دانشکده‌ی مهندسی کامپیوتر

الگوریتم تخلیه پردازش چندکاربره برای کمینه کردن تاخیر در دستگاه‌های اینترنت اشیاء

پروژه‌ی پایانی کارشناسی مهندسی کامپیوتر

محمدمبین داریوش همدانی

استاد راهنما

رضا انتظاری ملکی

تیر ۱۴۰۱


بسم الله الرحمن الرحيم

تأییدیه‌ی هیأت داوران جلسه‌ی دفاع از پروژه

نام دانشکده: دانشکده‌ی مهندسی کامپیوتر

نام دانشجو: محمدمبین داریوش همدانی

عنوان پروژه : الگوریتم تخلیه پردازش چند کاربره برای کمینه کردن تاخیر در دستگاه‌های اینترنت اشیاء

تاریخ دفاع: تیر ۱۴۰۱

رشته: مهندسی کامپیوتر

ردیف	سمت	نام و نام خانوادگی	مرتبه‌ی دانشگاهی	دانشگاه یا مؤسسه	امضـــــــــــا
۱	استاد راهنما	دکتر رضا انتظاری ملکی	استادیار	دانشگاه علم و صنعت ایران	
۲	استاد داور داخلی	دکتر	.....	دانشگاه علم و صنعت ایران	

تأییدیه‌ی صحت و اصالت نتایج

باسمه تعالی

اینجانب محمدمبین داریوش همدانی به شماره دانشجویی ۹۶۵۲۱۱۹۱ دانشجوی رشته مهندسی کامپیوتر مقطع تحصیلی کارشناسی تأیید می‌نمایم که کلیه‌ی نتایج این پروژه حاصل کار اینجانب و بدون هرگونه دخل و تصرف است و موارد نسخه‌برداری‌شده از آثار دیگران را با ذکر کامل مشخصات منبع ذکر کرده‌ام. درصورت اثبات خلاف مندرجات فوق، به تشخیص دانشگاه مطابق با ضوابط و مقررات حاکم (قانون حمایت از حقوق مؤلفان و مصنفان و قانون ترجمه و تکثیر کتب و نشریات و آثار صوتی، ضوابط و مقررات آموزشی، پژوهشی و انضباطی) با اینجانب رفتار خواهد شد و حق هرگونه اعتراض درخصوص احقاق حقوق مکتسب و تشخیص و تعیین تخلف و مجازات را از خویش سلب می‌نمایم. در ضمن، مسؤولیت هرگونه پاسخگویی به اشخاص اعم از حقیقی و حقوقی و مراجع ذی‌صلاح (اعم از اداری و قضایی) به عهده‌ی اینجانب خواهد بود و دانشگاه هیچ‌گونه مسؤولیتی در این خصوص نخواهد داشت.

نام و نام خانوادگی: محمدمبین داریوش همدانی

تاریخ و امضا:

مجوز بهره‌برداری از پایان‌نامه

بهره‌برداری از این پایان‌نامه در چهارچوب مقررات کتابخانه و با توجه به محدودیتی که توسط استاد راهنما به شرح زیر تعیین می‌شود، بلامانع است:

□ بهره‌برداری از این پایان‌نامه برای همگان بلامانع است.

□ بهره‌برداری از این پایان‌نامه با اخذ مجوز از استاد راهنما، بلامانع است.

□ بهره‌برداری از این پایان‌نامه تا تاریخ ..... ممنوع است.

استاد راهنما: رضا انتظاری ملکی

تاریخ:

امضا:

چکیده

رایانش لبه‌ای الگویی از محاسبات توزیع‌شده است که با نزدیک کردن منابع پردازشی به لبه‌ی شبکه، سعی دارد تا مزایایی مانند زمان پاسخگویی کمتر، مصرف باتری پایین‌تر و تحرک‌پذیری را برای کاربران به ارمغان بیاورد. از زمان معرفی رایانش لبه‌ای، یکی از چالش‌های مهم این حوزه، طراحی استراتژی‌های کارآمد برای تخلیه‌ی وظایف بوده است. علاوه بر این با رشد روزافزون صنعت اینترنت اشیاء، تعداد زیادی کاربرد نرم‌افزاری جدید در سطح شبکه به وجود آمده است که هر کدام دارای نیازمندی‌های محاسباتی و شبکه‌ای خاص خود می‌باشند. بنابراین یک ویژگی مهم در طراحی استراتژی‌های تخلیه‌ی وظیفه در رایانش لبه‌ای، در نظر گرفتن ناهمگونی کاربردها از نظر میزان منابع مورد نیاز است. در پروژه‌ی فعلی روشی برای بدست آوردن استراتژی تخلیه‌ی وظیفه‌ی تاخیر-کمینه تحت محدودیت توان مصرفی ارائه می‌دهیم. روش پیشنهادی شامل دو قسمت می‌باشد. در قسمت اول، سیستم تخلیه‌ی وظایف را با کمک زنجیره‌ی مارکوف گسسته-زمان مدل‌سازی می‌کنیم و در قسمت دوم، با استفاده از الگوریتمی مبتنی بر برنامه‌ریزی خطی، استراتژی تخلیه‌ی بهینه را برای مدل ساخته شده محاسبه می‌کنیم. علاوه بر تشریح و حل مسئله به صورت تئوری، چارچوب نرم‌افزاری جدیدی در زبان Kotlin ارائه می‌شود که می‌توان با استفاده از آن، استراتژی تخلیه‌ی بهینه را برای سیستم مورد نظر بدست آورد و عملکرد آن استراتژی را با کمک شبیه‌سازی بررسی کرد.

واژگان کلیدی: تخلیه‌ی وظیفه، رایانش لبه‌ای، زنجیره‌ی مارکوف، برنامه‌ریزی خطی، رایانش ابری

فهرست مطالب

فهرست تصاویر

فهرست جداول

فهرست علائم اختصاری

حالت دستگاه کاربر . . . . . τ

تعداد وظایف موجود در صف i-اُم . . . . . q_i

نرخ ورود وظیفه به صف i-اُم . . . . . α_i

احتمال ارسال موفق بسته . . . . . β

مجموعه‌ی تمام حالت‌های دستگاه کاربر . . . . . S

مجموعه‌ی تمام کنش‌های ممکن . . . . . A

کسری از وظایف نوع i که محلی اجرا می‌شوند . . . . . η_i

توان مصرفی لازم برای ارسال یک بسته . . . . . P_{tx}

توان مصرفی لازم برای اجرای محلی به اندازه‌ی یک بازه زمانی . . . . . P_{loc}

حداکثر توان مصرفی قابل قبول . . . . . P_{max}

تعداد بازه زمانی لازم برای پردازش محلی هر وظیفه‌ی نوع i . . . . . L_i

تعداد بازه زمانی لازم برای تخلیه‌ی هر وظیفه‌ی نوع i . . . . . M_i

تعداد بازه زمانی لازم برای پردازش هر وظیفه‌ی نوع i در سرور لبه‌ای . . . . . C_i

زمان اضافه‌ی لازم برای بازدریافت وظیفه از سرور لبه‌ای . . . . . t_{rx}

ظرفیت هر صف وظیفه . . . . . Q

تعداد قسمت اجرا شده از وظیفه‌ی تخصیص داده شده به پردازنده محلی . . . . . C_L

تعداد قسمت ارسال شده از وظیفه‌ی تخصیص داده شده به واحد ارسال . . . . . C_R

نوع وظیفه‌ی تخصیص داده شده به پردازنده محلی . . . . . T_L

نوع وظیفه‌ی تخصیص داده شده به واحد ارسال . . . . . T_R

طول هر بازه زمانی . . . . . Δ

احتمال حضور در حالت τ در توزیع پایدار زنجیره‌ی مارکوف . . . . . π_τ

احتمال گذر از حالت τ' به τ در زنجیره‌ی مارکوف . . . . . $\chi_{\tau',\tau}$

احتمال انتخاب کنش a در حالت τ در استراتژی g . . . . . g_τ^a

فصل اول

۱. مقدمه

افزایش روز افزون تعداد دستگاه‌های موجود در لبه‌ی شبکه در سال‌های اخیر، و همچنین معرفی کاربردهای نرم افزاری جدید که نیازمند منابع محاسباتی بالا هستند باعث شده است که تقاضای زیادی برای خدمات رایانش ابری بوجود بیاید. رایانش ابری این امکان را به دستگاه‌های هوشمند از جمله تلفن همراه و اینترنت اشیاء می‌دهد که بخشی از پردازش‌های سنگین خود را به سرورهای قدرتمند «تخلیه» کنند تا بر محدودیت‌های پردازشی خود غلبه کنند و کاربردهای نرم افزاری پیچیده‌ای مانند واقعیت افزوده و خودروهای هوشمند را برای کاربران فراهم سازند. با این وجود، پیاده‌سازی‌های سنتی رایانش ابری یک ایراد ذاتی دارند، و آن فاصله زیاد سرورهای ابری با دستگاه‌های پایانی است. الگوی «رایانش لبه‌ای» و معماری‌های استاندارد آن مانند رایانش لبه‌ای دسترسی-چندگانه که توسط سازمان ETSI ارائه‌شده است، سعی دارند تا با آوردن بخشی از منابع محاسباتی به نزدیکی لبه‌ی شبکه، این مشکل را تا حدی برطرف کنند.

Hii asdfj aksdfj kasjfd

علاوه بر تمایل دستگاه‌های لبه‌ی شبکه به کمتر شدن این فاصله و به عبارتی «کشش» منابع پردازشی توسط آن‌ها به منظور بهبود کیفیت سرویس، شرکت‌های ارائه‌دهنده‌ی خدمات ابری نیز تمایل دارند تا با «فشردن» بخشی از منابع پردازشی خود به لبه‌ی شبکه، بار محاسباتی و هزینه تجهیزاتی خود را کاهش دهند [1]. در جدول ۱-۱ مقایسه‌ای کلی از رایانش ابری و لبه‌ای ارائه‌شده است. با دقت در این جدول متوجه می‌شویم که رایانش لبه‌ای جایگزین رایانش ابری نیست بلکه مکمل آن است.

ویژگی	رایانش ابری	رایانش لبه‌ای
موضع‌گیری	مرکزی	توزیع شده
فاصله تا دستگاه کاربر	زیاد	کم
تاخیر	زیاد	کم
تقدم و تأخر	زیاد	کم
منابع پردازشی	فراوان	محدود
فضای ذخیره‌سازی	فراوان	محدود

جدول ۱-۱: مقایسه‌ی رایانش ابری و لبه‌ای

یک امر مهم در پیاده‌سازی کارآمد رایانش لبه‌ای، طراحی استراتژی‌های تخلیه‌ی وظیفه به صورت هوشمند و موثر است. این استراتژی‌ها نحوه‌ی تخصیص منابع توسط دستگاه کاربر[1] را مشخص می‌کنند و این امکان را به دستگاه کاربر می‌دهند تا درباره تخلیه یا عدم تخلیه‌ی وظایف محاسباتی در طول زمان تصمیم بگیرد. همچنین به استراتژی تخلیه‌ای که یک تابع «هدف» خاص را تحت شرایط محیطی مشخص، کمینه یا بیشینه کند «استراتژی تخلیه‌ی بهینه» می‌گوییم. توابع هدف بر حسب یک یا چند معیار سیستم تعریف می‌شوند. برخی از این معیارها عبارتند از:

- تاخیر سرویس[2]
- تقدم و تاخر[4]
- مصرف توان [3]
- هزینه

مقدار توابع هدف و شروط مسئله‌ی تخلیه، بستگی به پارامترهای زیادی دارند، از جمله میزان منابع موجود در دستگاه کاربر، نیازمندی‌های کاربر، کیفیت شبکه‌ی دسترسی و شلوغی سرورهای رایانش لبه‌ای. علاوه بر پارامترهای محیطی، ساختار کاربردهای نرم‌افزاری نیز در مسئله‌ی تخلیه‌ی وظیفه، تاثیر می‌گذارند. برای مثال ممکن است که تمام یا بخشی از یک کاربرد خاص قابلیت تخلیه نداشته باشد [2].

در پروژه‌ی فعلی **تاخیر سرویس** به عنوان تابع هدف در نظر گرفته شده است. تاخیر همواره یک

[1] User Equipment

[2] Service Delay

[3] Power Consumption

[4] Jitter

معیار اصلی در سنجش کیفیت سامانه‌های کامپیوتری بوده است. همچنین با رشد روز افزون صنعت اینترنت اشیاء، کاربردهای جدیدی در سطح شبکه به وجود آمده است که نیازمندی‌های تاخیر بسیار پایینی دارند، به طوری که سرورهای رایانش ابری پاسخگوی این نیازمندی نخواهند بود. و از طرفی نیازمندی‌های پردازشی بالا امکان اجرای این کاربردها به صورت محلی و بی‌درنگ را نمی‌دهد. یک نمونه از این دسته از کاربردها «سامانه مدیریت ترافیک هوشمند» است که نیازمند انجام پردازش‌های سنگین در زمان بسیار کم می‌باشد.

تاخیر سرویس بسته به اجرای محلی و یا تخلیه از مولفه‌های متفاوتی تشکیل می‌شود. در صورت اجرای محلی تاخیر سرویس از موارد زیر تشکیل خواهد بود:

۱. تاخیر انتظار در صف وظیفه d_q

۲. تاخیر اجرا به صورت محلی d_{loc}

و در صورت تخلیه از موارد زیر تشکیل خواهد شد:

۱. تاخیر صف d_q

۲. تاخیر ارسال d_{tx}

۳. تاخیر انتشار $d_{propagation}$

۴. تاخیر اجرا در سرور لبه‌ای d_{server}

۵. تاخیر بازدریافت وظیفه از سرور d_{rx}

در پروژه‌ی فعلی روشی برای بدست آوردن استراتژی تخلیه‌ی وظیفه‌ی تاخیر-کمینه[5] تحت محدودیت توان مصرفی ارائه می‌دهیم. روش ارائه‌شده، مبتنی بر زنجیره‌ی مارکوف گسسته-زمان و برنامه‌ریزی خطی می‌باشد و گسترشی بر روش ارائه‌شده در [3] می‌باشد. نوآوری و مزیت اصلی روش پیشنهادی نسبت به مقاله ذکر شده، قابلیت پشتیبانی از وظایف با نیازمندی‌های پردازشی و شبکه‌ای متفاوت

[5] Delay-optimal

(وظایف ناهمگون) می‌باشد. انگیزه‌ی اصلی از این گسترش، تنوع محاسباتی وظایف در محیط‌های اینترنت اشیاء بوده است. به طور مثال در بسیاری از پژوهش‌های حوزه تخلیه‌ی وظیفه در اینترنت اشیاء، وظایف به دو دسته «سبک» و «سنگین» تقسیم می‌شوند [4, 5]. برای درک مفهوم وظایف سبک و سنگین می‌توان مثال اتومبیل خودران را در نظر گرفت. در این کاربرد، وظیفه پردازش اطلاعات تصاویر به منظور راندن خودرو یک وظیفه سنگین محسوب می‌شود، در حالی که وظیفه روشن کردن سیستم گرمایشی خودرو بر حسب داده‌ی سنسور دما، یک وظیفه سبک محسوب می‌شود.

ادامه‌ی پروژه به پنج فصل تقسیم شده است. در فصل ۲ پژوهش‌های مرتبط انجام شده را مرور می‌کنیم. در فصل ۳ به شرح مسئله‌ی تخلیه‌ی وظیفه و ساختار رایانش لبه‌ای می‌پردازیم. در فصل ۴ روش پیشنهادی برای بدست آوردن استراتژی تخلیه‌ی بهینه را شرح می‌دهیم. در فصل ۵ نحوه‌ی پیاده‌سازی عملی روش پیشنهادی را در قالب چارچوب نرم‌افزاری جدیدی با نام «کامپیوت[۶]» ارائه می‌دهیم. در فصل ۶ با استفاده از چارچوبِ کامپیوت، به آزمایش و شبیه‌سازی روش پیشنهادی می‌پردازیم. در انتها در فصل ۷ یک جمع‌بندی کلی از تمامی مطالب ارائه می‌دهیم و پیشنهاداتی برای گسترش روش پیشنهادی عنوان می‌کنیم.

