سامانه‌های بی‌درنگ

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

«پروژه»

نگار نوبختی ۹۸۱۷۱۲۰۱

شهاب حسینی مقدم ۹۸۱۰۵۷۱۶

مفاهیم به کار رفته در پروژه

TMR

این تکنیک یک روش مقابله با اشکال است که در آن، از یک جاب (معمولا جاب‌های با اولویت بالا) ۳ نمونه تولید می‌کنیم و هر سه را اجرا می‌کنیم. در نهایت، اگر نتیجه‌ی تولید شده‌ی یکی از این سه نمونه یکسان نبود، جوابی که دو نمونه تولید کرده‌اند را استفاده می‌کنیم.

WFD

این عبارت که مخفف worst-fit Decreasing است، یکی از اصطلاحات bin-packing است. در این مسئله، ما تعدادی جاب داریم که می‌خواهیم آن‌ها را روی چند هسته‌ی CPU مپ کنیم. البته این مسئله NP-hard است و یک الگوریتم تقریبی برای آن این است که به ترتیب، هر جاب را به هسته‌ای تخصیص دهیم که بیشترین ظرفیت خالی (که جاب در آن جا بشود) را داشته باشد. این الگوریتم همان WFD است.

FFD

این عبارت که مخفف first-fit Decreasing است، یک روش دیگر برای bin-packing است که در آن هر جاب را به اولین هسته‌ای که ظرفیت خالی (که جاب در آن جا بشود) را داشته باشد تخصیص می‌دهیم.

Overrun

در سیستم‌های مختلط-بحرانی، برای جاب‌هایی که از سطح بحرانی بالایی برخوردار هستند، دو نوع WCET داریم. نوع اول $WCET_{LO}$ است که طراح سیستم مقدار آن را مشخص می‌کند و نوع دوم $WCET_{HI}$ است که با اندازه‌گیری‌های دقیق به دست می‌آید. زمانی که یک جاب از سطح بحرانی بالا، پس از اجرای $WCET_{LO}$ واحد زمانی تمام نشود، یعنی ممکن است که به اندازه‌ی $WCET_{HI}$ طول بکشد. به این حالت overrun می‌گوییم که schedulability سیستم را در خطر می‌اندازد. پس از وقوع overrun، تسک‌ها به اندازه‌ی $WCET_{HI}$ طول خواهند کشید. در این شرایط می‌توان تسک‌های با سطح بحرانی پایین را به طور کلی از سیستم خارج کرد و یا آنها را به یک هسته بدون overrun منتقل کرد که در این پروژه مورد دوم مدنظر است.

EDF-VD

یک روش زمان‌بندی مشابه EDF است که برای سیستم‌های بحرانی-مختلط بهینه شده است. در این روش، برای این که schedulability تسک‌ها را تضمین کنیم، یک ددلاین مجازی به تسک‌های HC اختصاص می‌دهیم و EDF را با توجه به آن اجرا می‌کنیم. در نتیجه وقتی که وارد حالت overrun می‌شویم، این زمان اضافه باعث می‌شود که جاب ها ددلاین را میس نکنند.

UUNIFAST

برای این که یک مجموعه تسک با utilization مشخص داشته باشیم، می‌توانیم از این الگوریتم استفاده کنیم. این الگوریتم یک utilization و تعداد تسک‌ها را می‌گیرد و به ما یک لیست از utilizationها با توزیع استاندارد می‌دهد. ما در این پروژه از الگوریتم uunifast-discard استفاده کردیم که یک نسخه از uunifast برای محیط‌های چندهسته‌ای است. این الگوریتم سعی می‌کند لیستی از utilizationها به ما بدهد که در نهایت utilization هر هسته به اندازه تعیین شده باشد و اگر موفق نبود، نتیجه را discard می‌کند و دوباره محاسبات را انجام می‌دهد. به همین دلیل خیلی بهینه نیست.

DBF

این واژه مخفف demand bound function است که یک روش برای تست زمان‌بندی پذیری است. فرمول آن در زیر نوشته شده است. مقداری که این تابع محاسبه می‌کند این است که تا زمان x، در مجموع چقدر از جاب‌ها باید تمام شده باشند. اگر که در لحظه‌ای این مقدار بزرگتر از x شود به این معنی است که زمان‌بندی قابل انجام نیست.

$$\forall x: \sum_{\tau_i \in \tau} dbf_i^{LO}(x) \leq x$$

$$dbf_i^{LO}(x) = n_i(x) \times C_i^{LO}$$

$$n_i(x) = max(floor(x - D_l)/T_i) + 1, 0)$$

پیاده سازی

در این بخش به توضیح پیاده‌سازی پروژه می‌پردازیم.

task.py

این کلاس، تسک‌ها را تعریف می‌کند. ویژگی‌های این کلاس، utilization، دوره، سطح بحرانی، حداکثر زمان‌های اجرا، و ددلاین‌هاست.

```
class Task:
   def __init__(
       self, number, name, utilization, period, criticality, low_wcet, high_wcet
   ):
       self.utilization = utilization
       self.period = period
       self.number = number
       self.name = name
       self.criticality = criticality
       self.high_wcet = high_wcet
       self.low_wcet = low_wcet if criticality == 1 else high_wcet
       self.relative_deadline = period
       self.virtual_deadline = period
       self.executed_jobs = 0
```

چون گفته‌شده که برای برای نگاشت اولویت با تسک‌های با سطح بحرانی بیشتر است، برای مقایسه بین تسک‌ها از این سنجه استفاده می‌کنیم.

```
   def __lt__(self, other):
       return self.criticality < other.criticality
```

از آنجایی که برای TMR، نیاز به دو نمونه‌ی دیگر از تسک‌ها داریم، یک کلاس جدا برای آنها می‌سازیم.

```
class TaskCopy(Task):
   def __init__(self, task, copy_number):
       super().__init__(
           task.number,
           task.name,
           task.utilization,
           task.period,
           task.criticality,
           task.low_wcet,
           task.high_wcet,
       )
       self.name = f"{task.name}_copy-{copy_number}"
```

