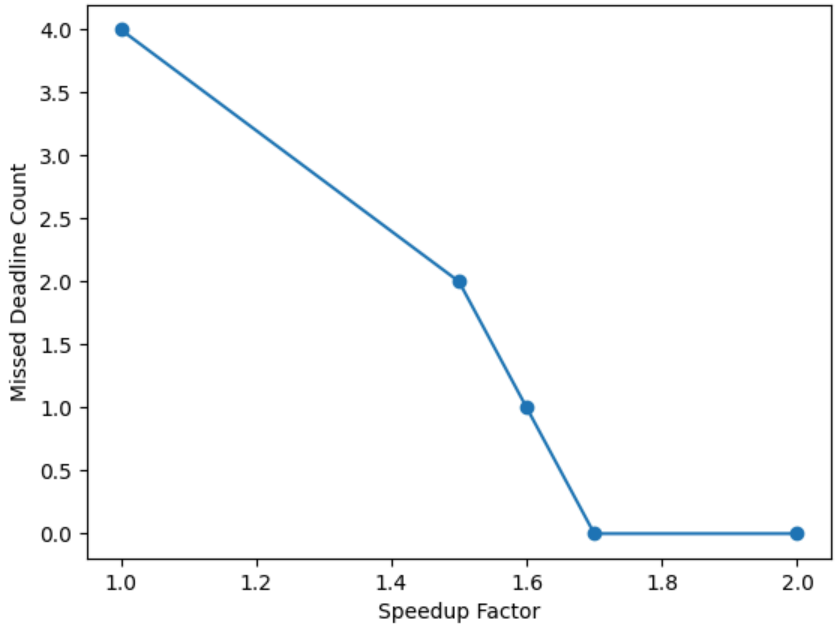



Critically EDF - Speedup Factor 1.6
T5-J5
T5-J4
T5-J3
T5-J2
T5-J1
T3-J5
T3-J4
T3-J3
T3-J2
T3-J1
T1-J5
T1-J4
T1-J3
T1-J2
T1-J1
Tasks
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
Time

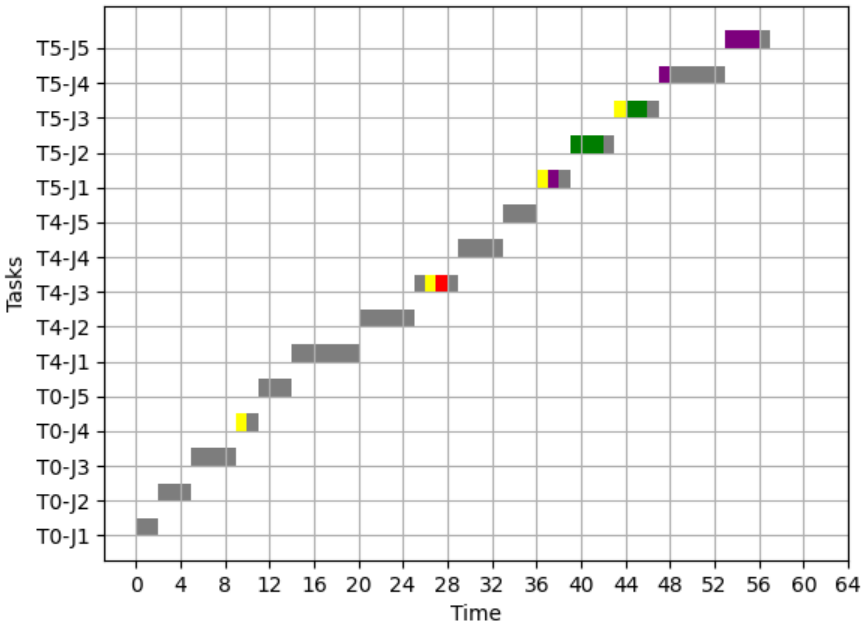

Critically EDF - Speedup Factor 1.7
T5-J5
T5-J4
T5-J3
T5-J2
T5-J1
T3-J5
T3-J4
T3-J3
T3-J2
T3-J1
T1-J5
T1-J4
T1-J3
T1-J2
T1-J1
Tasks
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
Time


Critically EDF - Speedup Factor 2
T5-J5
T5-J4
T5-J3
T5-J2
T5-J1
T3-J5
T3-J4
T3-J3
T3-J2
T3-J1
T1-J5
T1-J4
T1-J3
T1-J2
T1-J1
Tasks
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
Time


100
90
80
70
60
50
40
30
Quality of Service
1.0 1.2 1.4 1.6 1.8 2.0
Speedup Factor