[6] Kompute

فصل دوم

۲. مروری بر ادبیات و کارهای انجام شده

پژوهش‌های انجام شده در زمینه تخلیه‌ی پردازش را می‌توان بر حسب «ویژگی‌های محیط مسئله» و همینطور «الگوریتم استفاده شده برای حل مسئله» دسته‌بندی کرد. در این فصل ابتدا به معرفی این ویژگی‌ها و الگوریتم‌ها می‌پردازیم و سپس برخی از مقالاتی که ارتباط نزدیکی با پروژه‌ی فعلی دارند را معرفی می‌کنیم.

۲-۱ بررسی مقالات از نظر ویژگی‌های محیط مسئله

در جدول ۲-۱ که برگرفته از [6] می‌باشد، برخی از ویژگی‌های محیط مسئله و حالت‌های ممکن برای این ویژگی‌ها مشاهده می‌شود که در ادامه به توضیح هر کدام از آنها می‌پردازیم.

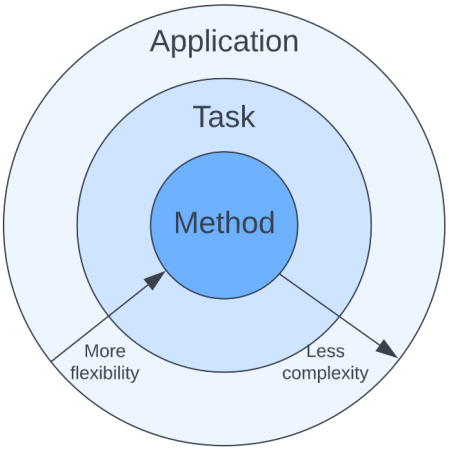
Granularity	User Count	Computation Node	Destination	Metric
Application	Single UE	Cloud Server	Single-server	Delay
Task	Multi UE	Edge Server	Multi-server	Energy
Method		Ad hoc		Cost

جدول ۲-۱: تقسیم‌بندی شرایط محیطی مسئله‌ی تخلیه‌ی پردازش

۲-۱-۱ دانه‌بندی

دانه‌بندی به نوع مولفه‌های پردازشی قابل‌تخلیه در سیستم اشاره دارد. طبق [6] دانه‌بندی را با سه دسته مختلف (به ترتیب از دانه‌ریز به دانه درشت) بیان می‌کنیم: کاربرد، وظیفه، شگرد. هر چه دانه‌بندی ریزتر باشد انعطاف‌پذیری سیستم تخلیه بیشتر خواهد بود، به طوری که به توسعه‌دهندگان نرم‌افزار اجازه خواهد داد تا به طور دقیق مشخص کنند که کدام قسمت‌ها از یک کاربرد خاص تخلیه شوند و کدام قسمت‌ها نشوند. با این حال پیاده‌سازی سیستم‌های تخلیه‌ی پردازش به صورت دانه‌ریز به

مراتب پیچیده‌تر است. پیاده‌سازی‌های دانه‌ریز همچنین سربار زمانی بیشتری برای ساخت محیط‌های مجازی در سرور دارند. به عنوان نمونه در [7] دانه‌بندی در سطح شگرد صورت گرفته است، در حالی که در [3] دانه‌بندی در سطح وظیفه صورت گرفته است. ما نیز در پروژه‌ی فعلی مسئله‌ی تخلیه‌ی پردازش را در سطح وظیفه حل کرده‌ایم. خلاصه‌ای از انواع دانه‌بندی‌ها در شکل ۲-۱ آورده شده است.



شکل ۲-۱: سه نوع دانه‌بندی مختلف در سامانه‌ی تخلیه‌ی پردازش

۲-۱-۲ تعداد کاربر و سرور

در برخی از مقالات مانند [3] مسئله‌ی تخلیه‌ی پردازش تنها برای یک کاربر در نظر گرفته می‌شود، در حالیکه در برخی از مقالات مانند [8] از چندین کاربر همزمان نیز پشتیبانی می‌شود. در پروژه‌ی فعلی تعداد کاربران را برای سادگی یک در نظر می‌گیریم. طبیعتا در حالت تک‌کاربر نیز سرور خدمت‌دهنده می‌تواند به کاربران مختلفی خدمت‌دهی کند. اما تفاوتی که در این حالت وجود دارد این است که برای هر کاربر محیط تخلیه‌ی پردازش جدایی در نظر گرفته می‌شود و کاربران متفاوت از وضعیت یکدیگر اطلاعی ندارند. بنابراین در مقالاتی که تخلیه‌ی پردازش برای چندین کاربر در نظر گرفته می‌شود، عموما یک چاچوبی برای ارتباط و همکاری بین دستگاه‌های مختلف در شبکه نیز در نظر گرفته می‌شود.

مشابه با تعداد کاربران، تعداد سرورهای پردازشی در سامانه تخلیه نیز می‌تواند یک یا بیشتر باشد. برای نمونه در [8] مسئله‌ی تخلیه‌ی پردازش برای چندین سرور بررسی شده است. در پروژه‌ی فعلی ما حالت تک‌سرور را در نظر می‌گیریم. افزایش تعداد سرورها عموما فضای حالت مسئله بهینه‌سازی

را بزرگ‌تر می‌کند، و حل آن را پیچیده‌تر.

۲-۱-۳ گره پردازشی

در رایانش توزیع‌شده به انجام پردازش توسط هر گره پردازشی به جز گره ایجادکننده‌ی آن پردازش، تخلیه‌ی پردازش گفته می‌شود. با این حال بسته به نوع گره خدمت‌دهنده، مسئله تخلیه‌ی پردازش می‌تواند ویژگی‌های بسیار متفاوتی داشته باشد. در ادبیات تخلیه‌ی پردازش، عموما سه معماری مختلف برای سامانه در نظر گرفته می‌شود:

۱. پردازش در سرور ابری

۲. پردازش در سرور لبه‌ای

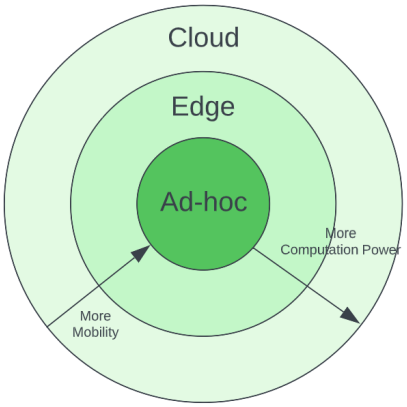
۳. پردازش در شبکه‌ای بدون ساختار[1] (از دستگاه‌های کاربر)

برای مثال در [3] گره‌های پردازشی سرورهای لبه‌ای در نظر گرفته شده است، در حالیکه در [9] مسئله‌ی تخلیه وظیفه در سطح رایانش ابری و به طور دقیق‌تر، «رایانش ابری متحرک[2]» بررسی شده است. در [10] گره‌های پردازشی پهبادهای متحرک هستند و مسئله‌ی تخلیه وظیفه در محیط «شبکه بدون ساختار متحرک[3]» در نظر گرفته شده است. نوع گره‌های پردازشی همچنین بر روی «تحرک‌پذیری» دستگاه‌های کاربر تاثیر می‌گذارد. به طور کلی هر چه فاصله گره‌های پردازشی نسبت به دستگاه کاربر کمتر باشد و تعداد گره‌های پردازشی در حومه دستگاه کاربر بیشتر باشد، تحرک پذیری افزایش می‌یابد (مطابق شکل ۲-۲).

[1] Ad-hoc

[2] Mobile Cloud Computing

[3] Mobile Ad-hoc Network



شکل ۲-۲: تحرک‌پذیری در سه نوع گره پردازشی مختلف

۲-۱-۴ معیار بهینه‌سازی

معیار بهینه‌سازی به کمیتی اشاره دارد که استراتژی تخلیه‌ی پردازش سعی در بهینه‌سازی آن دارد. برخی از معیارهای رایج عبارتند از: تاخیر، انرژی، کیفیت سرویس و هزینه. برای مثال در [3] معیار تاخیر، در [11] معیار انرژی و در [12] معیار هزینه در نظر گرفته شده است.

۲-۲ بررسی مقالات از نظر روش حل مسئله

در جدول ۲-۲ که برگرفته از [13] می‌باشد، یک دسته‌بندی کلی از الگوریتم‌های رایج برای حل مسئله‌ی تخلیه‌ی پردازش مشاهده می‌شود. در پروژه‌ی فعلی ما از **روشی** قطعی بر پایه‌ی برنامه‌ریزی خطی برای یافتن **استراتژی تخلیه** تصادفی استفاده می‌کنیم. برای آشنایی بیشتر با روش‌های حل مسئله تخلیه وظیفه به[6] و [13] رجوع شود.

Model	Examples
Stochastic	Machine learning, Generalized poison distribution, Game theory, Queuing theory, Markov processes, Gaussian processes
Deterministic	Some supervised Machine Learning approaches (e.g., KNN), Linear and non-linear programming, Linear regression equation

جدول ۲-۲: تقسیم‌بندی الگوریتم‌های حل مسئله‌ی تخلیه‌ی پردازش

۲-۳ پژوهش‌های مرتبط

در [3] مسئله‌ی تخلیه‌ی وظیفه‌ی تاخیر-کمینه با استفاده از روشی مبتنی بر زنجیره‌ی مارکوف و برنامه‌ریزی خطی حل شده است. در این پژوهش محیط تک کاربر و تک سرور در نظر گرفته شده است. نویسندگان این مقاله نشان می‌دهند که روش ارائه‌شده در درازمدت عملکرد بهینه دارد. با این حال روش ارائه شده توسط این مقاله چندین کاستی دارد، از جمله عدم پشتیبانی از وظایف با نیازمندی‌های پردازشی متفاوت و عدم پشتیبانی از موازی‌سازی. پروژه‌ی فعلی گسترشی بر این مقاله است.

در [14] مکانزیم تخلیه‌ی وظیفه‌ای با هزینه‌ی کمینه در محیط رایانش لبه‌ای متحرک ارائه‌شده است. محیط در نظر گرفته شده از نظر ثابت بودن طول بازه‌های زمانی و تفاوت وظایف و همچنین نحوه‌ی تعریف مسئله‌ی بهینه‌سازی، شبیه به پروژه‌ی فعلی می‌باشد. اما از نظر معیار و تعداد سرور متفاوت می‌باشد. روش بهینه‌سازی استفاده شده در پروژه‌ی فعلی روش «ضرایب لاگرانژ» می‌باشد که عملکرد سریعی دارد اما لزوما جواب بهینه سراسری را پیدا نمی‌کند و فقط جواب‌های بهینه محلی را پیدا می‌کند.

در [9] و [15] مشابه با پروژه‌ی فعلی ، ناهمگونی وظایف و تقابل[4] تاخیر و انرژی در نظر گرفته شده است. با این تفاوت که در این دو مقاله از روش بهینه‌سازی لیاپانوف استفاده شده است. همچنین این دو مقاله مسئله‌ی تخلیه‌ی وظیفه را در محیط رایانش ابری در نظر گرفته‌اند و نه رایانش لبه‌ای. علاوه بر این هیچ یک از این دو مقاله چارچوبی نرم‌افزاری برای حل مسئله در محیط‌های خاص ارائه نداده‌اند.

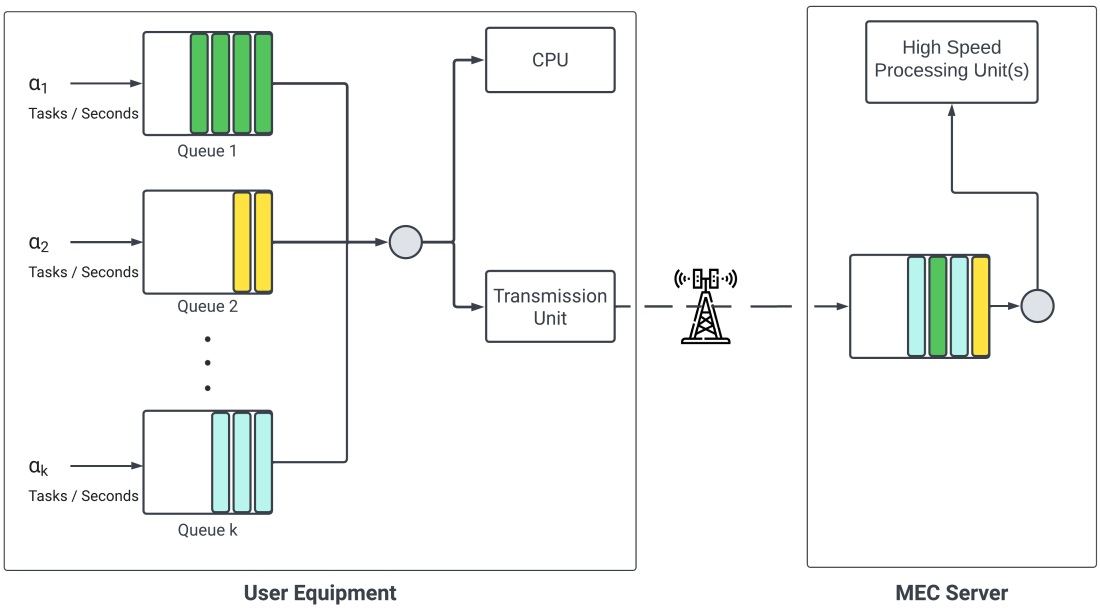
در [16] مسئله‌ی تخلیه و زمان‌بندی ارسال و اجرا به صورت همزمان در نظر گرفته شده است و کانال بیسیم به صورت تصادفی مدل شده است و از این ابعاد به پروژه‌ی فعلی شباهت دارد. بر خلاف پروژه فعلی معیار بهینه‌سازی در این پروژه انرژی مصرفی است. روش ارائه‌شده عملکرد خوبی دارد و به میزان قابل توجی در انرژی مصرفی صرفه‌جویی می‌کند. با این حال مدل در نظر گرفته شده کاستی‌هایی دارد. یک ایراد اصلی فرض وجود تنها یک کاربرد در سیستم است. به عبارت دیگر تاخیر ایجاد شده به واسطه انتظار کاربردها در صف در نظر گرفته نشده است.

[4] Tradeoff

فصل سوم

۳. شرح مسئله

در پروژه‌ی فعلی قصد داریم در یک سامانه‌ی رایانش لبه‌ای مطابق با شکل ۳-۱، استراتژی تخلیه‌ای بیابیم که تاخیر سرویس میانگین $\bar{T}$ را تحت محدودیت توان مصرفی P_{max} در درازمدت کمینه کند.



شکل ۳-۱: ساختار کلی سامانه‌ی تخلیه‌ی پردازش

همانطور که در شکل ۳-۱ مشاهده می‌شود، در سامانه مد نظر سه مولفه اصلی وجود دارد:

۱. دستگاه کاربر (User Equipment)

۲. سرور رایانش لبه‌ای چند-دسترسی (Multi-access Edge Computing Server)

۳. کانال بیسیم

در فصل جاری نحوه‌ی عملکرد هر کدام از این مولفه‌ها در قالب مدل‌های تئوری شرح داده می‌شود.

۳-۱ مدل وظایف

فرض می‌شود که k نوع وظیفه‌ی مختلف در سیستم رایانش لبه‌ای وجود دارد و به ازای هر نوع وظیفه دقیقا یک صف در سیستم وجود دارد. وظایف نوع i-اُم برای اجرا به صورت محلی احتیاج به L_i بازه زمانی پردازش توسط پردازنده دارند و به منظور تخلیه به سرور رایانش لبه‌ای احتیاج به M_i واحد زمانی ارسال توسط واحد ارسال[1] دارند. همچنین فرض می‌شود که وظایف نوع i-اُم در سرور رایانش لبه‌ای به C_i بازه زمانی پردازش توسط سرور نیاز دارند. برای سادگی بیشتر در ادامه‌ی پروژه‌ی فعلی برای اشاره به یک واحد زمانی پردازش توسط پردازنده از عبارت «**قسمت**»[2] استفاده می‌کنیم که انتزاعی از قسمت‌های کد اجرایی است. برای اشاره به یک واحد زمانی ارسال توسط واحد ارسال نیز از عبارت «**بسته**» استفاده می‌شود.