هر جاب از یک تسک را با این کلاس مدل می‌کنیم. هر جاب برای خودش یک ددلاین دارد که از روی ددلاین نسبی و یا ددلاین مجازی محاسبه می‌شود.

```
class Job:
   def __init__(self, task, is_in_overrun):
       self.task = task
       self.number = task.executed_jobs
       self.deadline = (task.period * self.number) + (
           task.relative_deadline if is_in_overrun else task.virtual_deadline
       )
       self.remaining_exec_time = task.high_wcet if is_in_overrun else task.low_wcet
```

در الگوریتم EDF و EDF-VD، برای این که یک جاب را از داخل ready queue خارج کنیم و اجرا کنیم باید جاب با نزدیکترین ددلاین را انتخاب کنیم. اگر چند جاب با ددلاین یکسان داشتیم جاب با سطح بحرانی بالاتر را انتخاب می‌کنیم.

```
    def __lt__(self, other):
       if self.deadline == other.deadline:
           return self.task.criticality < other.task.criticality
       return self.deadline < other.deadline
```

از این کلاس هم برای مدل‌سازی جاب‌های مهاجرت کرده استفاده می‌کنیم.

```
class MigratedJob(Job):
   def __init__(self, job, migration_time):
       super().__init__(job.task)
       self.number = job.number
       self.deadline = job.deadline
       self.remaining_exec_time = job.remaining_exec_time
       self.migration_time = migration_time
```

task_generator.py

این کلاس وظیفه‌ی تولید تسک‌ها را بر عهده دارد.

این تابع وظیفه‌ی ساختن یک لیست از utilizationها با الگوریتم uunifast_discard را دارد. نحوه‌ی کار این است که از مجموع مقدار utilizationی که می‌خواهیم داشته باشیم، در هر مرحله با توزیع استاندارد یک قطعه‌ی کوچک از آن را جدا می‌کند و به عنوان utilization یک تسک در نظر می‌گیرد. وقتی که به تعداد تمام تسک‌ها utilization ساخته شد، این تابع چک می‌کند که آیا تمام این تسک‌ها utilization کوچکتر از ۱ دارند یا خیر و اگر داشتند، آرایه را به عنوان جواب در یک فایل می‌ریزد و در غیر این صورت، دوباره این کار را امتحان می‌کند. علت استفاده از retries این است که اگر یک task set قابل ساختن نداشتیم، اجرا متوقف شود. همچنین همانطور که مشاهده می‌کنید، در اینجا utilization را در ابتدای کار تقسیم بر دو کرده ایم. دلیل این کار این است که از تسک‌های HC سه نمونه داریم و به همین علت، میزان utilization واقعی دو برابر utilization تسک‌ها خواهد بود (با فرض این که نسبت HC به LC برابر ۱ باشد.)

```
def generate_uunifastdiscard(u: float, n: int, filename: str):
    retries = 0
    while retries < 1000:
        utilizations = []
        sumU = u * 0.5
        for i in range(1, n):
            nextSumU = sumU * random.random() ** (1.0 / (n - i))
            utilizations.append(sumU - nextSumU)
            sumU = nextSumU
        utilizations.append(sumU)

        if all(ut <= 1 for ut in utilizations):
            with open(filename, "w", newline="", encoding="UTF-8") as csvfile:
                writer = csv.writer(csvfile)
                writer.writerow(["Task " + str(i) for i in range(1, n + 1)])
                writer.writerow(utilizations)

            return utilizations

        retries += 1

    raise Exception("Could not generate utilization set")
```

این تابع هم وظیفه‌ی ساختن task setها از روی utilizationها را بر عهده دارد. نحوه کار آن به این صورت است که به ازای هر یک از utilizationهایی که داریم، یک period را به صورت رندوم انتخاب کرده و WCET تسک را بر اساس این دو مقدار محاسبه می‌کند. برای تسک‌های HC، دو مقدار برای WCET نیاز داریم. یکی مقدار WCET_LO و دیگری WCET_HI. مقدار دومی که از روی utilization به دست می‌آید. مقدار اولی را هم از ضرب یک عدد رندوم در حد 0.5 در مقدار WCET_HI به دست می‌آوریم. برای تسک‌های HC، دو کپی از آن را هم در لیست تسک‌ها درج می‌کنیم. سپس تسک‌های LC را تولید می‌کنیم. (علت استفاده از اعداد از پیش تعیین شده برای period این بود که یک hyper period کوچک داشته باشیم تا مشاهده‌ی زمان بندی راحت تر باشد.)

```
def generate_tasksets(utilizations, periods):
    task_set = []

    break_point = len(utilizations) // 2

    for indx, u in enumerate(utilizations[:break_point]):
        task_period = random.choice(periods)
        wcet_multiplier = random.uniform(0.3, 0.5)
```

```
            wcet = round(u * task_period, 3)

            new_task = t.Task(
                indx,
                f"Task-{indx}",
                u,
                task_period,
                t.TASK_PRIORITIES["high"],
                round(wcet_multiplier * wcet, 3),
                wcet,
            )

            task_set.append(new_task)

            for i in range(1, 3):
                task_set.append(t.TaskCopy(new_task, i))

    for indx, u in enumerate(utilizations[break_point:]):
        real_indx = indx + break_point
        task_period = random.choice(periods)
        wcet = round(u * task_period, 3)

        task_set.append(
            t.Task(
                real_indx,
                f"Task-{real_indx}",
                u,
                task_period,
                t.TASK_PRIORITIES["low"],
                wcet,
                wcet,
            )
        )
    return task_set
```