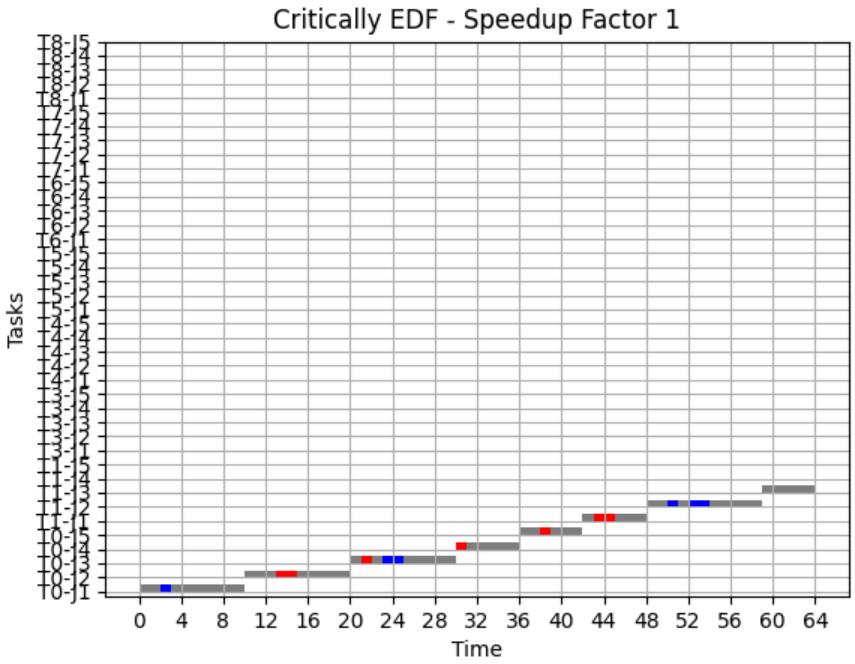


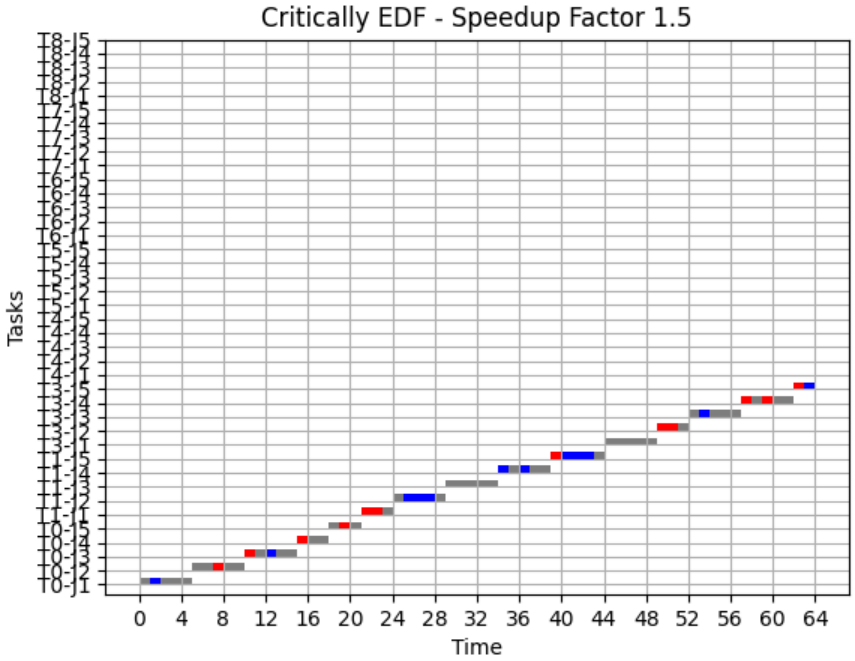
۳.۱ نمودار زمان‌بندپذیری برای حالتی که ۳۰ درصد از وظایف دچار overrun شوند