۳-۲ مدل دستگاه کاربر

دستگاه کاربر مطابق با شکل ۳-۱ شامل دو مولفه‌ی پردازنده و واحد ارسال می‌باشد. همچنین همانطور که اشاره شد k صف مختلف به ازای هر کدام از انواع وظایف در سیستم وجود دارد. ظرفیت هر صف را برابر با مقدار ثابت Q در نظر می‌گیریم. در هر بازه زمانی، پردازنده یا به اندازه‌ی یک قسمت پردازش انجام می‌دهد و یا بیکار[3] است. اجرای هر قسمت پردازش توسط پردازنده به میزان P_{loc} وات توان مصرف می‌کند. به طور مشابه واحد ارسال در هر بازه زمانی یا یک بسته را به شبکه ارسال می‌کند یا بیکار است. نکته قابل توجه در مورد واحد ارسال این است که با توجه به شرایط کانال بیسیم، در یک بازه زمانی خاص ممکن است ارسال موفقیت آمیز باشد یا نباشد. فرض می‌شود که ارسال موفقیت آمیز هر بسته به میزان P_{tx} وات توان مصرف می‌کند. توضیحات بیشتر در مورد نحوه‌ی کارکرد کانال بی‌سیم در بخش ۳-۴ آورده شده است.

[1] Transmission Unit

[2] Section

[3] Idle

با توجه به توضیحات داده شده می‌توان مدلی برای «حالت دستگاه کاربر»[۴] تعریف کرد. در [3] برای مشخص کردن حالت دستگاه در زمان t از یک سه تایی مانند $\boldsymbol{\tau}[t] = (q[t], c_T[t], c_L[t])$ استفاده شده است، که در آن $q[t]$ مشخص کننده تعداد وظایف موجود در صف وظایف، $c_T[t]$ مشخص کننده تعداد بسته ارسال شده از وظیفه‌ی تخصیص داده شده به واحد ارسال است، و $c_L[t]$ مشخص کننده تعداد قسمت اجرا شده از وظیفه تخصیص داده شده به پردازنده است. همچنین حالت $c_T[t] = 0$ معادل با بیکار بودن واحد ارسال و $c_L[t] = 0$ معادل با بیکار بودن پردازنده تعریف می‌شود. برای مثال سه تایی $(4, 2, 1)$ به این معنی است که ۴ وظیفه در صف وظایف وجود دارد، واحد پردازش در حال تخلیه‌ی وظیفه‌ای است و تا کنون یک بسته از آن وظیفه را ارسال کرده و به عنوان قدم بعدی باید بسته شماره ۲ را ارسال کند. پردازنده نیز در حال اجرای وظیفه‌ای به صورت محلی است و تا کنون یک قسمت از آن وظیفه را اجرا کرده است.

[4] User Equipment State

با این حال مدل فوق در مسئله‌ی تخلیه‌ی وظیفه‌ی با چند نوع وظیفه، قابل استفاده نیست و نیاز به تغییر دارد. ما در پروژه‌ی فعلی برای تعیین حالت دستگاه کاربر از یک چندتایی[5] به طول $k+4$ مطابق با رابطه‌ی ۳-۱.۲ استفاده می‌کنیم. در این رابطه‌ی متغیرهای $q_1[t], \cdots, q_k[t]$ تعداد وظایف موجود از هر نوع وظیفه در صف مربوطه را مشخص می‌کنند. متغیرهای $c_R[t]$ و $c_L[t]$ مشابه با حالت تک صف تعریف می‌شوند و به ترتیب وضعیت واحد ارسال و پردازنده را مشخص می‌کنند. دو متغیر جدید $T_R[t]$ و $T_L[t]$ به ترتیب مشخص کننده نوع وظیفه در حال ارسال توسط واحد ارسال و نوع وظیفه در حال اجرا توسط پردازنده اند.

$$\tau[t] = (q_1[t], q_2[t], \ldots, q_k[t], c_R[t], c_L[t], T_R[t], T_L[t]) \tag{۱.۲-۳}$$

در پروژه‌ی فعلی به منظور خوانایی بیشتر، چندتایی بیان شده در رابطه‌ی ۳-۱.۲ را به صورت زیر نیز نمایش می‌دهیم و این دو صورت معادل هم می‌باشند:

$$\tau[t] = ([q_1[t], q_2[t], \ldots, q_k[t]], c_R[t], c_L[t], T_R[t], T_L[t]) \tag{۲.۲-۳}$$

رابطه‌ی ۳-۳.۲ با تعریف شروط مختلف فضای حالت مسئله را توصیف می‌کند. **(نکته: در رابطه‌ی ۳-۳.۲ و سراسر پروژه‌ی فعلی منظور از $\tau\{X\}$ مقدار متغیر X در حالت τ است.)**

$$\begin{aligned}
&\forall \tau \in S, i \in \{1, 2, \ldots, k\} \quad 0 \leqslant \tau\{q_i\} \leqslant Q \\
&\forall \tau \in S \quad \tau\{T_L\}, \tau\{T_R\} \in \{0, 1, 2, \ldots, k\} \\
&\forall \tau \in \{\tau' \in S \mid \tau'\{T_R\} = 0\} \quad \tau\{C_R\} = 0 \\
&\forall \tau \in \{\tau' \in S \mid \tau'\{T_R\} \neq 0\} \quad 1 \leqslant \tau\{C_R\} \leqslant M_{\tau\{T_R\}} \\
&\forall \tau \in \{\tau' \in S \mid \tau'\{T_L\} = 0\} \quad \tau\{C_L\} = 0 \\
&\forall \tau \in \{\tau' \in S \mid \tau'\{T_L\} \neq 0\} \quad 1 \leqslant \tau\{C_L\} \leqslant L_{\tau\{T_L\}} - 1
\end{aligned} \tag{۳.۲-۳}$$

[5] Tuple

۳-۳ مدل زمان

وضعیت سیستم تخلیه‌ی وظیفه در فواصل زمانی با طول ثابت Δ میلی ثانیه بررسی می‌شود. به طور دقیق‌تر حالت دستگاه کاربر را در بازه زمانی t-اُم با $\tau[t]$ مشخص می‌کنیم، و حالت دستگاه در بازه زمانی $t+1$ را با $\tau[t+1]$ مشخص می‌کنیم و فاصله بین این دو بازه زمانی را برابر با Δ میلی‌ثانیه در نظر می‌گیریم. بررسی زمان به صورت واحدهای گسسته، به منظور ساده‌سازی مسئله و همچنین ایجاد گسترش‌پذیری روش ارائه‌شده به شرایط محیطی مختلف صورت گرفته است. در عمل یک مقدار قابل استفاده برای Δ طول بازه‌های زمانی شبکه‌ی دسترسی[۶] مورد نظر است. برای مثال در شبکه‌های LTE طول هر بازه زمانی ۰/۵ میلی‌ثانیه می‌باشد. [17]

۴-۳ مدل کانال بیسیم

در پروژه‌ی فعلی مشابه با [3] کانال بی‌سیم را به صورت تصادفی مدل می‌کنیم[۷] یکی از دلایل اصلی برای مدل‌سازی کانال به صورت تصادفی، وجود نویز و ناپایداری در ارتباطات بیسیم است. کانال بی‌سیم را با یک مدل ساده احتمالی دوجمله‌ای مدل می‌کنیم به این صورت که ارسال هر بسته توسط واحد ارسال با احتمال β موفقیت آمیز خواهد بود و با احتمال $1-\beta$ ناموفق خواهد بود. در عمل مقدار β با توجه به رابطه‌ی ۳-۴.۴ (رابطه‌ی شنون) محاسبه می‌شود، که در آن R مشخص کننده سایز هر بسته است، $r(t)$ مشخص کننده‌ی نرخ ارسال در زمان t ، B پهنای باند سیستم، $\gamma[t]$ مقدار بهره‌ی کانال[۸] و N_0 مشخص کننده اندازه‌ی نویز کانال است.

$$\begin{aligned} \beta &= P(r(t) \geq R) \\ r(t) &= B \log_r \left(1 + \frac{\gamma[t] P_{\text{tx}}}{N_0 B}\right) \end{aligned} \qquad (۳-۴.۴)$$

[6] Access Network

[7] Stochastic Channel

[8] Channel Gain

۳-۵ مفهوم کنش

یک استراتژی تخلیه در هر بازه زمانی مانند t می‌بایست یک کنش[9] مانند $v[t]$ را برای اجرا توسط دستگاه کاربر انتخاب کند. اجرای هر کنش می‌تواند حالت دستگاه کاربر را تغییر دهد. برای درک بهتر مفهوم کنش، ابتدا مشابه با [3] حالتی را در نظر می‌گیریم که تنها یک صف (یک نوع وظیفه) در سیستم وجود داشته باشد. در این حالت می‌توانیم مجموعه‌ی کنش‌ها را با چهار عضو مطابق جدول ۳-۱ مشخص کنیم.

ID	Transmit	Local Execution	Description
1	False	False	No operation
2	False	True	Add to CPU
3	True	False	Add to Transmission Unit
4	True	True	Add to both units

جدول ۳-۱: لیست کنش‌ها در سیستمی با یک صف وظیفه

به طور مشابه در شرایطی که بیش از یک نوع وظیفه در سیستم وجود داشته باشد مجموعه‌ی کنش‌های ممکن مطابق با جدول ۳-۲ بدست می‌آید.

ID	Transmit	Local Execution	Description	Count
$\{1\}$	False	False	No operation	1
$\{2, ..., k+1\}$	False	True	Add to CPU	k
$\{k+2, ..., 2k+1\}$	True	False	Add to Transmission Unit	k
$\{2k+2, ..., 2k+k*k-1\}$	True	True	Add to both units	k^2

جدول ۳-۲: تقسیم‌بندی کنش‌ها در سیستمی با k صف وظیفه

[9] Action

اجرای هر کنش ممکن است که حالت سیستم را تغییر دهد. به طور مثال با اجرای کنشی از نوع Add To CPU یک وظیفه از صف انتخاب‌شده برداشته می‌شود، بنابراین طول صف به صورت $q_i[t+1] = q_i[t] - 1$ تغییر می‌کند. با اجرای این کنش همچنین وضعیت پردازنده از $c_L[t] = 0$ یعنی حالت بیکار به $c_L[t+1] = 1$ تغییر می‌کند زیرا قسمت اول وظیفه‌ی مربوطه در بازه زمانی t انجام خواهد شد. به طور مشابه برای سایر کنش‌ها نیز میتوان توابع انتقال[10] مشخصی تعریف کرد که با گرفتن یک حالت ورودی، حالت خروجی را محاسبه نماید. به دلیل پیچیدگی روابط این توابع، از توضیح بیشتر در این بخش صرف نظر شده است. برای مشاهده منطق دقیق این توابع در قالب کد، به پیوست ۱ مراجعه شود.

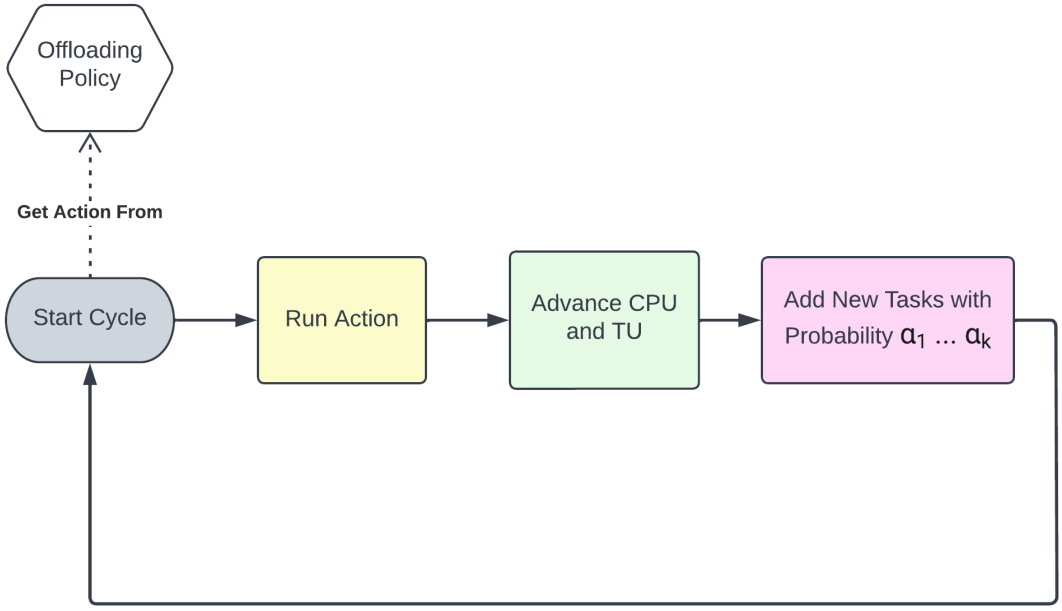
۳-۶ استراتژی تخلیه‌ی وظیفه

استراتژی تخلیه‌ی وظیفه در هر بازه زمانی تصمیم می‌گیرد که دستگاه کاربر چه کنشی را اجرا کند. بنابراین استراتژی تخلیه، یک تابع مانند $G(\tau)$ می‌باشد که با گرفتن حالت دستگاه کاربر $\tau[t]$ به عنوان ورودی، یک کنش مانند a را به عنوان خروجی می‌دهد. لازم به ذکر است که در اینجا این تابع را به صورت مفهومی انتزاعی در نظر می‌گیریم و در فصل‌های آتی به طور دقیق به نحوه‌ی بدست آوردن تابع بهینه $g(\tau)^*$ می‌پردازیم.

[10] Transition Function

۳-۷ روند فعالیت سیستم تخلیه‌ی وظیفه

نحوه‌ی عملکرد دستگاه کاربر در هر بازه زمانی مطابق با فرآیند مشخص شده در شکل ۳-۲ می‌باشد. در هر باز، دستگاه کاربر ابتدا کنش اجرایی را از یک استراتژی تخلیه دریافت می‌کند. سپس کنش انتخاب شده توسط دستگاه کاربر اجرا خواهد شد که ممکن است منجر به تغییر حالت دستگاه شود. سپس پردازنده و واحد ارسال هر کدام در صورت فعال بودن به اندازه‌ی یک بازه زمانی فعالیت خواهند کرد. در انتها وظایف جدید با احتمالات $\alpha_1, \cdots, \alpha_k$ به صف‌های وظایف اضافه خواهند شد.



شکل ۳-۲: روند فعالیت دستگاه کاربر

فصل چهارم

۴. روش پیشنهادی

در این فصل الگوریتمی ارائه می‌دهیم که با استفاده از آن می‌توان مسئله‌ی یافتن استراتژی تخلیه با تاخیر کمینه را که در فصل قبل تشریح شد، حل کرد. استراتژی خروجی توسط الگوریتم، از نوع تصادفی می‌باشد و برای بدست آوردن آن، از روشی مبتنی بر زنجیره‌ی مارکوف و برنامه‌ریزی خطی استفاده می‌کنیم.

۴-۱ استراتژی تخلیه‌ی وظیفه‌ی تصادفی

با استفاده از مدل‌های توصیف شده در فصل قبل می‌توانیم یک تعریف ریاضی از «استراتژی تخلیه وظیفه‌ی تصادفی» داشته باشیم. مشابه با مقاله [3] استراتژی تخلیه‌ی تصادفی را به صورت یک توزیع احتمالی مانند g_τ^a بر روی مجموعه‌ی $S \times A$ تعریف می‌کنیم. در اینجا عبارت $S \times A$ نمایانگر ضرب دکارتی مجموعه‌ی تمام حالت‌های سیستم در مجموعه‌ی تمام کنش‌های ممکن در سیستم است. یک نکته قابل توجه این است که برخی از دوتایی‌های حاصل از این ضرب دکارتی هیچ‌گاه در واقعیت امکان‌پذیر نیست. برای مثال در حالتی که صف خالی باشد، تنها یک کنش امکان‌پذیر است و آن هم کنش شماره ۱ (No Operation) است. با این حال برای سادگی در توضیح روش حل مسئله، این دوتایی‌ها را نیز در دامنه تابع توزیع احتمالی استراتژی تخلیه در نظر می‌گیریم تا همواره تعداد اعضای دامنه‌ی تابع توزیع احتمال برابر با $|S| \cdot |A|$ باشد.