این تابع هم وظیفه‌ی ایجاد تسک‌ها را بر عهده دارد و توابع قبلی را صدا می‌زند.

```
def generate_tasks(utilization: float, n: int, available_periods: list):
    utilizations = generate_uunifastdiscard(
        u=utilization, n=n, filename="task_utilizations.csv"
    )

    task_set = generate_tasksets(utilizations, available_periods)

    return task_set
```

این تابع وظیفه‌ی اجرای dbf بر روی سیستم را دارد. به این صورت که برای هر هسته، تسک‌هایی که روی آن هسته هستند را پیدا کرده و dbf آن‌ها را بررسی می‌کند.

```
def dbf_by_core(task_set: list[t.Task], processor, hyper_period):
    for core in processor.cores:
        core_tasks = [task for task in task_set if task.assigned_core == core]
        demand_bound_function_tester(core_tasks, hyper_period)
```

این تابع مقدار dbf را برای یک هسته در تمام زمان ها از ۱ تا هایپر-پریود محاسبه می‌کند و اگر موفقیت آمیز نبود خطا برمیگرداند.

```
def demand_bound_function_tester(task_set: list[t.Task], hyper_period):
    for i in range (0, hyper_period):
        demand = 0
        for task in task_set:
            demand += demand_bound_function(task, i)
        if demand > i:
            raise Exception("DBF failed")
```

این تابع مقدار dbf را برای یک لحظه و یک تسک محاسبه می‌کند. فرمول محاسبه‌ی آن در ابتدای گزارش ذکر شده است.

```
def demand_bound_function(task: t.Task, x):
    maximal_jobs = max(0, 1 + math.floor((x - task.relative_deadline)/task.period))
    dbf = maximal_jobs * task.high_wcet
    return dbf
```

processor.py

این کلاس نشان‌دهنده‌ی یک هسته است.

```
class Core:
   def __init__(self, number, max_utilization):
       self.number = number
       self.max_utilization = max_utilization
       self.utilization = 0
       self.assigned_tasks = []
       self.is_in_overrun = False
       self.is_susceptible_to_overrun = False
```

این کلاس نشان‌دهنده‌ی سیستم است که تعدادی هسته دارد.

```
class Processor:
   def __init__(self, num_of_cores, core_utilization):
       self.cores = []
       for i in range(num_of_cores):
           self.cores.append(Core(i + 1, core_utilization))
```

این تابع وظیفه‌ی نگاشت وظایف روی هسته‌ها را دارد. به این صورت که لیست تسک‌ها را دریافت کرده و با توجه به الگوریتمی که در ورودی برنامه مشخص شده (wfd, ffd)، این نگاشت را انجام می‌دهد. همانطور که در توضیحات پروژه ذکر شده، اولویت با نگاشت تسک‌های با سطح بحرانی بالا بوده‌است.

```
    def map_tasks(self, task_set, method):
       tasks = task_set.copy()
       tasks.sort(key=lambda task: task.utilization, reverse=True)
       assigned_tasks = []
       sorted_cores = self.cores.copy()
       for task in tasks:
           if method == "wfd":
               sorted_cores.sort(key=lambda core: core.utilization)

           selected_core = None

           for core in sorted_cores:
               if (task.number not in core.assigned_tasks) and (
                   core.utilization + task.utilization <= core.max_utilization
               ):
                   selected_core = core
                   break

           if selected_core is None:
               for core in sorted_cores:
                   if (task.number not in core.assigned_tasks) and (
                       core.utilization + task.utilization <= 1
                   ):
                       selected_core = core
                       break

           if selected_core is not None:
               selected_core.assigned_tasks.append(task.number)
               selected_core.utilization += task.utilization
               task.assigned_core = selected_core
              else:
               raise Exception(
                   "task is set is not schedulable with worst fit assignment"
               )

           assigned_tasks.append(task)

       return assigned_tasks
```

این تابع در زمان overrun شدن هسته صدا زده می‌شود. وظیفه‌ی آن این است که نگاشت تسک‌های با اولویت پایین به هسته را از بین ببرد و ددلاین و زمان اجرای تسک hc که در هنگام اجرا overrun شده است را درست کند.

```
   def handle_overrun(self, active_jobs):
```

```
        for job in active_jobs:
            if job.task.criticality == t.TASK_PRIORITIES["high"]:
                task = job.task
                job.remaining_exec_time = (
                    task.high_wcet - task.low_wcet + job.remaining_exec_time
                )
                job.deadline = (task.period * job.number) + task.relative_deadline
            else:
                job.task.assigned_core = None
```