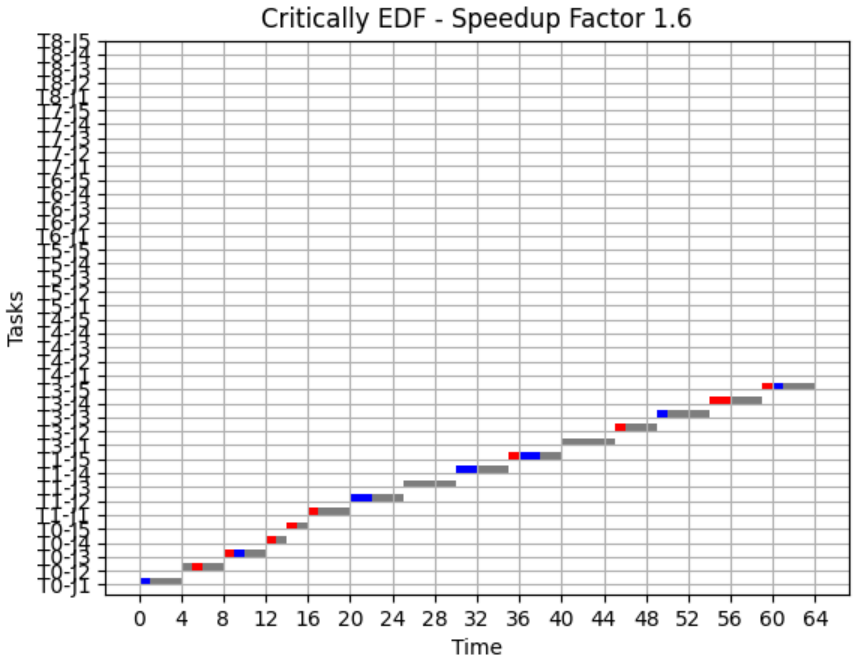


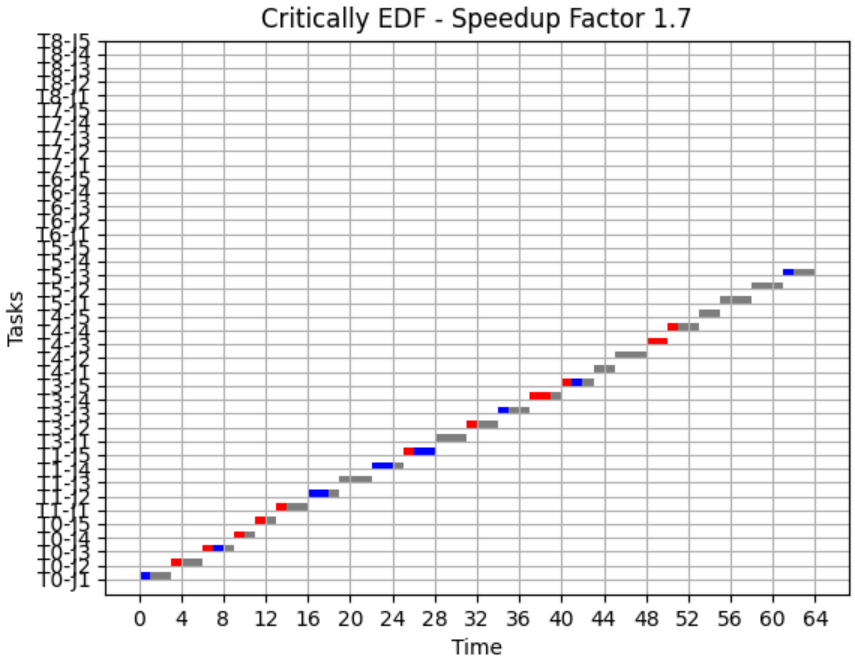
۳.۲. و ۳.۳. نمودار کیفیت خدمات و speedup factorهای مختلف

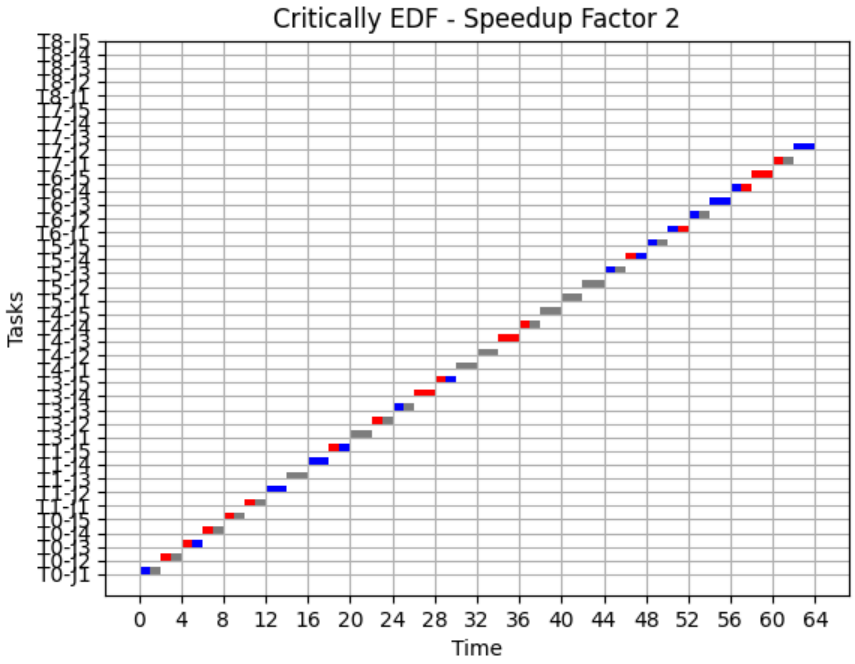
در نمونه‌های زیر ۴ وظیفه T5، T6، T7 و T8 همگی LC هستند و بقیه وظیفه‌ها HC.