همچنین طبق تعریف توزیع احتمال، رابطه‌ی ۴-۱.۱ باید برای هر استراتژی تخلیه تصادفی برقرار باشد.

$$\sum_{\tau \in S} \sum_{a \in A} g_\tau^a = 1 \qquad (۴-۱.۱)$$

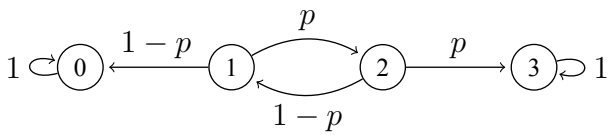
۴-۲ مدل زنجیره‌ی مارکوف دستگاه کاربر

در این قسمت ابتدا مدل آماری زنجیره‌ی مارکوف گسسته-زمان را معرفی می‌کنیم و سپس توضیح می‌دهیم که چگونه می‌توان با استفاده از این مدل معیارهای تاخیر و توان مصرفی میانگین را برای یک سیستم تخلیه‌ی وظیفه محاسبه کرد.

تعریف ۱.۴. دنباله‌ای از متغیرهای تصادفی $X_1, X_2, \ldots$ را که احتمال تغییر وضعیت از زمان t به $t+1$ مستقل از وضعیت‌های قبلی باشد را یک **زنجیره‌ی مارکوف گسسته-زمان** می‌نامند. این گزاره را به بیان متغیرهای تصادفی و تابع احتمال به صورت رابطه‌ی زیر نشان می‌دهیم.

$$\Pr\left(X_{t+1} = x \mid X_1 = x_1, X_2 = x_2, \ldots, X_n = x_t\right) = \Pr\left(X_{t+1} = x \mid X_t = x_t\right)$$

زنجیره‌ی مارکوف گسسته-زمان را می‌توان با گراف جهت‌دار نیز نمایش داد. در شکل ۴-۱ یک زنجیره‌ی نمونه مشاهده می‌شود.



شکل ۴-۱: یک زنجیره‌ی مارکوف نمونه برای مسئله‌ی پاکباختگی قمارباز[1]

[1] The Gambler's ruin

تعریف ۲.۴. زنجیره‌ی مارکوف گسسته-زمان $X(t)$ را **همگن-زمان** می‌گوییم اگر شرط زیر همواره برقرار باشد:

$$P\left(X_{n+1}=j \mid X_n=i\right)=P\left(X_1=j \mid X_0=i\right)$$

به عبارت دیگر یعنی احتمالات مربوط به انتقال بین حالت‌ها به زمان t وابسته نیستند. در این حالت احتمال انتقال زنجیره از حالت i به j را با عبارت $p_{ij}=P\left(X_1=j \mid X_0=i\right)$ نمایش می‌دهیم و همچنین ماتریس انتقال را با $P=(p_{ij})$ نمایش می‌دهیم.

طبق تعاریف ۱.۴ و ۲.۴ می‌توان حالت دستگاه کاربر در طی زمان را به صورت یک زنجیره‌ی مارکوف گسسته‌زمان در نظر گرفت به طوری که $\tau[t]$ حالت زنجیره در زمان t را مشخص می‌کند. همچنین ماتریس انتقال χ را اینگونه تعریف می‌کنیم که $\chi_{\tau,\tau'}$ احتمال انتقال از حالت τ به τ' را مشخص می‌کند. این ماتریس انتقال به ازای یک استراتژی تخلیه‌ی داده شده و پارامترهای سیستمی مشخص، قابل محاسبه می‌باشد. به طور دقیق‌تر احتمال انتقال از حالتی مانند τ به τ' بستگی به مقادیر زیر دارد:

- استراتژی تخلیه g_τ^a
- احتمال ورود وظایف $\alpha_1, \cdots, \alpha_k$
- احتمال موفقیت واحد ارسال β

برای نمونه در سیستم تخلیه‌ی وظیفه‌ای با ویژگی‌های مشخص شده در جدول ۴-۱ احتمال انتقال به حالات بعدی مطابق با جدول ۴-۲ بدست می‌آید. برای سادگی در اینجا از توضیح بیشتر در مورد نحوه‌ی محاسبه‌ی مقادیر درایه‌های ماتریس انتقال صرف نظر می‌کنیم. برای آگاهی از نحوه‌ی محاسبه‌ی این مقادیر در قالب کد به پیوست ۳ رجوع شود.

Parameter	M_1	M_2	L_1	L_2	C_1	C_2	β	P_{tx}	P_{loc}	P_{max}	t_{rx}
Value	1	3	7	2	1	1	0.95	1	0.8	1.6	0

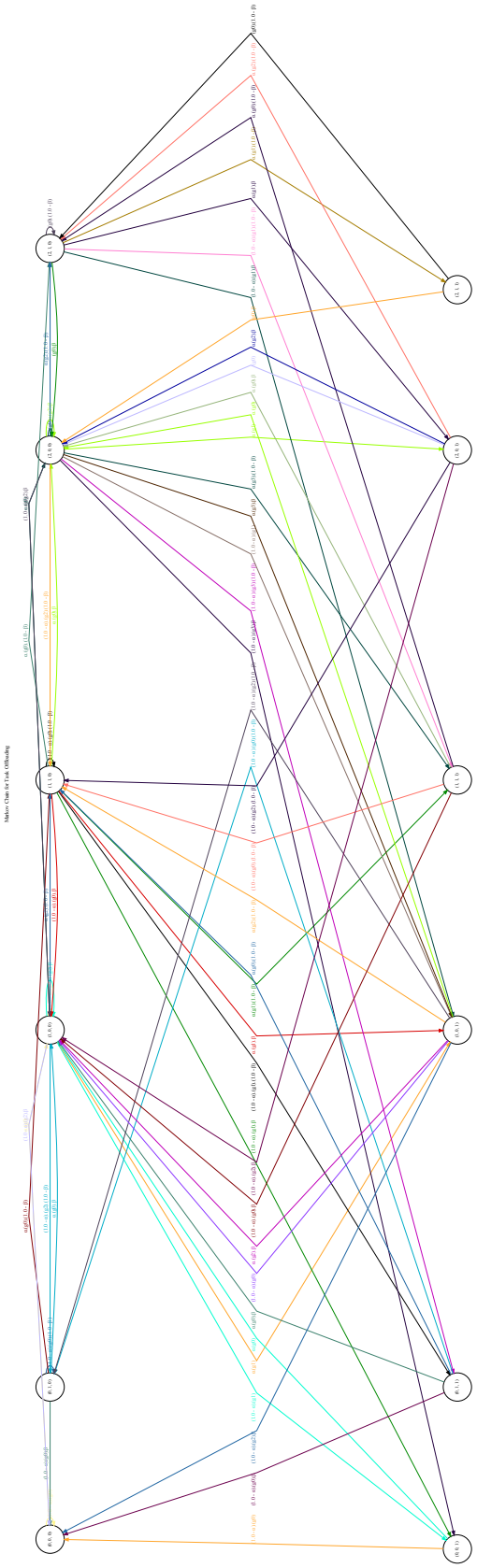
جدول ۴-۱: پارامترهای محیط رایانش لبه‌ای در سناریوی دو صف با یک صف ثابت

τ'	$\chi_{\tau,\tau'}$
$([1,1],0,2,0,1)$	$g_\tau^{NoOperation} * (1-\alpha_1) * (1-\alpha_2)$
$([2,1],0,2,0,1)$	$g_\tau^{NoOperation} * \alpha_1 * (1-\alpha_2)$
$([1,2],0,2,0,1)$	$g_\tau^{NoOperation} * (1-\alpha_1) * \alpha_2$
$([2,2],0,2,0,1)$	$g_\tau^{NoOperation} * \alpha_1 * \alpha_2$
$([0,1],0,2,0,1)$	$g_\tau^{AddToTU(1)} * \beta * (1-\alpha_1) * (1-\alpha_2)$
$([0,1],1,2,1,1)$	$g_\tau^{AddToTU(1)} * (1-\beta) * (1-\alpha_1) * (1-\alpha_2)$
$([1,1],0,2,0,1)$	$g_\tau^{AddToTU(1)} * \beta * \alpha_1 * (1-\alpha_2)$
$([1,1],1,2,1,1)$	$g_\tau^{AddToTU(1)} * (1-\beta) * \alpha_1 * (1-\alpha_2)$
$([0,2],0,2,0,1)$	$g_\tau^{AddToTU(1)} * \beta * (1-\alpha_1) * \alpha_2$
$([0,2],1,2,1,1)$	$g_\tau^{AddToTU(1)} * (1-\beta) * (1-\alpha_1) * \alpha_2$
$([1,2],0,2,0,1)$	$g_\tau^{AddToTU(1)} * \beta * \alpha_1 * \alpha_2$
$([1,2],1,2,1,1)$	$g_\tau^{AddToTU(1)} * (1-\beta) * \alpha_1 * \alpha_2$
$([1,0],2,2,2,1)$	$g_\tau^{AddToTU(2)} * \beta * (1-\alpha_1) * (1-\alpha_2)$
$([1,0],1,2,2,1)$	$g_\tau^{AddToTU(2)} * (1-\beta) * (1-\alpha_1) * (1-\alpha_2)$
$([2,0],2,2,2,1)$	$g_\tau^{AddToTU(2)} * \beta * \alpha_1 * (1-\alpha_2)$
$([2,0],1,2,2,1)$	$g_\tau^{AddToTU(2)} * (1-\beta) * \alpha_1 * (1-\alpha_2)$
$([1,1],2,2,2,1)$	$g_\tau^{AddToTU(2)} * \beta * (1-\alpha_1) * \alpha_2$
$([1,1],1,2,2,1)$	$g_\tau^{AddToTU(2)} * (1-\beta) * (1-\alpha_1) * \alpha_2$
$([2,1],2,2,2,1)$	$g_\tau^{AddToTU(2)} * \beta * \alpha_1 * \alpha_2$
$([2,1],1,2,2,1)$	$g_\tau^{AddToTU(2)} * (1-\beta) * \alpha_1 * \alpha_2$

جدول ۴-۲: مقادیر ماتریس انتقال $\chi_{\tau,\tau'}$ در صورت حضور در حالت $\tau = ([1,1],0,1,0,1)$

در نمایش زنجیره‌ی مارکوف به صورت گراف، هر درایه ماتریس انتقال مانند $p_{i,j}$ معادل یک یال جهت‌دار از راس i به راس j با وزن $p_{i,j}$ می‌باشد. بنابراین می‌توان گفت که جدول ۴-۲ یال‌های گراف از راس مبدا $\tau = ([1,1],0,1,0,1)$ را مشخص می‌کند. در شکل ۴-۲ گراف جهت‌دار متناظر با سیستم تخلیه‌ی وظیفه‌ی نمونه‌ای رسم شده است که در آن یک صف وظیفه وجود دارد، $Q = 2$ و هر وظیفه دو قسمت و یک بسته دارد.[۲] با توجه به اینکه تنها یک نوع وظیفه وجود دارد، از متغیرهای T_L و T_R در فضای حالت صرف نظر شده است. در این زنجیره، سه‌تایی (x,y,z) بیانگر حالتی است که در آن x وظیفه در صف وجود دارد، واحد ارسال در وضعیت y قرار دارد و پردازنده z قسمت از وظیفه‌ی تخصیص داده شده به خودش را تاکنون اجرا کرده است.

[۲] کد استفاده شده برای رسم این گراف در آدرس https://github.com/dalisyron/OffloadingVisualizer موجود می‌باشد



شکل ۴-۲: زنجیره‌ی مارکوف سیستم تخلیه در قالب گراف جهت‌دار (برای مشاهده جزئیات زوم کنید)

۴-۳ محاسبه‌ی تاخیر و توان میانگین با کمک توزیع پایدار

به منظور محاسبه‌ی معیارهای توان مصرفی میانگین و تاخیر سرویس میانگین لازم است که بتوانیم درباره وضعیت سیستم تخلیه‌ی وظیفه در طولانی‌مدت استنتاج کنیم. در همین راستا مفهوم **توزیع پایدار** را برای زنجیره‌ی مارکوف تعریف می‌کنیم.

تعریف ۳.۴. توزیع احتمالی مانند p_i را یک **توزیع پایدار** برای زنجیره‌ی مارکوف با ماتریس انتقال P می‌گوییم هر گاه شرط زیر در آن برقرار باشد:

$$\pi = \pi P \quad \Longleftrightarrow \quad \pi_j = \sum_i \pi_i P_{ij} \quad \forall j.$$

یک سوالی که ممکن است بوجود بیاید این است که آیا هر زنجیره‌ی مارکوف گسسته‌زمانی توزیع پایدار دارد؟ برای پاسخ به این سوال لازم است دو مفهوم زنجیره‌ی مارکوف تقلیل‌ناپذیر و غیرمتناوب را تعریف کنیم.

تعریف ۴.۴. اگر رسیدن از هر نقطه به نقطه دیگر از فضای حالت با احتمال مثبت در زنجیره‌ی مارکوف میسر باشد، زنجیره را **تقلیل‌ناپذیر** گویند. به بیان ریاضی می‌توان تقلیل‌ناپذیر بودن زنجیره‌ی مارکوف را به صورت زیر نشان داد.

$$\Pr\left(X_{n_{ij}} = j \mid X_0 = i\right) = p_{ij}^{(n_{ij})} > 0$$

تعریف ۵.۴. تناوب $d(i)$ برای حالت i به صورت $d(i) = \gcd\{n : P_{ii}^n > 0\}$ تعریف می‌شود، که به معنی بزرگ‌ترین مقسوم علیه مشترک تعداد مراحل ممکن است به صورتی که از i شروع کرده و به i برگردیم. یک زنجیره‌ی مارکوف تقلیل‌ناپذیر را متناوب با تناوب d می‌گوییم اگر تمامی حالت‌ها تناوبی برابر با $d > 1$ را داشته باشند. یک زنجیره‌ی مارکوف تقلیل‌ناپذیر را **غیرمتناوب** می‌گوییم اگر تمامی حالت‌ها تناوب برابر با ۱ داشته باشند.

قضیه ۱.۴. (همگرایی) هر زنجیره‌ی مارکوف تقلیل‌ناپذیر و غیر متناوب دارای توزیع پایدار منحصر به فردی مانند π می‌باشد.

حال با استفاده از قضیه ۱.۴ ثابت می‌کنیم که زنجیره‌ی مارکوف سیستم تخلیه‌ی وظیفه، دارای توزیع پایدار منحصر به فرد است. برای سادگی فرض می‌کنیم که سامانه یک صف دارد و سپس نحوه‌ی بسط نتیجه به چندین صف را توضیح می‌دهیم.

قضیه ۲.۴. زنجیره‌ی مارکوف مربوط به سیستم تخلیه تک صف تقلیل‌ناپذیر است.

اثبات:

قسمت الف) با توجه به تعریف سیستم تخلیه می‌دانیم که از هر حالت غیر شروع مانند $(x, y, z) \neq (0, 0, 0)$ می‌توان به حالت شروع رفت. به این منظور کافی است که تمام وظایف داخل صف به نحوی (اجرا یا ارسال) به اتمام برسند و وظیفه‌ی جدیدی نیز در این حین وارد سیستم نشود.