این تابع وظیفه‌ی زمان بندی edf و edf_vd را بر عهده دارد. به این صورت که در یک حلقه‌ی while، به ازای هر واحد زمانی، در ابتدا به ازای هر تسک، اگر زمان arrival time یک جاب جدید از آن رسیده بود، آن را وارد سیستم می‌کند. سپس به ازای هر هسته، یک جاب را برای اجرا روی آن انتخاب می‌کند؛ به این صورت که ابتدا بین تمام جاب‌هایی که مربوط به آن هسته هستند، با اولویت‌ترین آن‌ها را انتخاب می‌کند و اگر هیچ جابی در لحظه برای آن هسته وجود نداشت، یکی از جاب‌هایی که migrate کرده اند را انتخاب می‌کند. جاب انتخاب شده به اندازه‌ی یک واحد زمانی اجرا می‌شود و آپدیت می‌شود.

```
    def schedule_edf(self, task_set, duration, overrun_time, use_vd):
        current_time = 0
        schedule_timeline = []
        active_jobs = []
        while duration > current_time:
            # find active_jobs
            for task in task_set:
                if current_time % task.period == 0:
                    is_core_in_overrun = False
                    if task.assigned_core is not None:
                        is_core_in_overrun = task.assigned_core.is_in_overrun

                    new_job = t.Job(
                        task,
                        is_core_in_overrun,
                        use_vd and is_core_in_overrun,
                    )
                    active_jobs.append(new_job)
                    task.executed_jobs += 1

            timestamp = []
            for core in self.cores:
                core_jobs = []
                selected_job = None

                # find acive jobs for a core, or migrating jobs
                for job in active_jobs:
                    if job.task.assigned_core == core:
                        core_jobs.append(job)

                if len(core_jobs) == 0:
                    # if core empty, execute a migrated job
                    migrated_jobs = [
                        job for job in active_jobs if job.task.assigned_core == None
                    ]
                    if len(migrated_jobs) > 0:
                        migrated_jobs.sort()
                        selected_job = migrated_jobs[0]
                else:
                    # schedule job with highest priority on core
                    core_jobs.sort()
                    selected_job = core_jobs[0]

                if selected_job is not None:
                    selected_task = selected_job.task

                    selected_job.remaining_exec_time = round(
                        selected_job.remaining_exec_time - 0.001, 3
                    )
                    if selected_job.remaining_exec_time == 0:
                        active_jobs.remove(selected_job)
                        if (
                            overrun_time is not None
```

```
                        and core.is_susceptible_to_overrun
                        and not core.is_in_overrun
                        and current_time >= overrun_time
                        and selected_task.criticality == t.TASK_PRIORITIES["high"]
                    ):
                        core.is_in_overrun = True
                        self.handle_overrun(active_jobs)

                if (
                    selected_job.remaining_exec_time > 0
                    and current_time == selected_job.deadline
                    and selected_task.criticality == t.TASK_PRIORITIES["high"]
                ):
                    raise Exception(
                        f"deadline for job {selected_job.number} of {selected_job.task.name} missed!"
                    )

                timestamp.append(
                    {
                        "task": selected_task,
                        "job": selected_job,
                        "core": core.number,
                        "overrun": core.is_in_overrun,
                    }
                )
            else:
                timestamp.append(
                    {
                        "task": None,
                        "job": None,
                        "core": core.number,
                        "overrun": core.is_in_overrun,
                    }
                )

        schedule_timeline.append(timestamp)

        current_time = round(current_time + 0.001, 3)

    return schedule_timeline
```

این تابع، مقدار virtual deadline را برای تسک‌های یک هسته حساب می‌کند. به این صورت که مقدار utilization تسک‌های با سطح بحرانی پایین و بالا را برای آن هسته محاسبه کرده و با توجه به آن، مقدار ضریب x را به دست می‌آورد و ددلاین مجازی را برای تسک‌های HC آن هسته محاسبه می‌کند. در نهایت هم تعدادی از هسته ها را مشخص می‌کند که overrun برای آن‌ها اتفاق بیفتد و سپس تابع `schedule_edf` را صدا می‌زند تا زمان‌بندی انجام شود.

```
def schedule_tasks(self, task_set, duration, overrun_rate, scheduling_method):
    for core in self.cores:
        # get tasks assigned to this core
        core_tasks = [task for task in task_set if task.assigned_core == core]

        # calculate virtual deadline for tasks in this core
        sum_of_high_crit_util = sum(
            [
                task.low_wcet / task.period
                for task in core_tasks
                if task.criticality == t.TASK_PRIORITIES["high"]
            ]
        )
        sum_of_high_crit_high_wcet_util = sum(
            [
                task.utilization
                for task in core_tasks
                if task.criticality == t.TASK_PRIORITIES["high"]
            ]
        )
        sum_of_low_crit_util = sum(
            [
                task.utilization
                for task in core_tasks
```

```
                if task.criticality == t.TASK_PRIORITIES["low"]
            ]
        )
        virtual_deadline_multiplier = sum_of_high_crit_util / (
            1 - sum_of_low_crit_util
        )

        # test schedulability using gained x
        if (
            (virtual_deadline_multiplier * sum_of_low_crit_util)
            + sum_of_high_crit_high_wcet_util
        ) > 1:
            raise Exception("not schedulable")

        # apply virtual deadline
        for task in core_tasks:
            if task.criticality == t.TASK_PRIORITIES["high"]:
                task.virtual_deadline = task.period * virtual_deadline_multiplier

n = math.floor(overrun_rate * len(self.cores))
print(n)
susceptible_cores = random.sample(self.cores, n)
for core in susceptible_cores:
    core.is_susceptible_to_overrun = True

return self.schedule_edf(
    task_set,
    duration,
    None if overrun_rate == 0 else duration / 2,
    scheduling_method == "edf-vd",
)
```

