عنوان: فاز اول پروژه

استاد: خانم دکتر صفری

مسئول: آقای اثنی عشری

نویسندگان: امیررضا آذری – بزرگمهر ضیا

شماره دانشجویی: 99101087 - 99100422

سیستم‌های بی‌درنگ

پاییز 1403

مقدمه

این سند پیاده‌سازی یک سیستم زمان‌بندی تسک را توضیح می‌دهد که شامل یادگیری تقویتی(RL) ، زمان‌بندی بر اساس زودترین ددلاین(EDF) ، و مکانیزم تولید تسک بر اساس الگوریتم UUniFast است. هر بخش به تفصیل بحث شده و نقش آن در سیستم کلی توضیح داده شده است.

الگوریتم UUNIFAST

الگوریتم UUniFast برای تولید استفاده‌پذیری تسک‌ها استفاده می‌شود. هدف این الگوریتم توزیع یک مقدار کل استفاده‌پذیری بین تعداد مشخصی از تسک‌ها است، به گونه‌ای که مجموع استفاده‌پذیری‌ها برابر مقدار کل باشد.

جزئیات پیاده‌سازی:

- ورودی‌ها:

✓ `num_tasks` : تعداد تسک‌هایی که باید تولید شوند.
✓ `total_utilization` : مجموع استفاده‌پذیری که باید بین تسک‌ها تقسیم شود.

- منطق:

✓ باقی‌مانده استفاده‌پذیری (sum_u) به طور تکراری بین تسک‌ها تقسیم می‌شود.
✓ یک کسر تصادفی تعیین می‌کند چه مقدار از استفاده‌پذیری باقی‌مانده به تسک فعلی اختصاص داده شود.
✓ تسک نهایی تمام استفاده‌پذیری باقی‌مانده را دریافت می‌کند.

- خطوط کلیدی کد:

✓ `next_u = sum_u * (random.uniform(0, 1) ** (1 / (num_tasks - i)))`: تعیین تصادفی کسر بعدی استفاده‌پذیری.
✓ `utilizations.append(sum_u - next_u)`: اضافه کردن استفاده‌پذیری محاسبه‌شده برای تسک فعلی.

- خروجی:

✓ یک آرایه از بهره‌وری‌ها که مجموع آن برابر total_utilization است.

```python
def uunifast(num_tasks, total_utilization):
    utilizations = []
    sum_u = total_utilization

    for i in range(1, num_tasks):
        next_u = sum_u * (random.uniform(0, 1) ** (1 / (num_tasks - i)))
        utilizations.append(sum_u - next_u)
        sum_u = next_u

    utilizations.append(sum_u)
    return utilizations
```

تولید تسک‌ها

تسک‌ها با استفاده از استفاده‌پذیری‌های تولیدشده توسط UUniFast تولید می‌شوند. هر تسک دارای ویژگی‌هایی مانند دوره، بدترین زمان اجرا(WCET) ، و ددلاین است.

جزئیات پیاده‌سازی:

- ورودی‌ها:

✓ num_tasks, total_utilization: به UUniFast ارسال می‌شود.
✓ min_period, max_period: محدوده‌ای برای دوره تسک‌ها تعریف می‌کند.

- منطق:

✓ دوره هر تسک به صورت تصادفی در محدوده انتخاب می‌شود.
✓ WCETبه صورت utilization * period محاسبه می‌شود.
✓ ددلاین‌ها برابر دوره‌ها تنظیم می‌شوند.

- خطوط کلیدی کد:

✓ انتخاب تصادفی دوره تسک: period = random.randint(min_period, max_period)

✓ محاسبه WCET: wcet = utilization * period

- خروجی:

یک لیست از تسک‌ها که هر تسک به صورت یک دیکشنری با ویژگی‌های زیر نمایش داده می‌شود:

✓ task_id: شناسه.

✓ utilization, period, wcet, deadline.

```python
def generate_tasks(num_tasks, total_utilization, min_period, max_period):
    utilizations = uunifast(num_tasks, total_utilization)
    tasks = []

    for i, utilization in enumerate(utilizations):
        period = random.randint(min_period, max_period)
        wcet = utilization * period
        task = {
            "task_id": i,
            "utilization": round(utilization, 4),
            "period": period,
            "wcet": round(wcet, 4),
            "deadline": period
        }
        tasks.append(task)

    return tasks
```

یادگیری تقویتی:

عامل RL یاد می‌گیرد که تسک‌ها را به کورها تخصیص دهد با استفاده از شبکه Q عمیق .(DQN)
محیط شبیه‌سازی اجراهای تسک و محاسبه پاداش بر اساس بهره‌وری انرژی و رعایت ددلاین است.

کلاس DVFSEnvironment

- هدف:
  - ✓ مدل‌سازی کورها و تسک‌ها.
  - ✓ شبیه‌سازی اجرای تسک و محاسبه معیارهایی مانند مصرف انرژی.
- متودها:
  - ✓ initialize_state: تنظیم اولیه کورها و صف تسک.
  - ✓ step: اجرای یک اقدام (تخصیص تسک به کور) و به‌روزرسانی وضعیت.
  - ✓ reset: بازنشانی محیط برای یک قسمت جدید.

شبکه Q عمیق (DQN)

- معماری:
  - ✓ سه لایه کاملاً متصل با فعال‌سازی ReLU.
  - ✓ ورودی: بردار وضعیت (صف تسک، وضعیت کورها).
  - ✓ خروجی: مقادیر Q برای تمام اقدامات ممکن.