قسمت ب) همچنین می‌توان ثابت کرد که از حالت شروع $(0, 0, 0)$ می‌توان به هر حالت دیگر (x, y, z) رفت. به این منظور دنباله رخدادهای زیر را در نظر بگیرید:

۱. ورود x وظیفه‌ی جدید

۲. انتقال یک وظیفه به واحد ارسال و ورود یک وظیفه جدید، هر دو در صورتی که $y > 0$

۳. پیشرفت واحد ارسال به مدت y سیکل و عدم ورود وظیفه‌ی جدید در این حین

۴. انتقال یک وظیفه به پردازنده و ورود یک وظیفه جدید، هر دو در صورتی که $z > 0$

۵. پیشرفت واحد ارسال به مدت z سیکل و عدم ورود وظیفه‌ی جدید در این حین

با توجه به نتایج بخش الف و ب می‌توان نتیجه گرفت که از گشتی با احتمال مثبت از هر حالت به حالت دیگر وجود دارد بنابراین طبق تعریف زنجیره تقلیل‌ناپذیر است.

قضیه ۳.۴. زنجیره‌ی مارکوف مربوط به سیستم تخلیه تک صف غیر متناوب است.

اثبات:

به منظور اثبات این قضیه فقط کافی است که به این نکته توجه کنیم که حالت $(0,0,0)$ دارای تناوب یک می‌باشد زیرا با احتمالی مثبت (متناظر با رخداد عدم ورود وظیفه و کنش *No Operation*) می‌توان در همان حالت ماند. با توجه به همین نکته و تقلیل‌ناپذیر بودن زنجیره میتوانیم نتیجه بگیریم که سایر حالت‌ها نیز باید تناوب یک داشته باشند. بنابراین زنجیره غیرمتناوب است.

با توجه به قضایای ۲.۴ و ۳.۴ و قضیه همگرایی می‌توان نتیجه گرفت که زنجیره‌ی مارکوف سیستم تخلیه تک صف دارای توزیع پایدار منحصر به فرد می مطابق با رابطه‌ی ۴-۲.۳ می‌باشد. برای بسط این اثبات به حالت چند صف اثبات غیرمتناوب بودن یکسان خواهد بود و در اثبات تقلیل‌ناپذیر بودن، رخداد اول به ورود $x_1, \cdots, x_k$ وظیفه از انواع مختلف تغییر پیدا می‌کند.

$$\begin{cases} \sum_{\boldsymbol{\tau}' \in \mathcal{S}} \chi_{\boldsymbol{\tau}', \boldsymbol{\tau}} \pi_{\boldsymbol{\tau}'} = \pi_{\boldsymbol{\tau}}, \forall \boldsymbol{\tau} \in \mathcal{S} \\ \sum_{\boldsymbol{\tau} \in \mathcal{S}} \pi_{\boldsymbol{\tau}} = 1 \end{cases} \tag{۲.۳-۴}$$

۴-۴ محاسبه‌ی تاخیر میانگین

تاخیر هر وظیفه، شامل تاخیر انتظار در صف وظایف و تاخیر پردازش می‌باشد. به منظور بدست آوردن تاخیر میانگین سیستم ابتدا θ_i را به عنوان کسری از وظایف سیستم در طولانی مدت که از نوع i هستند تعریف می‌کنیم. اگر طول صف‌ها به مقدار کافی بزرگ باشد و همچنین استراتژی تخلیه‌ای داشته باشیم که منجر به پر شدن صف و اتلاف وظیفه[3] نشود مقدار θ_i طبق رابطه‌ی ۴-۳.۴ بدست می‌آید.

$$\theta_i = \frac{\alpha_i}{\sum_{j=1}^{k} \alpha_j} \tag{۳.۴-۴}$$

[3] Task Loss

پارامتر t_q^i را برابر با مقدار میانگین تاخیر انتظار در صف مربوط به وظایف نوع i تعریف می‌کنیم. طبق قانون لیتِل[۴] می‌توان مقدار این تاخیر را بر اساس رابطه‌ی ۴-۴.۴ بدست آورد. همانطور که پیش‌تر ذکر شد برای برقراری این رابطه لازم است که اتلاف وظیفه در صف رخ ندهد. به عبارت دیگر فقط با فرض اینکه استراتژی تخلیه‌ی ارائه‌شده «کارامد» باشد رابطه ۴-۴.۴ برقرار است. در پیاده‌سازی عملی، محدودیت «کارآمد» بودن یک استراتژی بدین گونه تعریف شده است که احتمال پر بودن صف مقداری ناچیز باشد.

$$t_q^i = \frac{\theta_i}{\alpha_i} \sum_{j=0}^{Q} i \cdot \Pr\{q_i[t] = i\} = \frac{1}{\alpha} \sum_{\tau \in S} \tau\{q_i\} \cdot \pi_\tau \qquad (۴.۴-۴)$$

همچنین t_{tx}^i را به عنوان تاخیر ارسال میانگین یک وظیفه از نوع i توسط واحد ارسال تعریف می‌کنیم که مقدار آن بر اساس امید ریاضی موفقیت در فرآیند برنولی مطابق با رابطه‌ی ۵-۴.۴ بدست می‌آید.

$$t_{tx}^i = M_i \sum_{j=1}^{\infty} j(1-\beta)^{(j-1)}\beta \qquad (۵.۴-۴)$$

به یاد داریم که مقدار تاخیر در صورت پردازش محلی برای وظایف نوع i برابر L_i می‌باشد. تاخیر اجرا در صورت تخلیه‌ی وظیفه به صورت مجموع زمان ارسال وظیفه t_{tx}^i زمان اجرا در سرور لبه‌ای C_i و تاخیر دریافت نتیجه از سرور t_{rx}^i محاسبه می‌شود.

$$t_c^i = t_{tx}^i + C_i + t_{rx} \qquad (۶.۴-۴)$$

در نتیجه می‌توان تاخیر اجرای میانگین وظایف نوع i را نیز مطابق رابطه‌ی ۷-۴.۴ بیان کرد.

$$t_p^i = \eta_i L_i + (1-\eta_i) t_c^i \qquad (۷.۴-۴)$$

که در آن η_i بیانگر کسری از وظایف نوع i می‌باشد که در طولانی‌مدت به صورت محلی اجرا می‌شوند

[4] Little's Law

و مطابق با رابطه‌ی ۴-۸.۴ بدست می‌آید.

$$\eta_i = \frac{\sum_{\boldsymbol{\tau},\boldsymbol{a}\in\mathcal{S}_1^i\cup\mathcal{S}_3^i\cup\mathcal{S}_5^i} \pi_{\boldsymbol{\tau}} g_{\boldsymbol{\tau}}^a}{\sum_{\boldsymbol{\tau},\boldsymbol{a}\in\mathcal{S}_1^i\cup\mathcal{S}_2^i\cup\mathcal{S}_3^i\cup\mathcal{S}_4^i} \pi_{\boldsymbol{\tau}} g_{\boldsymbol{\tau}}^a + 2\sum_{\boldsymbol{\tau},\boldsymbol{a}\in S_5^i} \pi_{\boldsymbol{\tau}} g_{\boldsymbol{\tau}}^a} \tag{۴-۸.۴}$$

که در آن $S_1^i, \cdots, S_5^i$ به صورت زیر تعریف می‌شوند:

(۴-۹.۴)

$$S_1^i = \{\boldsymbol{\tau}, \boldsymbol{a} \in \mathcal{S} \times A | type(a) = AddToCPU \wedge cpuQueue(a) = i\}$$

$$S_2^i = \{\boldsymbol{\tau}, \boldsymbol{a} \in \mathcal{S} \times A | type(a) = AddToTU \wedge tuQueue(a) = i\}$$

$$S_3^i = \{\boldsymbol{\tau}, \boldsymbol{a} \in \mathcal{S} \times A | type(a) = AddToBoth \wedge cpuQueue(a) = i \wedge tuQueue(a) \neq i\}$$

$$S_4^i = \{\boldsymbol{\tau}, \boldsymbol{a} \in \mathcal{S} \times A | type(a) = AddToBoth \wedge cpuQueue(a) \neq i \wedge tuQueue(a) = i\}$$

$$S_5^i = \{\boldsymbol{\tau}, \boldsymbol{a} \in \mathcal{S} \times A | type(a) = AddToBoth \wedge cpuQueue(a) = i \wedge tuQueue(a) = i\}$$

در رابطه‌ی فوق تابع $type(a)$ نوع کنش را مشخص می‌کند و یکی از چهار نوع بیان شده در بخش ۳-۵ می‌باشد. توابع $cpuQueue(a)$ و $tuQueue(a)$ نیز نوع وظیفه‌ی مربوط به کنش a را مشخص می‌کنند.

با استفاده از روابط بالا همچنین می‌توانیم میانگین تاخیر سرویس هر وظیفه در سیستم را طبق رابطه‌ی ۴-۱۰.۴ محاسبه کنیم. رابطه‌ی بدست آمده برای $\bar{T}$ همچنین مشخص کننده تابع هدف در مسئله‌ی پیدا کردن استراتژی تخلیه‌ی بهینه می‌باشد.

$$\bar{T} = \sum_{i=1}^{k} \theta_i \left(t_q^i + t_p^i\right) \tag{۴-۱۰.۴}$$

۴-۵ توان مصرفی میانگین

اگر پارامتر μ_τ^{loc} و μ_τ^{tx} را به ترتیب به عنوان احتمال فعالیت پردازنده در حالت τ و احتمال وجود درخواست ارسال وظیفه در حالت τ تعریف کنیم، و v_{loc} و v_{tx} را به صورت مجموع این دو پارامتر در فضای حالت مسئله تعریف کنیم، آنگاه توان مصرفی میانگین طبق رابطه‌ی زیر بدست می‌آید:

$$\begin{aligned}\bar{P} &= \sum_{\boldsymbol{\tau}\in\mathcal{S}} \pi_{\boldsymbol{\tau}} \left(\mu_{\boldsymbol{\tau}}^{loc} P_{loc} + \beta \mu_{\boldsymbol{\tau}}^{tx} P_{tx}\right) \\ &= \sum_{\boldsymbol{\tau}\in\mathcal{S}} \pi_{\boldsymbol{\tau}} \left(\mu_{\boldsymbol{\tau}}^{loc} P_{loc}\right) + \sum_{\boldsymbol{\tau}\in\mathcal{S}} \pi_{\boldsymbol{\tau}} \left(\beta \mu_{\boldsymbol{\tau}}^{tx} P_{tx}\right) \\ &= v_{loc} P_{loc} + \beta v_{tx} P_{tx}\end{aligned} \tag{۱۱.۵-۴}$$

در اینجا فرض می‌کنیم که مقادیر μ_τ^{loc} و μ_τ^{tx} از قبل معلوم است. در پیوست ۲ نحوه‌ی بدست آوردن مقادیر v_{tx} و v_{loc} در قالب کد شرح داده شده است.

۴-۶ استراتژی تخلیه‌ی وظیفه‌ی بهینه

با توجه به توابع بدست آمده برای تاخیر و توان مصرفی میانگین در بخش‌های پیشین، حال می‌توانیم مسئله‌ی پیدا کردن استراتژی تخلیه‌ی بهینه را به صورت یک مسئله‌ی بهینه سازی مانند $\mathcal{P}_1$ بیان کنیم:

$$\mathcal{P}_1 : \min_{\{g_\tau^a\}} \bar{T} = \left(\sum_{i=1}^{k} \frac{1}{\alpha_i} \sum_{\tau\in S} \tau\{q_i\} \cdot \pi_\tau\right) + T_p^0$$

$$\text{s.t.} \begin{cases} \bar{P} \le \bar{P}_{\max} \\ \sum_{\boldsymbol{\tau}'\in\mathcal{S}} \chi_{\boldsymbol{\tau}',\boldsymbol{\tau}} \pi_{\boldsymbol{\tau}'} = \pi_{\boldsymbol{\tau}}, \boldsymbol{\tau} \in \mathcal{S}, \\ \sum_{\boldsymbol{\tau}\in S} \pi_{\boldsymbol{\tau}} = 1, \\ \sum_{\alpha\in A} g_\tau^\alpha = 1, \forall \tau \in S \\ g_\tau^a \ge 0, \forall \tau \in S,\ a \in A \end{cases} \tag{۱۲.۶-۴}$$

که در آن T_p^0 برابر با تاخیر اجرای میانگین است که به ازای مقادیر داده شده از $\eta_0, \cdots, \eta_k$ مقداری ثابت دارد و از رابطه‌ی زیر بدست می‌آید:

$$T_p^0 = \sum_{i=1}^{k} (\eta_i L_i + (1-\eta_i) t_c^i) \tag{۱۳.۶-۴}$$

مسئله‌ی $\mathcal{P}_1$ به دلیل وجود پارامتر η_i در تابع هدف یک مسئله خطی نیست. با این حال می‌توانیم با استفاده از تغییری کوچک مسئله را به مجموعه‌ای از مسائل برنامه‌ریزی خطی تبدیل کنیم. به این منظور مشابه با [3] ابتدا از تعریف «معیار احاطه[۵]» در زنجیره‌ی مارکوف استفاده می کنیم. به این منظور مجموعه متغیرهای جایگزین $\{x_\tau^a\}$ را طبق رابطه‌ی $x_\tau^a = \pi_\tau g_\tau^a$ تعریف می‌کنیم. به عبارتی x_τ^a برابر با احتمال حضور در حالت τ و انتخاب کنش a می‌باشد. همچنین طبق تعریف می‌دانیم که $\sum_{a\in A} g_\tau^a = 1$ بنابراین خواهیم داشت $\pi_{\boldsymbol{\tau}} = \sum_{a\in A} x_{\boldsymbol{\tau}}^a$

حال با جایگذاری $\{x_\tau^a\}$ به جای $\{\pi_\tau\}$ در $\mathcal{P}_1$ خواهیم داشت:

$$\mathcal{P}_2 : \min_{\boldsymbol{x},\eta} \bar{T} = (\sum_{i=1}^{k} \frac{1}{\alpha_i} \sum_{\tau\in\mathcal{S}} \sum_{a\in A} \tau\{q_i\} \cdot x_\tau^a) + T_p^0$$

$$\text{s.t.} \begin{cases} \nu_{loc}(\boldsymbol{x}) P_{loc} + \beta \nu_{tx}(\boldsymbol{x}) P_{tx} \le \bar{P}_{\max} \\ \Gamma(\boldsymbol{x}, \eta_i) =, \forall i \in \{1, \cdots, k\} \\ F_\tau(\boldsymbol{x}) = 0, \forall \tau = (i, m, n) \in \mathcal{S} \\ \sum_{\tau\in\mathcal{S}} \sum_{a\in A} = 1 \\ \eta_i \in [0,1], \forall i \in \{1, \cdots, k\} \\ x_\tau^a \ge 0, \forall \tau \in S, a \in A \end{cases} \tag{۱۴.۶-۴}$$

که در آن ν_{loc} و ν_{tx} به ترتیب احتمال فعالیت پردازنده و واحد ارسال را در یک واحد زمانی دلخواه مشخص می‌کنند و به ازای یک استراتژی داده شده قابل محاسبه اند. [۶] تابع $\Gamma(\boldsymbol{x}, \eta_i)$ بر اساس

[5] Occupation Measure

[۶] برای مشاهده روش محاسبه این دو پارامتر در قالب کد به پیوست ۲ مراجعه شود.