main

این فایل، نقطه‌ی شروع برنامه است. در ابتدا، تعدادی پریود را از قبل تعیین کرده‌ایم. دلیل این کار این است که میزان hyper-period از قبل معلوم باشد و زمان‌بندی راحت‌تر مشخص شود. سپس ورودی‌ها گرفته می‌شود، تسک‌ست ساخته می‌شود و عمل نگاشت، تست زمان‌بندی، و خود زمان بندی انجام می‌شود. نتیجه‌ی نگاشت و زمان‌بندی هم در یک فایل نوشته می‌شود و کار تمام می‌شود.

```
AVAILABLE_PERIODS = [10, 20, 25, 50, 100, 200, 250, 500, 1000]

core_utilization = float(input("Enter utilization of each core: "))
num_of_cores = int(input("Enter number of cores: "))
num_of_tasks = int(input("Enter number of tasks: "))
assignment_method = input("Enter assignment method (wfd or ffd): ")
scheduling_method = input("Should edf-vd be used?(y/n) ")
scheduling_method = "edf-vd" if scheduling_method == "y" else "edf"
overrun_rate = float(input("Enter overrun rate: "))

task_set = tg.generate_tasks(
    num_of_cores * core_utilization, num_of_tasks, AVAILABLE_PERIODS
)

processor = p.Processor(num_of_cores, core_utilization)
assigned_tasks = processor.map_tasks(task_set, assignment_method)
tg.dbf_by_core(assigned_tasks, processor, 1000)


mock_cores = [0 for _ in range(num_of_cores)]

with open(
    f"{num_of_cores}_cores_{core_utilization}_utilization_{assignment_method}_mapping.csv",
    "w",
    newline="",
    encoding="UTF-8",
) as file:
    writer = csv.writer(file)
    writer.writerow(
        [
            "task",
            "utilization",
            "period",
            "criticality",
            "assigned core",
            "core utilization",
        ]
    )
    for i, task in enumerate(assigned_tasks):
        assigned_core_number = task.assigned_core.number
        mock_cores[assigned_core_number - 1] += task.utilization
        writer.writerow(
            [
                task.name,
                task.utilization,
                task.period,
                "high" if task.criticality == t.TASK_PRIORITIES["high"] else "low",
                task.assigned_core.number,
                mock_cores[assigned_core_number - 1],
            ]
        )

schedules = processor.schedule_tasks(assigned_tasks, 1000, overrun_rate, scheduling_method)

with open(

f"{num_of_cores}_cores_{core_utilization}_utilization_{assignment_method}_{scheduling_method}_scheduling.csv",
    "w",
    newline="",
    encoding="UTF-8",
) as file:
    writer = csv.writer(file)
    header = ["time"]
    for i in range(num_of_cores):
```

```
        header.append(f"core {i + 1}")

    writer.writerow(header)

    for time, timeslot in enumerate(schedules):
        row = [time / 1000]
        timeslot.sort(key=lambda x: x["core"])

        # for core in timeslot:
        #     row.append(f'{(overrun)} if core["overrun"] else ''}')

        writer.writerow(row)
```

خروجی‌ها

در ابتدا نمونه‌ای از فایل‌های اکسل خروجی نمایش می‌دهیم:

tasks

task	utilization	period	relative deadline	wcet	criticality
Task-0	0.0925903679364336	200	200	{6.127, 18.518}	high
Task-1	0.332600570123756	25	25	{2.709, 8.315}	high
Task-2	0.0901639746835258	200	200	{7.741, 18.033}	high
Task-3	0.00280287768723692	10	10	{0.01, 0.028}	high

utilization

Task 1	Task 2	Task 3	Task 4	Task 5
0.234297171744434	0.286719322442019	0.190798640396587	0.234532850883459	0.0536520145335018

mappings

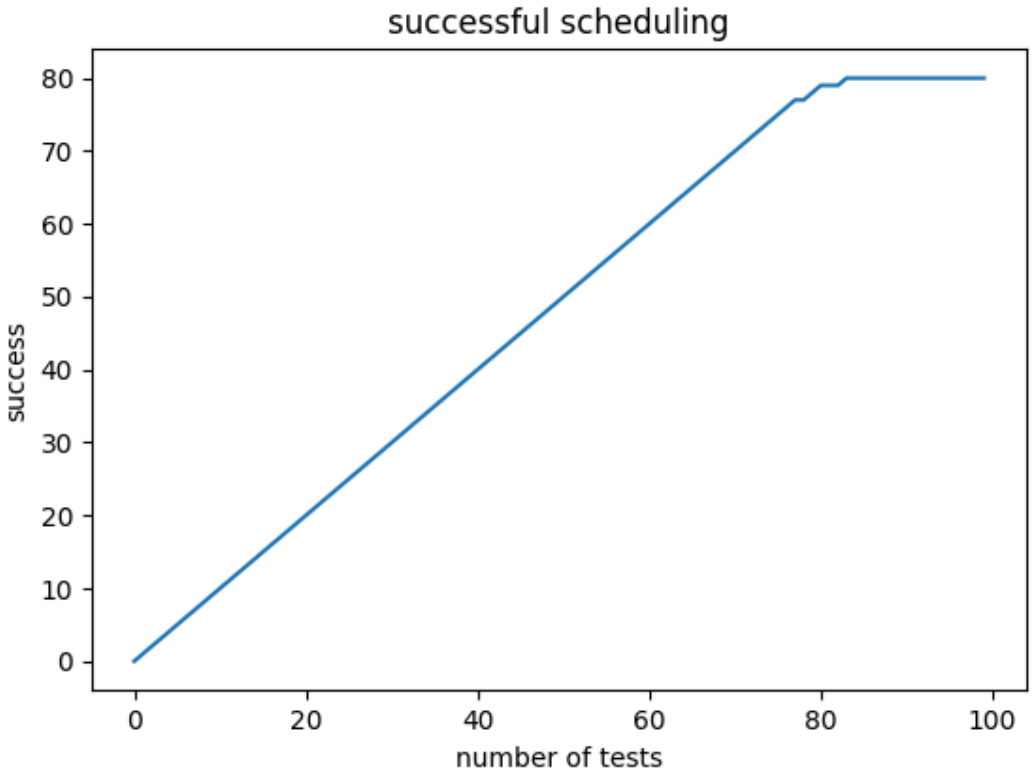
task	utilization	period	criticality	assigned core	core utilization
Task-1	0.286719322442019	4	high	1	0.286719322442019
Task-1_copy-1	0.286719322442019	4	high	2	0.286719322442019
Task-1_copy-2	0.286719322442019	4	high	3	0.286719322442019
Task-3	0.234532850883459	4	low	4	0.234532850883459
Task-0	0.234297171744434	2	high	4	0.468830022627893