حلقه آموزش

- ✓ در هر قسمت، عامل RL با محیط تعامل دارد.
- ✓ اقدامات بر اساس مقادیر Q انتخاب می‌شوند و شبکه با استفاده از معادله بلمن آموزش داده می‌شود.
- ✓ تخصیص نهایی تسک به کورها ذخیره می‌شود.

```python
class DVFSEnvironment:
    def __init__(self, num_cores, frequencies, tasks):
        self.num_cores = num_cores
        self.frequencies = frequencies
        self.tasks = tasks
        self.state = self._initialize_state()
        self.energy_consumption = 0

    def _initialize_state(self):
        return {
            "cores": [{"load": 0, "frequency": self.frequencies[0]} for _ in range(self.num_cores)],
            "task_queue": self.tasks.copy()
        }

    def step(self, action):
        task_id, core_id, freq_id = action

        # Ensure task_id is within the current task queue
        if task_id >= len(self.state["task_queue"]):
            raise IndexError("Task ID out of range for current task queue.")

        task = self.state["task_queue"][task_id]
        core = self.state["cores"][core_id]
        frequency = self.frequencies[freq_id]

        core["load"] += task["wcet"] / frequency
        core["frequency"] = frequency

        energy = frequency * task["wcet"]
        self.energy_consumption += energy

        reward = -energy
        if core["load"] > task["deadline"]:
            reward -= 100

        self.state["task_queue"].pop(task_id)
        done = len(self.state["task_queue"]) == 0

        return self.state, reward, done

    def reset(self):
        self.state = self._initialize_state()
        self.energy_consumption = 0
        return self.state
```

زمان‌بندی بر اساس زودترین ددلاین(EDF)

پس از تخصیص تسک‌ها به کورها، تسک‌ها در هر کور با استفاده از EDF زمان‌بندی می‌شوند.

- منطق:
  - مرتب‌سازی:

    تسک‌های هر کور بر اساس ددلاین به صورت صعودی مرتب می‌شوند.

  - اجرا:

    زمان شروع و پایان به صورت ترتیبی بر اساس WCET محاسبه می‌شود.

  - خطوط کلیدی کد:

```
final_assignments[core_id].sort(key=lambda x:
x["deadline"]):
```

مرتب‌سازی تسک‌ها بر اساس ددلاین.

```
finish_time = start_time + task["wcet"]:
```

محاسبه زمان پایان تسک.

- خروجی:

✓ زمان شروع، زمان پایان، و ددلاین هر تسک.

✓ خلاصه استفاده‌پذیری برای تمام کورها.

خروجی‌ها و اعتبارسنجی

✓ تسک‌ها همراه با کورهای تخصیص‌یافته نمایش داده می‌شوند.

✓ زمان شروع و پایان برای رعایت EDF تضمین می‌شود.

✓ نمایش استفاده‌پذیری برای هر کور و اعتبارسنجی اینکه از حد مجاز (۱.۰) تجاوز نکرده است.

```python
def train_rl(num_tasks, total_utilization, num_cores, frequencies, min_period, max_period, episodes):
    tasks = generate_tasks(num_tasks, total_utilization, min_period, max_period)
    env = DVFSEnvironment(num_cores, frequencies, tasks)

    state_dim = (2 * num_cores) + (2 * num_tasks)
    action_dim = num_tasks * num_cores * len(frequencies)

    agent = DQN(state_dim, action_dim)
    optimizer = optim.Adam(agent.parameters(), lr=0.001)
    criterion = nn.MSELoss()

    gamma = 0.99  # Discount factor
    final_assignments = {core_id: [] for core_id in range(num_cores)}  # For tracking assignments
    task_to_core = {task["task_id"]: None for task in tasks}  # Map tasks to cores

    for episode in range(episodes):
        state = env.reset()
        done = False
        total_reward = 0

        while not done:
            state_vector = torch.tensor(
                [state_to_vector(state, num_cores, frequencies, num_tasks)], dtype=torch.float32
            )
            q_values = agent(state_vector)

            # Apply masking to valid actions
            valid_actions = []
            for task_id in range(len(state["task_queue"])):
                for core_id in range(num_cores):
                    for freq_id in range(len(frequencies)):
                        valid_actions.append((task_id, core_id, freq_id))

            valid_action_indices = [
                (task_id * num_cores * len(frequencies)) + (core_id * len(frequencies)) + freq_id
                for task_id, core_id, freq_id in valid_actions
            ]

            valid_q_values = q_values[0, valid_action_indices]
            best_action_index = torch.argmax(valid_q_values).item()

            task_id, core_id, freq_id = valid_actions[best_action_index]

            next_state, reward, done = env.step((task_id, core_id, freq_id))

            # Update final assignments for last episode
            if episode == episodes - 1:
                if task_to_core[tasks[task_id]["task_id"]] is None:
                    final_assignments[core_id].append(tasks[task_id])
                    task_to_core[tasks[task_id]["task_id"]] = core_id

            next_state_vector = torch.tensor(
                [state_to_vector(next_state, num_cores, frequencies, num_tasks)], dtype=torch.float32
            )
```

جمع‌بندی:

در این فاز به تولید تسک با الگوریتم UUNIFAST پرداختیم و سپس به بخش اصلی و چالشی پروژه که یادگیری تقویتی بود رفتیم. در این بخش اکشن‌ها و ریواردها را مشخص نموده و با کمک Q-Values توانستیم نگاشت را انجام بدهیم. در نهایت با EDF توانستیم زمانبندی را برای هر کور انجام بدهیم.