رابطه‌ی ۴-۸.۴ می‌باشد و به صورت زیر محاسبه می‌شود:

$$\Gamma(x,\eta) = \eta \sum_{\boldsymbol{\tau},\boldsymbol{a}\in\mathcal{S}_1^i\cup\mathcal{S}_2^i\cup\mathcal{S}_3^i\cup\mathcal{S}_4^i} x_\tau^a + 2\eta \sum_{\boldsymbol{\tau},\boldsymbol{a}\in S_5^i} x_\tau^a - \eta \sum_{\boldsymbol{\tau},\boldsymbol{a}\in\mathcal{S}_1^i\cup\mathcal{S}_3^i\cup\mathcal{S}_5^i} x_\tau^a \qquad (۱۵.۶-۴)$$

و تابع $F_\tau(\boldsymbol{x})$ به صورت زیر تعریف می‌شود:

$$F_\tau(\boldsymbol{x}) = \sum_{\tau'\in\mathcal{S}}\sum_{a\in A} \tilde{\chi}_{\tau',\tau,a} x_{\tau'}^a - \sum_{a\in A} x_\tau^a \qquad (۱۶.۶-۴)$$

در رابطه‌ی فوق منظور از $\tilde{\chi}_{\tau',\tau,a}$ احتمال شرطی این است که به شرط اینکه در حالت τ' باشیم و کنش a انتخاب شده باشد، آنگاه به حالت τ' برویم و مطابق با رابطه‌ی ۴-۱۷.۶ بدست می آید. لازم به ذکر است که مقدار $\tilde{\chi}_{\tau',\tau,a}$ بر خلاف $\chi_{\tau,\tau'}$ نسبت به استراتژی تخلیه احتمالی g_τ^a مستقل است.

$$\tilde{\chi}_{\tau,\tau',\alpha} = P\left(\tau[t+1] = \tau' \mid \tau[t] = \tau \ \wedge v[t] = a\right) \qquad (۱۷.۶-۴)$$

در صورتی که مقادیر $\eta_0, \cdots, \eta_k$ معلوم باشد آنگاه مسئله $\mathcal{P}_2$ تبدیل به یک مسئله‌ی برنامه‌ریزی خطی می‌شود. با یافتن مقادیر جواب بهینه $\{x_\tau^a\}$ می‌توان استراتژی بهینه را طبق رابطه‌ی زیر بدست آورد:

$$g_\tau^{a*} = \frac{x_\tau^{a*}}{\sum_{a\in A} x_\tau^{a*}}, \forall\tau\in\mathcal{S}, a\in A \qquad (۱۸.۶-۴)$$

بنابراین جهت یافتن استراتژی بهینه برای یک سیستم تخلیه‌ی وظیفه کافی است که مسئله‌ی برنامه‌ریزی خطی حاصل از $\mathcal{P}_2$ را به ازای مقادیر مختلف $\eta_0, \cdots, \eta_k$ حل کرده تا استراتژی بهینه بدست بیاید. مراحل این فرآیند جستجو در الگوریتم ۱ به صورت خلاصه آمده است. در این الگوریتم تابع $splitRange$ تابعی است که با گرفتن یک بازه از اعداد حقیقی مانند R و پارامتر $precision$، تعداد $precision$ نمونه با فاصله‌های یکسان از بازه R را در قالب یک لیست بر می‌گرداند. منظور از $splitRange([0,1], precision)^k$ نیز ضرب دکارتی k نمونه از این لیست‌های خروجی در یک دیگر می‌باشد.

الگوریتم ۱.۴ الگوریتم جستجوی استراتژی تخلیه‌ی وظیفه‌ی بهینه

Require: $precision \geq 2$
$etaSettings \leftarrow splitRange([0,1], precision)^k$
$optimalPolicy = null$
for each $s \in etaSettings$ **do**
  $(\eta_0, \cdots, \eta_k) \leftarrow s$
  $solution \leftarrow solveLP(\eta_0, \cdots, \eta_k)$
  if $optimalPolicy = null$ **or** $solution.delay < optimalPolicy.delay$ **then**
    $optimalPolicy \leftarrow solution.policy$
  end if
end for
return $optimalPolicy$

۴-۷ دو بهینه‌سازی برای الگوریتم جستجوی استراتژی

در این بخش دو بهینه‌سازی مختلف را به منظور بهبود عملکرد الگوریتم ۱.۴ معرفی می‌کنیم. این دو بهینه‌سازی در چارچوب Kompute که در فصل پیش رو ارائه خواهد شد پیاده‌سازی شده‌اند.

۴-۷-۱ کاهش تعداد متغیرها

در مسئله‌ی بهینه‌سازی $\mathcal{P}_2$ تعداد $|S| \cdot |A|$ متغیر وجود دارد. این مقدار برای تعداد صف‌های کم (برای مثال $k \leq 3$) قابل اجرا می‌باشد اما با افزایش تعداد صف‌ها اجرای الگوریتم را بسیار زمان‌بر و یا غیرممکن می‌کند. یک بهینه‌سازی خیلی ساده ولی کارآمد که در [3] به آن اشاره‌ای نشده است این است که می‌توان تمام متغیرهای مانند x_τ^a که کنش a جزو کنش‌های ممکن در τ نباشد را حذف کرد زیرا مقدار آنها در جواب مسئله‌ی همواره برابر صفر می‌باشد. برای مثال در جدول ۴-۳ مجموعه‌ی تمام کنش‌های سیستم تخلیه توصیف شده در جدول ۴-۱ به همراه امکان‌پذیری هر کنش در صورت حضور در حالت $\tau = ([3,0],0,1,0,1)$ مشخص شده است. همانطور که مشاهده می‌شود در این حالت فقط ۲ کنش از مجموعه‌ی ۹ کنش موجود در A امکان‌پذیر می‌باشند. کنش‌های ردیف ۲ و ۳ به دلیل مشغول بودن پردازنده امکان‌پذیر نمی‌باشند. کنش ردیف ۵ به دلیل خالی بودن صف وظایف نوع ۲ امکان‌پذیر نمی‌باشد. کنش‌های ردیف ۶ الی ۹ نیز به دلیل مشغول بودن پردازنده امکان‌پذیر

Row	Action	Is Possible
1	`NoOperation`	Yes
2	`AddToCPU(queueIndex = 1)`	No
3	`AddToCPU(queueIndex = 2)`	No
4	`AddToTransmissionUnit(queueIndex = 1)`	Yes
5	`AddToTransmissionUnit(queueIndex = 2)`	No
6	`AddToBothUnits(cpuQueueIndex = 1, tuQueueIndex = 1)`	No
7	`AddToBothUnits(cpuQueueIndex = 1, tuQueueIndex = 2)`	No
8	`AddToBothUnits(cpuQueueIndex = 2, tuQueueIndex = 1)`	No
9	`AddToBothUnits(cpuQueueIndex = 2, tuQueueIndex = 2)`	No

جدول ۴-۳: امکان‌پذیری کنش‌های مختلف در حالت $\tau = ([3, 0], 0, 1, 0, 1)$

نمی‌باشند. بنابراین می‌توانیم به سادگی ۷ متغیر متناظر این کنش‌ها را از مجموعه $\{x_\tau^a\}$ حذف کنیم بدون اینکه تغییری در جواب مسئله‌ی بهینه‌سازی $\mathcal{P}_2$ ایجاد شود.

۴-۷-۲ موازی‌سازی

الگوریتم ۱.۴ به گونه‌ای تعریف شده است که امکان موازی‌سازی و مقیاس‌پذیری آن به صورت خطی وجود دارد. به عبارت دیگر می‌توان مسئله‌ی برنامه‌ریزی خطی متناظر با هر مقداردهی از $\eta_0, \cdots, \eta_k$ را به یک هسته یا گره پردازشی خاص اختصاص داد. در شبیه‌سازی سناریوی وظایف سبک و سنگین (رجوع شود به ۶-۲-۳) مشاهده شد که الگوریتم موازی‌سازی شده هنگام اجرا بر روی سروری با ۲۴ هسته و تقسیم‌بندی به ۲۴ ریسمان عملکردی معادل ۲۰ برابر سریع‌تر از حالت تک‌ریسمان[7] دارد.

[7] Single-thread

الگوریتم ۲.۴ الگوریتم موازی‌سازی شده‌ی جستجوی استراتژی تخلیه‌ی بهینه

Require: $precision \geq 2$
Require: $threadCnt \geq 1$
1: **synchronized** $optimalPolicy = null$
2: **procedure** $findOptimalForEtaSettings(etaSettings)$
3: **for each** $s \in etaSettings$ **do**
4: $(\eta_0, \cdots, \eta_k) \leftarrow s$
5: $solution \leftarrow solveLP(\eta_0, \cdots, \eta_k)$
6: **if** $optimalPolicy = null$ **or** $solution.delay < optimalPolicy.delay$ **then**
7: $optimalPolicy \leftarrow solution.policy$
8: **end if**
9: **end for**
10: **end procedure**
11: $etaSettings \leftarrow splitRange([0, 1], precision)^k$
12: $etaBatches \leftarrow splitToBatches(etaSettings, threadCnt)$
13: **for each** $i \in 1 \cdots threadCnt$ **do**
14: $thread[i] = Thread\{findOptimalForEtaSettings(etaBatches[i])\}$
15: **end for**
16: **for each** $i \in 1 \cdots threadCnt$ **do**
17: $thread[i].start()$
18: **end for**
19: **for each** $i \in 1 \cdots threadCnt$ **do**
20: $thread[i].join()$
21: **end for**
22: **return** $optimalPolicy$

فصل پنجم

۵. پیاده‌سازی عملی

در این فصل چارچوب نرم‌افزاری «**کامپیوت**[1]» را ارائه می‌دهیم که با استفاده از آن می‌توان الگوریتم ارائه‌شده در فصل پیشین برای یافتن استراتژی تخلیه‌ی بهینه را به ازای پارامترهای محیطی مختلف اجرا کرد و عملکرد استراتژی تخلیه را با کمک شبیه‌سازی بررسی کرد. مخزن پروژه کامپیوت از لینک زیر در دسترس می‌باشد:

- https://github.com/dalisyron/Kompute

این چارچوب طبق یافته‌های ما اولین پیاده‌سازی متن‌باز در زمینه استراتژی تخلیه‌ی وظیفه‌ی ناهمگون در رایانش لبه‌ای است. در این فصل ابتدا توضیح مختصری در مورد نحوه‌ی کارکرد و معماری کامپیوت خواهیم داد و سپس برنامه‌های نمونه‌ای برای «پیدا کردن استراتژی تخلیه‌ی بهینه» و «شبیه‌سازی استراتژی تخلیه» ارائه خواهیم کرد. کامپیوت در زبان کاتلین[2] نوشته شده است که زبان برنامه‌نویسی چندمنظوره‌ای است که نخستین بار توسط شرکت «جت برینز[3]» ارائه شد. محبوب‌ترین نسخه این زبان نسخه ماشین مجازی جاوا می‌باشد. دلایل اصلی انتخاب کاتلین برای پیاده‌سازی پروژه‌ی فعلی عبارتند از:

- سادگی نحو زبان
- قابلیت‌های زیاد کتابخانه‌ی استاندارد
- پشتیبانی از واسط بومی جاوا[4] به منظور حل سریع برنامه‌های خطی در زبان C++

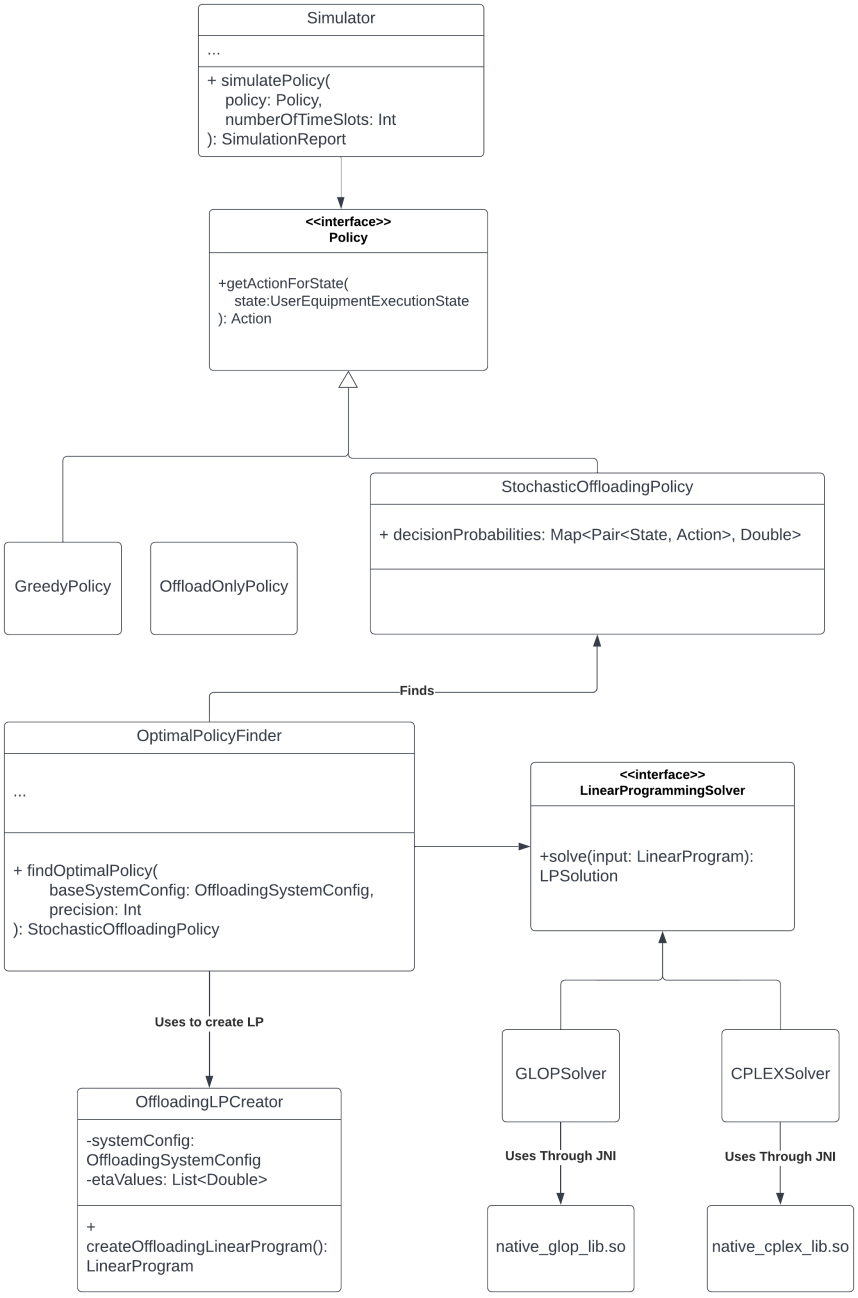
معماری کلی کامپیوت در قالب یک کلاس دیاگرام در شکل ۵-۱ آورده شده است.

[1] Kompute

[2] Kotlin

[3] JetBrains

[4] Java Native Interface



شکل ۵-۱: کلاس دیاگرام چارچوب Kompute

۵-۱ مولفه‌های اصلی چارچوب Kompute

۵-۱-۱ واسط Policy

واسط `Policy` قردادی است که تمام استراتژی‌های تخلیه‌ی وظیفه باید آن را پیاده‌سازی کنند. همانطور که در فصل ۴ گفته شد، یک استراتژی تخلیه‌ی وظیفه، می‌بایست که با توجه به حالت سیستم در زمانی مشخص، تصمیم بگیرد که چه کنشی برای اجرا در آن بازه زمانی انتخاب شود. این منطق در کامپیوت با رابط مشخص‌شده در قطعه کد ۵-۱ تعریف شده است.

قطعه کد ۵-۱: واسط Policy

```
interface Policy {

   fun getActionForState(state: UserEquipmentExecutionState): Action
}
```

به عنوان نمونه برای پیاده‌سازی استراتژی تخلیه‌ی وظیفه‌ی «حریصانه-تخلیه اول[5]» کلاس وارث مطابق با قطعه کد ۵-۲ تعریف می‌شود. این استراتژی تخلیه در صورتی که بتواند هم تخلیه و هم پردازش محلی انجام دهد، هر دو را انجام خواهد داد و در صورتی که تنها یک وظیفه در صف‌های وظایف باشد آن وظیفه را تخلیه خواهد کرد.