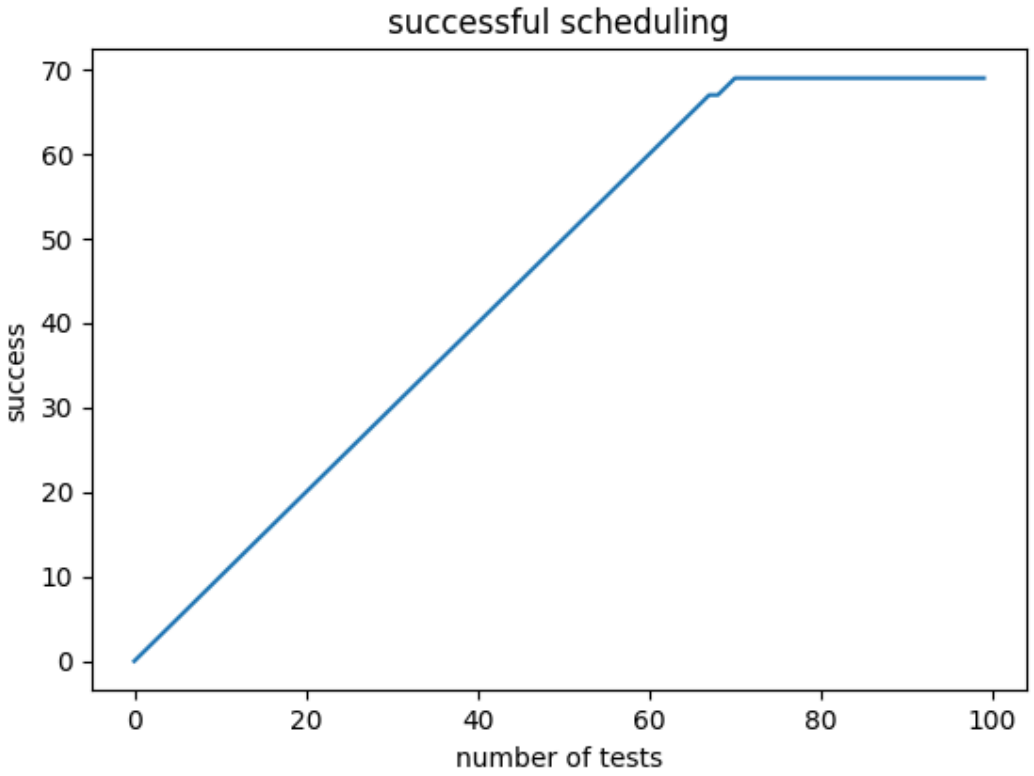
scheduling

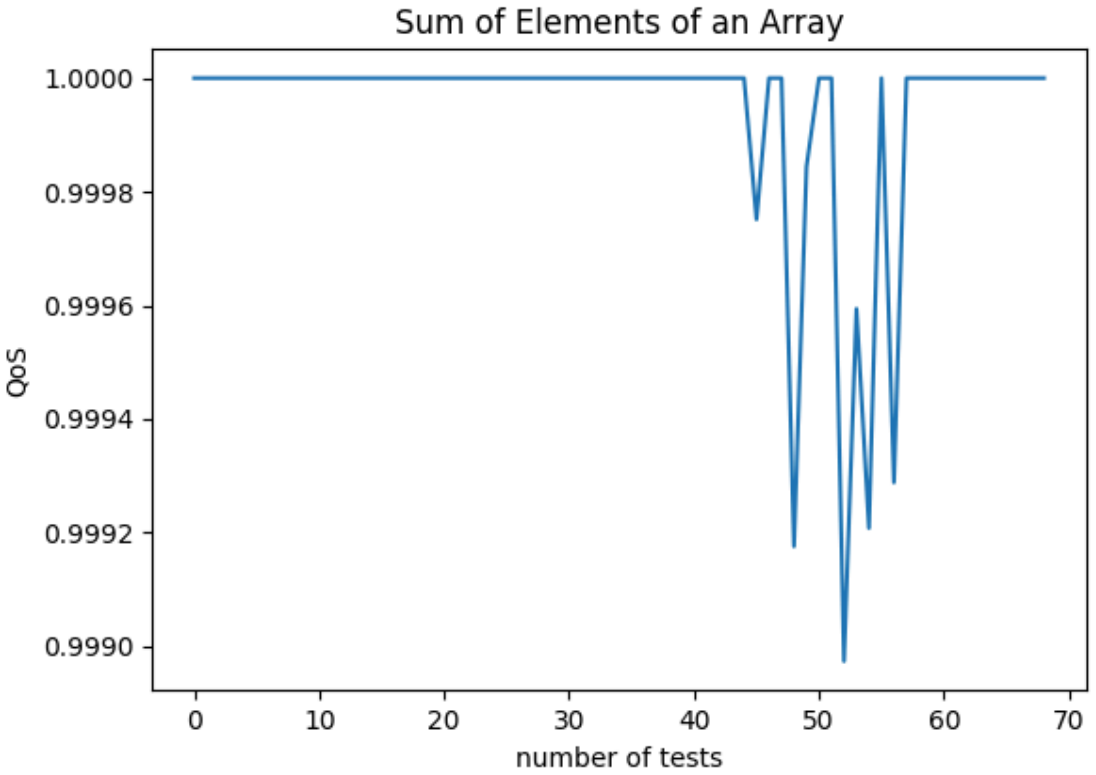
time	core 1	core 2	core 3	core 4
0.23	Task-1, Job_0	Task-0_copy-2, Job_0	Task-0_copy-1, Job_0	Task-0, Job_0
0.231	Task-1, Job_0	Task-0_copy-2, Job_0	Task-0_copy-1, Job_0	Task-0, Job_0
0.232	Task-1, Job_0	Task-1_copy-1, Job_0	Task-1_copy-2, Job_0	Task-4, Job_0

حال نمودارها را رسم می‌کنیم:

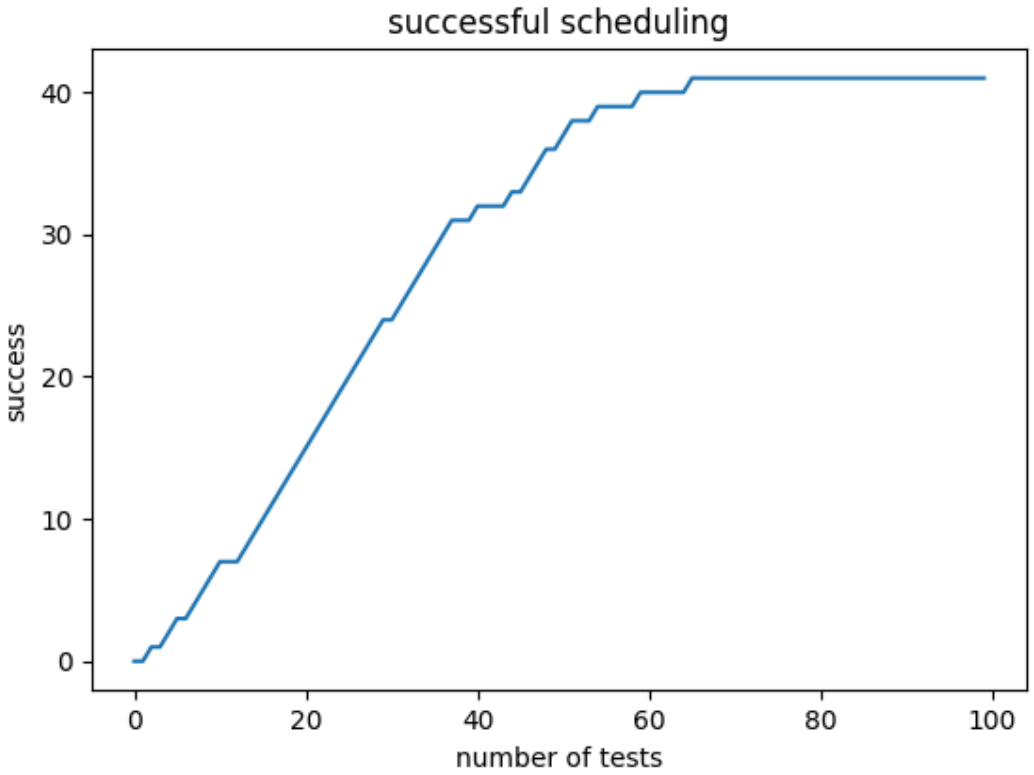
حالت اول: سامانه ۸ هسته‌ای با بهره‌وری ۰.۵

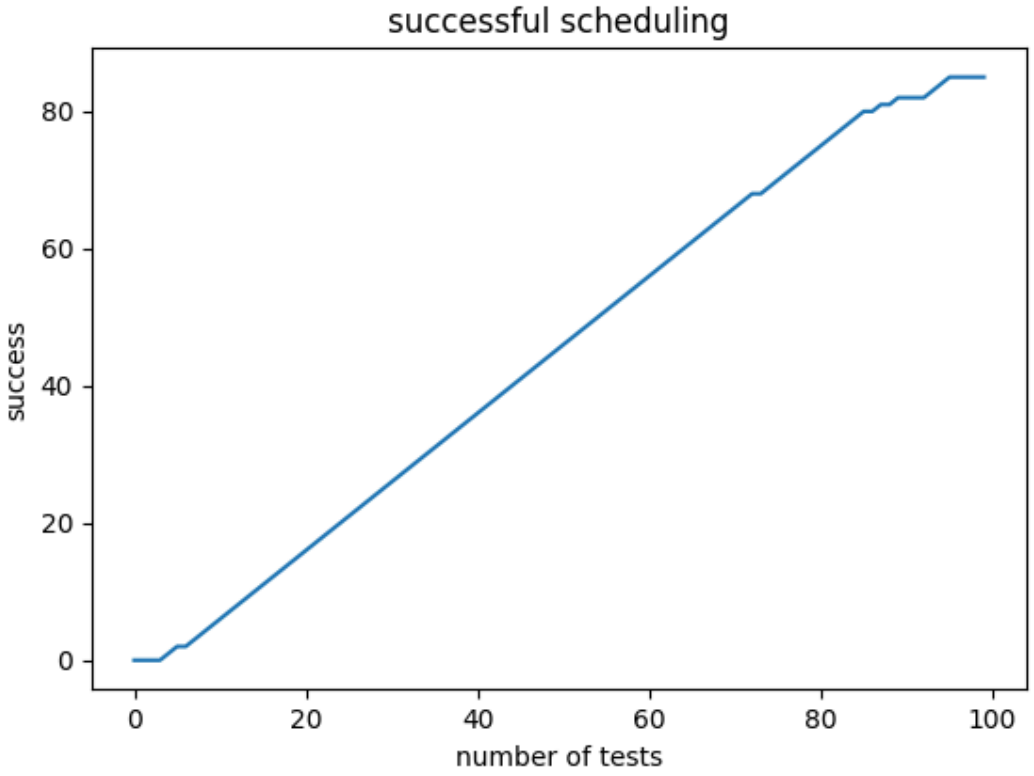

successful scheduling
success
number of tests
0
10
20
30
40
50
60
70
80
0
20
40
60
80
100