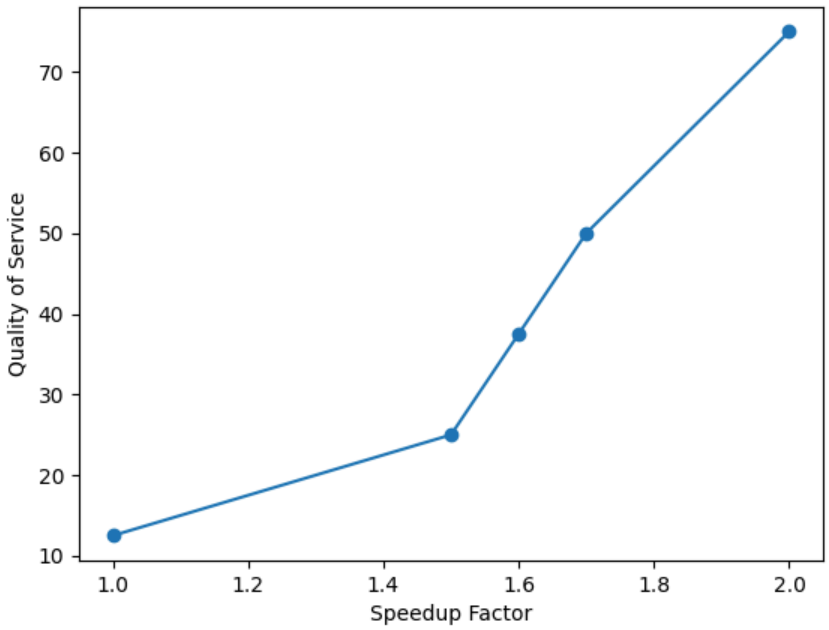


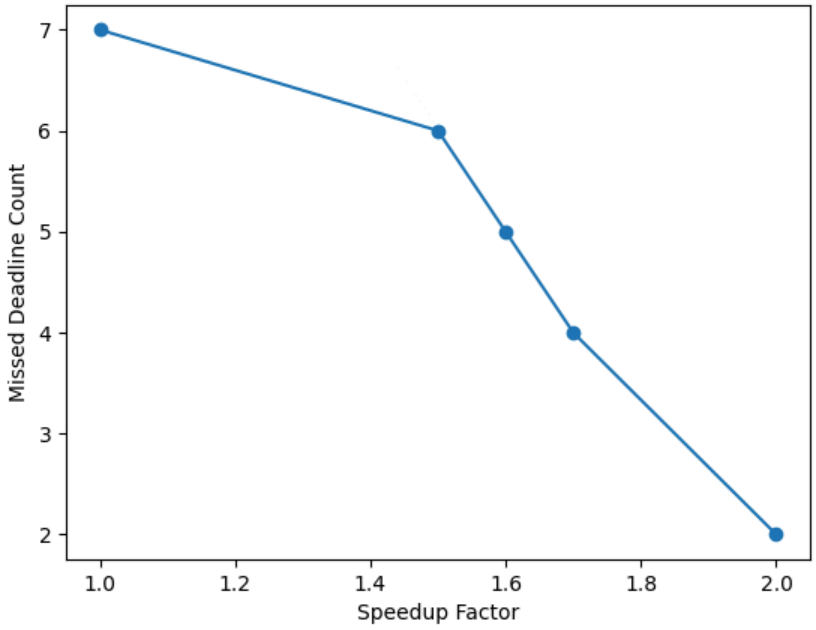



Critically EDF - Speedup Factor 1.6
Tasks
Time
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64


Critically EDF - Speedup Factor 1.7
Tasks
Time
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64


Critically EDF - Speedup Factor 2
Tasks
Time
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64





منابع:

1. https://ieeexplore.ieee.org/document/993206

2.