قطعه کد ۵-۲: پیاده‌سازی استراتژی تخلیه‌ی وظیفه‌ی حریصانه-تخلیه‌ی اول

```
class GreedyOffloadFirstPolicy : Policy {

   override fun getActionForStateGreedy(state: UserEquipmentExecutionState): Action {
      if (state.averagePower() > state.pMax) return Action.NoOperation
      if (state.ueState.isCPUActive() && state.ueState.isTUActive()) {
         return Action.NoOperation
      }
      if (state.taskQueueLength.all { it == 0 }) return Action.NoOperation
      if (state.ueState.isCPUActive()) {
         return OffloadOnlyPolicy.getActionForState(state)
      }
      if (state.ueState.isTUActive()) {
         return LocalOnlyPolicy.getActionForState(state)
      }
      val nonEmptyIndices = state.taskQueueLength.indices.filter {
         state.taskQueueLength[it] > 0
      }
      require(nonEmptyIndices.isNotEmpty())
      val queueIndices: Pair<Int, Int>? =
      state.ueState.getTwoRandomQueueIndicesForTwoTasks()
```

[5] Greedy-Offload First

```
        if (queueIndices == null) {
            return Action.AddToTransmissionUnit(nonEmptyIndices[0])
        } else {
            return Action.AddToBothUnits(queueIndices.first, queueIndices.second)
        }
    }
}
```

۵-۱-۲ کلاس OffloadingLPCreator

این کلاس وظیفه‌ی ساخت مسئله‌ی برنامه‌ریزی خطی $\mathcal{P}_2$ که در رابطه‌ی ۴-۱۴.۶ بیان شد را دارد. بدین منظور این کلاس پنج شگرد زیر را تعریف می‌کند:

- `fun getObjectiveEquation(): EquationRow`
- `fun getEquation2(): EquationRow`
- `fun getEquations3(): List<EquationRow>`
- `fun getEquation4s(): List<EquationRow>`
- `fun getEquation5(): EquationRow`

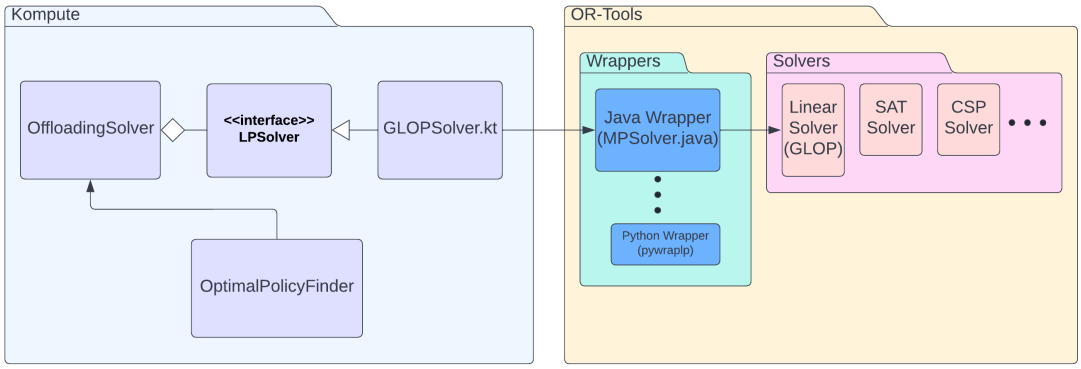
که به ترتیب تابع هدف مسئله‌ی بهینه‌سازی و چهار شرط ذکر شده در ۴-۱۴.۶ را محاسبه می‌کنند. در نهایت با ترکیب این پنج تابع یک شی برنامه خطی توسط شگرد `createOffloadingLinearProgram` ایجاد می‌شود که به منظور محاسبه‌ی استراتژی تخلیه‌ی بهینه نیاز داریم که آن را حل کنیم. در ادامه به نحوه‌ی حل این برنامه خطی توسط کلاس‌های `Solver` می‌پردازیم.

۵-۱-۳ حل‌کننده‌ی خطی

مولفه‌های شرکت‌کننده در حل مسئله‌ی بهینه‌سازی در کامپیوت، در شکل ۵-۲ مشخص شده اند. در پایین‌ترین لایه برای حل مسئله برنامه‌ریزی خطی از حل‌کننده خطی [۶] به نام GLOP استفاده شده است. این حل‌کننده جزئی از پروژه متن‌باز OR-Tools است[۷] که توسط شرکت گوگل ارائه‌شده و

[6] Linear Solver

[7] https://github.com/google/or-tools



شکل ۵-۲: مولفه‌های شرکت‌کننده در حل مسئله‌ی بهینه‌سازی در Kompute

نگه‌داری می‌شود. [18] این حل‌کننده مانند اکثر حل‌کننده‌های سریع و مدرن، در زبان C++ نوشته شده است، اما احاطه‌گرهای آن در زبان‌های دیگر مانند پایتون، جاوا، و سی شارپ وجود دارد. در پروژه‌ی فعلی ، ما از احاطه‌گر زبان جاوا استفاده کرده‌ایم که در پروژه OR-Tools با استفاده از رابط بومی جاوا و در قالب کلاس `MPSolver` پیاده‌سازی شده است.

در Kompute کلاسی به نام GLOPSolver وجود دارد که با گرفتن یک شی از نوع برنامه خطی در دامنه Kompute، آن شی را به برنامه‌خطی قابل شناسایی برای کلاس MPSolver تبدیل می‌کند و در نهایت نتیجه حل برنامه‌خطی را بر می‌گرداند. کلاس MPSolver قابلیت پشتیبانی از برخی از حل‌کننده‌های دیگر به جز GLOP را نیز دارد. با این وجود، به دلیل متن‌باز بودن پروژه، از برخی از حل‌کننده‌های تجاری معروف مانند CPLEX[8] پشتیبانی نمی‌کند. معماری Kompute به گونه‌ای طراحی شده است که امکان استفاده از هر حل‌کننده خطی در آن وجود داشته باشد. برای نمونه علاوه بر حل‌کننده پیش‌فرض GLOPSolver کلاسی با نام CPLEXSolver نیز در پروژه وجود دارد که در صورتی که نسخه تجاری CPLEX که دارای جواز معتبر باشد بر روی سیستم کاربر نصب باشد از آن حل‌کننده استفاده خواهد شد.

[8] https://www.ibm.com/analytics/cplex-optimizer

۵-۲ تعریف و حل یک مسئله‌ی تخلیه‌ی وظیفه‌ی نمونه در Kompute

در قطعه کد نمونه‌ی زیر، مسئله‌ی تخلیه‌ی وظیفه برای محیط رایانش لبه‌ای با دو صف با نوع وظایف «سبک» و «سنگین» حل شده است.

قطعه کد ۵-۳: تعریف و حل مسئله‌ی نمونه

```
fun main(args: Array<String>) {
   val systemConfig = OffloadingSystemConfig(
      userEquipmentConfig = UserEquipmentConfig(
         stateConfig = UserEquipmentStateConfig(
            taskQueueCapacity = 5,
            tuNumberOfPackets = listOf(1, 3),
            cpuNumberOfSections = listOf(7, 2),
            numberOfQueues = 2
         ),
         componentsConfig = UserEquipmentComponentsConfig(
            alpha = listOf,4.0) ,(9.0
            beta = ,90.0
            etaConfig = null,
            pTx = ,0.1
            pLocal = ,8.0
            pMax = 7.1
         )
      ),
      environmentParameters = EnvironmentParameters(
         nCloud = listOf(1, 1),
         tRx = ,5.0
      )
   )
   val optimalPolicy = RangedOptimalPolicyFinder.findOptimalPolicy(
      baseSystemConfig = systemConfig,
      precision = 10
   )
   /*
   // For multi-threaded execution use this instead:
   val optimalPolicy = ConcurrentRangedOptimalPolicyFinder(
      baseSystemConfig = systemConfig
   ).findOptimalPolicy(precision = 10, numberOfThreads = 8)
   */
   val decisionProbabilities: Map<StateAction, Double>
      = optimalPolicy.stochasticPolicyConfig.decisionProbabilities
   println(decisionProbabilities)
}
```

به این منظور ابتدا پارامترهای محیط تخلیه‌ی وظیفه در رایانش لبه‌ای را با استفاده از کلاس `OffloadingSystemConfig` مشخص می‌کنیم. سپس استراتژی بهینه را با استفاده از کلاس `RangedOptimalPolicyFinder` با دقت لازم پیدا می‌کنیم. در نهایت جواب خروجی به صورت توزیع احتمالی بر روی مجموعه‌ی $|S| \times |A|$ بدست می‌آید.

۵-۳ نحوه‌ی شبیه‌سازی استراتژی‌های تخلیه‌ی وظیفه

در قطعه کد نمونه‌ی زیر، استراتژی تخلیه‌ی بهینه و سه استراتژی تخلیه‌ی پایه شبیه‌سازی شده‌اند و نتایج تاخیر و توان مصرفی گزارش شده است.

قطعه کد ۵-۴: شبیه‌سازی استراتژی‌های تخلیه‌ی وظیفه

```
fun main(args: Array<String>) {
   val baseSystemConfig: OffloadingSystemConfig = Mock.doubleQueueConfig()
   val alpha0Start = 01.0
   val alpha0End = 60.0
   val sampleCount = 30
   val simulationCycles = 1_000_000
   for (i in 0 until sampleCount) {
      val alpha0 = (alpha0Start + i * ((alpha0End - alpha0Start) / (sampleCount - 1)))
      val systemConfig = baseSystemConfig.withAlpha(0, alpha0)
      val optimalPolicy = RangedOptimalPolicyFinder.findOptimalPolicy(
         baseSystemConfig = systemConfig,
         precision = 10
      )
      val simulator = Simulator(systemConfig)
      val simulationResults = with(simulator) {
         listOf(
            simulatePolicy(LocalOnlyPolicy, simulationCycles),
            simulatePolicy(OffloadOnlyPolicy, simulationCycles),
            simulatePolicy(GreedyLocalFirstPolicy, simulationCycles),
            simulatePolicy(optimalPolicy, simulationCycles)
         )
      }
      val (localOnlyDelay,
         offloadOnlyDelay,
         greedyLocalFirstDelay,
         optimalDelay) = simulationResults.map { it.averageDelay }
      val (localOnlyAveragePower,
         offloadOnlyAveragePower,
         greedyLocalFirstAveragePower,
         optimalAveragePower) = simulationResults.map { it.averagePowerConsumption }
      println("$localOnlyDelay | " +
         "$offloadOnlyDelay | " +
         "$greedyLocalFirstDelay | " +
         "$optimalDelay")
      println("$localOnlyAveragePower | " +
         "$offloadOnlyAveragePower | " +
         "$greedyLocalFirstAveragePower | " +
         "$optimalAveragePower")
   }
}
```

در این مثال به ازای مقادیر نرخ ورود مختلف برای صف شماره یک، با کمک کلاس `Simulator` معیارهای توان مصرفی و تاخیر محاسبه و گزارش شده است. پارامتر `simulationCycles` تعداد بازه‌های زمانی شبیه‌سازی را مشخص می‌کند. پرواضح است که هر چه این مقدار بالاتر باشد، دقت شبیه‌سازی بالاتر خواهد بود.

فصل ششم

۶. آزمایش و نتیجه

در این فصل قصد داریم عملکرد و درستی روش ارائه‌شده در فصل‌های ۴ و ۵ را با کمک آزمون‌های مبتنی بر شبیه‌سازی بررسی کنیم. در بخش نخست از این فصل، مدل تعریف شده برای مسئله بهینه‌سازی را صحت‌سنجی می‌کنیم. بدین منظور تاخیر سرویس بدست آمده توسط مدل برنامه‌ریزی خطی را با نتیجه اجرای شبیه‌سازی بر روی استراتژی تخلیه‌ی بدست‌آمده از همان مدل مقایسه خواهیم کرد. به عبارتی به کمک شبیه‌سازی مدل را با «خودش» مقایسه خواهیم کرد. در بخش دوم، به مقایسه‌ی عملکرد استراتژی حاصل از الگوریتم با سایر استراتژی‌ها می‌پردازیم. در این بخش چندین استراتژی رایج را به عنوان استراتژی پایه[1] در نظر می‌گیریم و عملکرد استراتژی تخلیه حاصل از الگوریتم را با کمک شبیه‌سازی در برابر آنها می‌سنجیم.

۶-۱ بررسی صحت مدل

در این بخش بررسی می‌شود که آیا مقدار تاخیر سرویس بدست آمده توسط شبیه‌سازی با مقدار تاخیر مشخص شده در جواب بهینه‌ی مسئله‌ی $\mathcal{P}_2$ (رابطه‌ی ۴-۱۴.۶) همخوانی دارد یا نه. بدین منظور نتایج الگوریتم جستجوی استراتژی تخلیه را در دو سناریو شبیه‌سازی مختلف بررسی می‌کنیم.

۶-۱-۱ سناریوی تک صف

در این آزمون محیطی با یک صف وظیفه در نظر گرفته شده است که ویژگی‌های آن در جدول ۶-۳ مشخص شده است. شبیه‌سازی به ازای ۲۹ مقدار مختلف η_0 اجرا شده است که نتایج آن در مقایسه با مقدار محاسبه شده توسط مدل برنامه‌ریزی خطی مسئله، در جدول ۶-۱ خلاصه شده است. تعداد سیکل‌های شبیه‌سازی در هر ۲۹ حالت برابر با 10^7 بوده است. همانطور که مشاهده می‌شود مقدار

[1] Baseline

تاخیر سرویس محاسبه شده توسط مدل برنامه‌ریزی خطی، بسیار نزدیک به مقدار تاخیر بدست آمده توسط شبیه‌سازی می‌باشد.

۶-۱-۲ سناریوی دو صف

در این آزمون محیطی با دو صف وظیفه در نظر گرفته شده است که ویژگی‌های آن در جدول ۶-۴ مشخص شده است. شبیه‌سازی به ازای ۲۲ مقداردهی مختلف به η_1, η_2 اجرا شده است که نتایج آن در مقایسه با مقدار محاسبه شده توسط مدل مسئله در جدول ۶-۲ خلاصه شده است. تعداد سیکل‌های شبیه‌سازی در هر ۲۹ حالت برابر با 10^7 بوده است. همانطور که مشاهده می‌شود مقدار تاخیر سرویس محاسبه شده توسط مدل برنامه‌ریزی خطی، بسیار نزدیک به مقدار بدست آمده توسط شبیه‌سازی می‌باشد.

۶-۲ بررسی عملکرد در مقایسه با استراتژی‌های پایه

در این بخش عملکرد استراتژی یافت شده توسط الگوریتم ۱.۴ را با چهار الگوریتم پایه زیر مقایسه می‌کنیم:

۱. استراتژی «فقط تخلیه»[2] که همه‌ی وظایف را تخلیه می‌کند

۲. استراتژی «حریصانه، تخلیه اول»[3] که در هر بازه زمانی اگر واحد ارسال یا پردازنده بیکار باشند به هر کدام از آنها یک وظیفه از صفی رندوم تخصیص می‌دهد و در صورتی که تنها یک وظیفه در صف باشد و مجبور به انتخاب بین تخلیه و اجرای محلی باشد، تخلیه را انتخاب می‌کند.

۳. استراتژی «حریصانه، محلی اول»[4] که در هر بازه زمانی اگر واحد ارسال یا پردازنده بیکار باشند به هر کدام از آنها یک وظیفه از صفی رندوم تخصیص می‌دهد و در صورتی که تنها یک وظیفه در صف باشد و مجبور به انتخاب بین تخلیه و اجرای محلی باشد، اجرای محلی را انتخاب می‌کند.