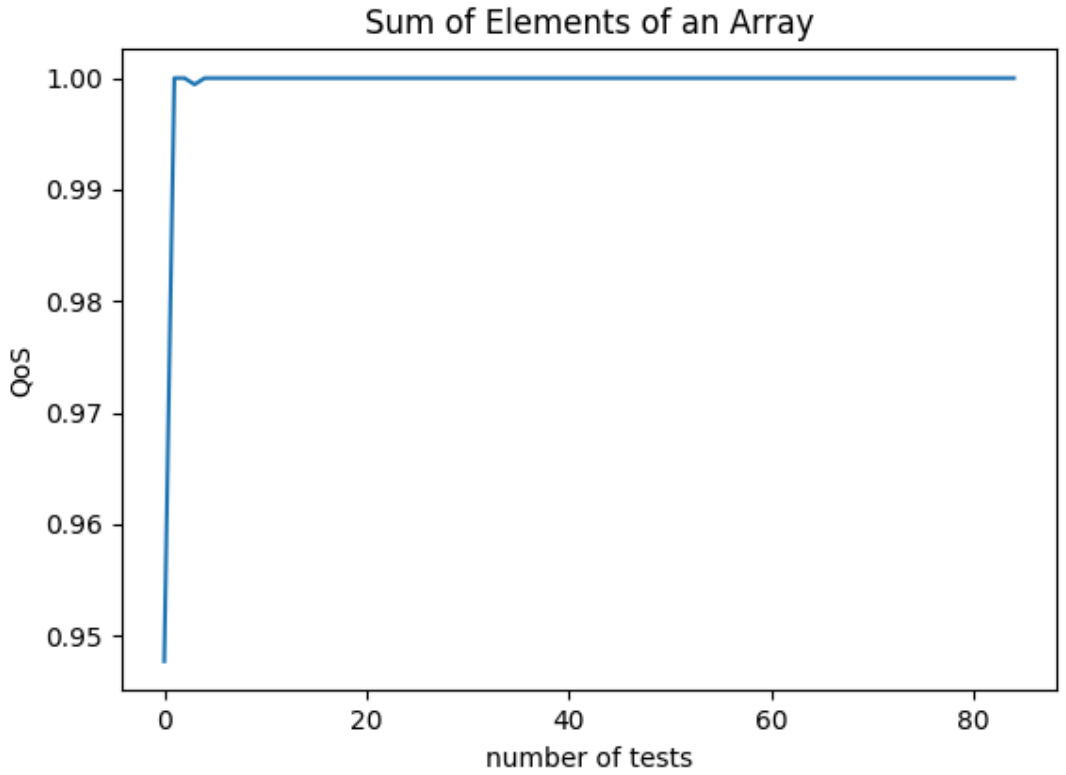

successful scheduling
success
number of tests
0
10
20
30
40
50
60
70
0
20
40
60
80
100

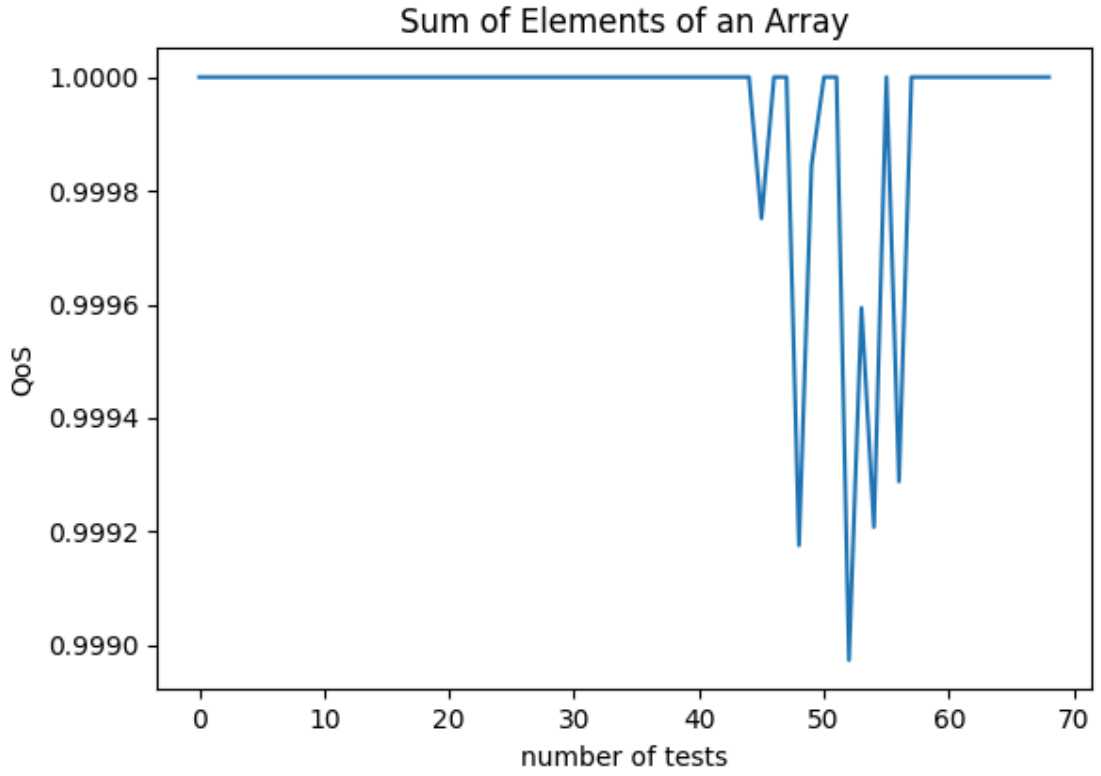


حالت دوم:سیستم ۱۶ هسته‌ای، با بهره‌وری ۷۵درصد و ۳۰ درصد خطا و ۵۰ درصد overrun:


successful scheduling
success
number of tests
0
10
20
30
40
0
20
40
60
80
100


successful scheduling
success
number of tests
0
20
40
60
80
0
20
40
60
80
100





علت مناسب نبودن الگوریتم EDF این است که این الگوریتم هیچ پیش‌بینی‌ای نسبت به اتفاق‌هایی مثل overrun و fault ندارد و در نتیجه این اتفاق باعث می‌شود که وقتی مشکلی در سیستم رخ داد، به علت

laxity پایین، تسک‌های HC به ددلاین خود نزدیک‌تر شوند. به همین دلیل این الگوریتم برای سامانه‌های بحرانی-مختلط مناسب نیست.
همچنین نتایج نگاشت وظایف در فایل‌های اکسل قرار داده شده است.