[2] Offload Only

[3] Greedy (Offload First)

[4] Greedy (Local First)

η_1	Delay (model estimate)	Delay (simulation result)	Error
0.01	5.9403373	5.945382	0.0050447
0.02	5.9413582	5.9441002	0.002742
0.03	5.9873212	5.9979591	0.0106379
0.04	6.0332846	6.0445199	0.0112353
0.05	6.0792458	6.0772735	-0.0019723
0.06	6.1653313	6.1611952	-0.0041361
0.07	6.2608539	6.2771005	0.0162466
0.08	6.3563761	6.3485279	-0.0078482
0.09	6.4518981	6.4535551	0.001657
0.1	6.5474205	6.5470207	-0.0003998
0.11	6.6429429	6.6441015	0.0011586
0.12	6.7384654	6.7475386	0.0090732
0.13	6.8339881	6.8304343	-0.0035538
0.14	6.963416	6.967968	0.004552
0.15	7.117219	7.1221635	0.0049445
0.16	7.2710211	7.2660879	-0.0049332
0.17	7.4248235	7.4243839	-0.0004396
0.18	7.578626	7.5757626	-0.0028634
0.19	7.7324285	7.7334524	0.0010239
0.2	7.8862311	7.8823844	-0.0038467
0.21	8.0400334	8.0431362	0.0031028
0.22	8.193836	8.1896367	-0.0041993
0.23	8.3476384	8.3507161	0.0030777
0.24	8.5014409	8.50166	0.0002191
0.25	8.6793609	8.6778177	-0.0015432
0.26	8.9366602	8.9342996	-0.0023606
0.27	9.334054	9.3359713	0.0019173
0.28	9.9963099	9.9920628	-0.0042471
0.29	11.6247515	11.6247351	-0.0000164
		Variance	0.0000314
		Mean absolute difference	0.0041032

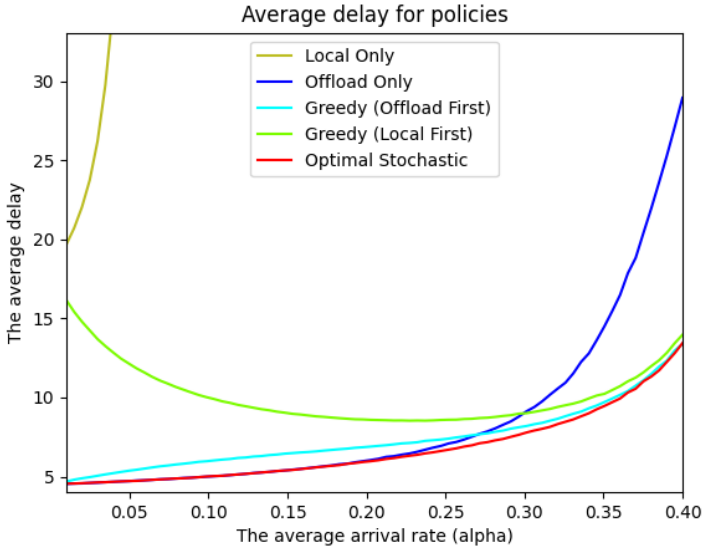
جدول ۶-۱: مقایسه‌ی میزان تاخیر بدست آمده از مدل و شبیه‌سازی در سناریوی تک صف

η_1	η_2	Delay (model estimate)	Delay (simulation result)	Error
0	0	5.3055545	5.3043168	-0.0012377
0	0.2	4.5749717	4.5740879	-0.0008838
0	0.4	4.2116748	4.2124139	0.0007391
0	0.6	3.9532195	3.954735	0.0015155
0	0.8	3.6947642	3.6941737	-0.0005905
0	1	3.4702381	3.4711606	0.0009225
0.2	0	5.4240034	5.425082	0.0010786
0.2	0.2	4.9543158	4.9546973	0.0003815
0.2	0.4	4.7082897	4.7077916	-0.0004981
0.2	0.6	4.5023325	4.5033802	0.0010477
0.2	0.8	4.3225922	4.3218913	-0.0007009
0.2	1	4.3916784	4.3898362	-0.0018422
0.4	0	6.0300612	6.0294958	-0.0005654
0.4	0.2	5.6038029	5.6044667	0.0006638
0.4	0.4	5.3794469	5.3817302	0.0022833
0.4	0.6	5.1951134	5.1951965	0.0000831
0.4	0.8	5.1865812	5.1870742	0.000493
0.4	1	5.4041958	5.4026623	-0.0015335
0.6	0	7.4340974	7.4330417	-0.0010557
0.6	0.2	7.2298104	7.2298944	0.000084
0.6	0.4	7.5736237	7.5743988	0.0007751
0.6	0.6	8.7051662	8.7024916	-0.0026746
			Variance	0.0000314
			Mean absolute difference	0.0041032

جدول ۶-۲: مقایسه‌ی میزان تاخیر بدست آمده از مدل و شبیه‌سازی در سناریوی دو صف

Parameter	M_1	L_1	β	P_{tx}	P_{loc}	P_{max}	C_1	t_{rx}
Value	1	17	0.4	1.0	0.8	1.6	1	0.0

جدول ۶-۳: پارامترهای محیط رایانش لبه‌ای در سناریوی تک صف



شکل ۶-۱: تاخیر سرویس بر حسب نرخ ورود در حالت تک صف
تاخیر سرویس بر حسب نرخ ورود α_1 در حالت تک صف

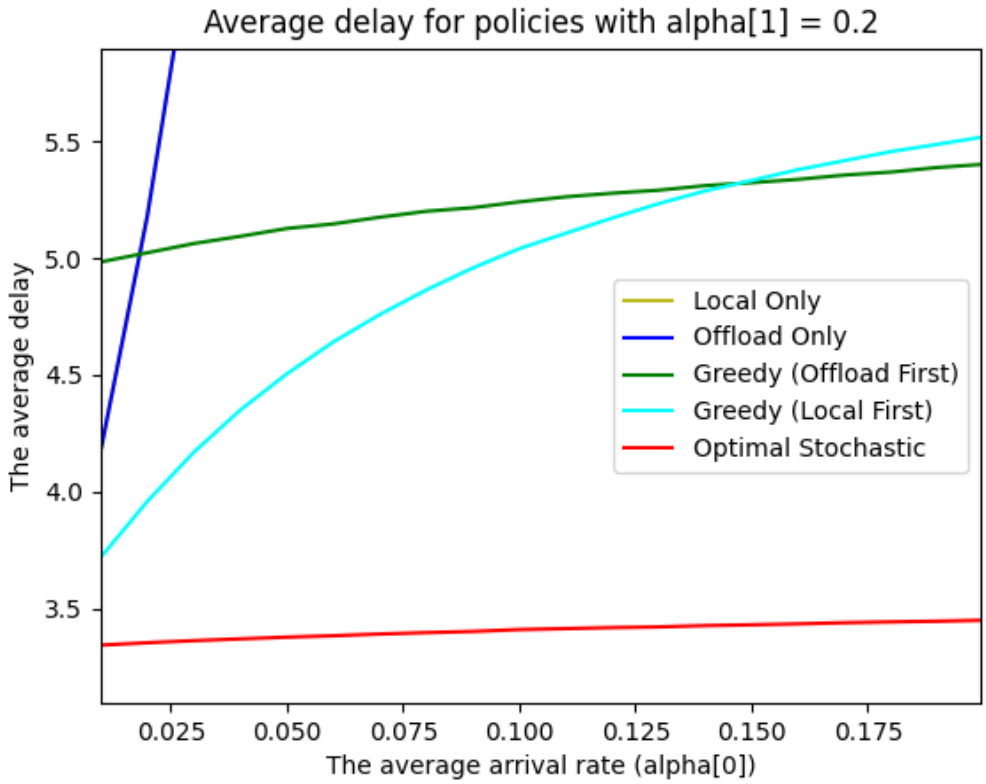
۴. استراتژی «فقط (اجرای) محلی»[5]

۶-۲-۱ شبیه‌سازی تک صف

با توجه به اینکه روش ارائه‌شده توسط ما حالت گسترش یافته [3] است، ابتدا محیط آزمایش ارائه‌شده در آن پژوهش را برای آزمون الگوریتم در نظر می‌گیریم. پارامترهای این محیط در جدول ۶-۳ خلاصه شده اند. نتیجه این آزمایش در شکل ۶-۱ مشاهده می‌شود.

همانطور که مشاهده می‌شود استراتژی تخلیه تصادفی یافت شده از تمام الگوریتم‌های پایه بهتر عمل می‌کند و شکل منحنی‌های نمودار با [3] مطابقت دارد.

[5] Local Only



Parameter	M_1	M_2	L_1	L_2	C_1	C_2	β	P_{tx}	P_{loc}	P_{max}	t_{rx}
Value	1	3	7	2	1	1	0.95	1	0.8	1.6	0

جدول ۶-۴: پارامترهای محیط رایانش لبه‌ای در سناریوی دو صف با یک صف ثابت

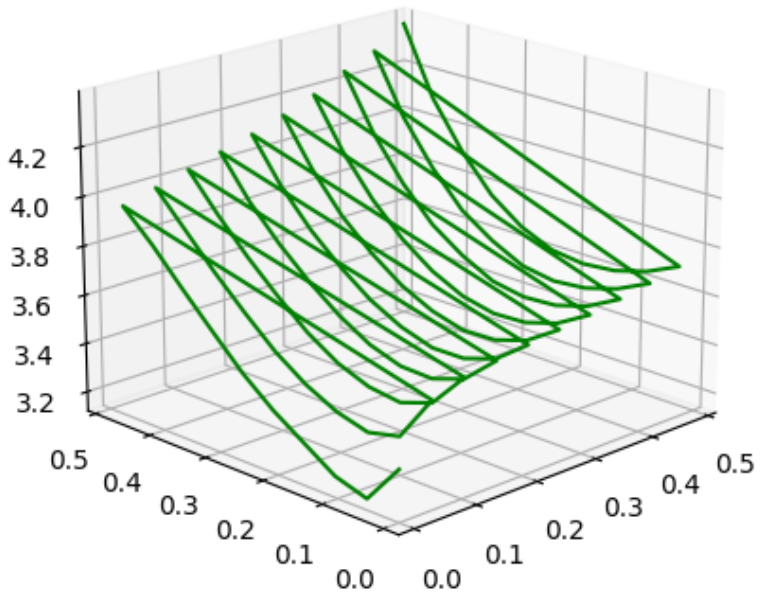
۶-۲-۲ شبیه‌سازی دو صف با یک صف ثابت در سناریوی سبک و سنگین

در این قسمت سناریوی آزمون به این گونه است که میزان تاخیر به ازای مقادیر مختلف نرخ ورود برای صف شماره یک و مقدار ثابت نرخ ورود برای صف شماره دو مشاهده می‌شود. پارامترهای محیطی در نظر گرفته شده در جدول ۶-۴ به طور خلاصه آمده است. همانطور که مشاهده می‌شود استراتژی تخلیه‌ی بهینه بسیار بهتر از الگوریتم‌های پایه عمل می‌کند. دلیل اصلی این تفاوت زیاد (نسبت به تفاوت کم در سناریوی با یک صف در بخش قبل) عدم هوشمندی استراتژی‌های حریصانه در انتخاب نوع وظیفه‌ی تخصیص داده شده به پردازنده و واحد ارسال است. به عبارت دیگر انتخاب تصادفی نوع وظیفه فرستاده شده به پردازنده و واحد ارسال در الگوریتم‌های حریصانه باعث می‌شود که در

شرایطی که تفاوت زیادی بین نوع وظایف وجود دارد (مانند سناریو سبک و سنگین) این الگوریتم‌ها عملکرد خیلی بدی داشته باشند. این در حالی است که در حالت تک صف انتخاب بین انواع وظیفه مطرح نبوده است و تنها عامل برای عملکرد غیربهینه‌ی استراتژی‌های حریصانه، عدم زمانبندی درست وظایف بوده است.

۶-۲-۳ شبیه‌سازی دو صف متغیر وظایف سبک و سنگین

در این قسمت مقدار تاخیر سرویس به ازای مقادیر مختلف نرخ ورود به هر دو صف محاسبه شده است. پارامترهای سیستمی این سناریو در جدول ۶-۵ آمده است. همانطور که مشاهده می‌شود استراتژی بهینه در بازه $\alpha_1, \alpha_2 \in [0, 0.4]$ عملکرد قابل قبول دارد.



شکل ۶-۲: تاخیر سرویس بر حسب نرخ ورود α_1 و α_2 در حالت دو صف

Parameter	M_1	M_2	L_1	L_2	C_1	C_2	β	P_{tx}	P_{loc}	P_{max}	t_{rx}
Value	1	3	7	2	1	1	0.95	1	0.8	1.6	0

جدول ۶-۵: پارامترهای محیط رایانش لبه‌ای در سناریوی دو صف متغیر

۶-۲-۴ شبیه‌سازی سه صف وظیفه

در این قسمت عملکرد الگوریتم ارائه‌شده در شرایطی که سه صف وجود دارد بررسی شده است. پارامترهای محیط رایانش لبه‌ای در جدول ۶-۷ آورده شده است. با توجه به اینکه رسم نمودار در شرایط چهار بعدی امکان‌پذیر نیست از مفهومی به نام آزمون «کارآمدی» استفاده می‌کنیم. مفهوم کارآمدی را اینگونه تعریف می‌کنیم که یک استراتژی کارآمد است اگر احتمال پر بودن یک یا چند صف در سیستم از $\frac{1}{|S|}$ کمتر باشد. در این آزمایش، کارآمدی استراتژی‌های مختلف را به ازای ۱۰۰۰ نمونه مختلف در بازه‌های $\alpha_1, \alpha_2, \alpha_3 \in [0, 0.2]$ آزمایش کردیم که نتایج آن در جدول ۶-۶ مشاهده می‌شود.

Policy	Optimal	Local Only	Greedy (Local First)	Greedy (Offload First)	Offload Only
Effectiveness	100.0%	8.5%	80.3%	79.3%	21.6%

جدول ۶-۶: درصد کارآمدی استراتژی‌ها

Parameter	M_1	M_2	M_3	L_1	L_2	L_3	C_1	C_2	C_3	β	P_{tx}	P_{loc}	P_{max}	t_{rx}
Value	1	3	2	4	2	3	1	1	2	0.95	1	0.8	1.6	0.5

جدول ۶-۷: پارامترهای محیط رایانش لبه‌ای در سناریوی سه صف

۶-۳ آزمون کارآیی

یک نکته که در دو بخش پیشین به آن اشاره‌ای نشد کارایی الگوریتم ارائه‌شده از نظر زمان اجرا و حافظه مصرفی می‌باشد. در آزمایش‌های بخش پیشین تعداد صف‌ها ۳ یا کمتر در نظر گرفته شده بود که اجرای الگوریتم مسئله را به راحتی میسر می‌ساخت. دلیل این امر این است که با افزایش تعداد صف‌ها، فضای حالت مسئله به صورت نمایی بزرگ خواهد شد. در جدول ۶-۸ تعداد حالت‌های زنجیره‌ی مارکوف $|S|$ به همراه زمان اجرا و حافظه مصرفی لازم جهت حل مسئله آورده شده است. برای تفسیر راحت‌تر نتایج آزمایش، تعداد بسته‌ها و قسمت‌های تمام صف‌ها برابر مقدار ثابت $L_i = M_i = 2$ در نظر گرفته شده است. البته در شرایط واقعی قطعا تعداد قسمت‌ها و بسته‌های هر صف متفاوت خواهند بود. زیرا در غیر این صورت صف‌های با ویژگی‌های یکسان را می‌توان به یک صف با نرخ ورود مجموع تبدیل کرد. پردازنده استفاده شده در این آزمایش Intel® Xeon® Processor E3-1220 بوده است. حل‌کننده خطی استفاده شده GLOP بوده است[۶]. طول هر وظیفه برابر با $Q = 6$ در نظر گرفته شده است. همانطور که مشاهده می‌شود زمان اجرای الگوریتم به صورت نمایی افزایش می‌یابد و مسئله فقط برای تعداد صف‌های کمتر از ۵ در زمان قابل قبول حل می‌شود. این تعداد در محیط‌های با تنوع وظایف نسبتا کم مانند سناریوی «سبک» «سنگین» که پیشتر بیان شد، انتزاع قابل قبولی از فضای مسئله ارائه می‌دهد و عملکرد بهتری از حالت تک وظیفه دارد. در فصل پیش رو چندین ایده که می‌تواند در کاهش فضای حالت مسئله و بهبود عملکرد الگوریتم موثر باشد، جهت پژوهش بیشتر ارائه‌شده اند.

Number of queues	State count ($\|S\|$)	Running time
1	14	80ms
2	147	433ms
3	1372	7003 ms
4	100842	24164 seconds (∼7 Hours)

جدول ۶-۸: زمان اجرا و اندازه‌ی فضای حالت به ازای تعداد صف $k = 1, 2, 3, 4$

[۶] در آزمایش‌های انجام شده مشاهده شد که GLOP عملکرد بهتری نسبت به CPLEX دارد و به این دلیل انتخاب گردید

فصل هفتم

۷. جمع‌بندی و پیشنهادها

در این پروژه روشی برای بدست آوردن استراتژی تخلیه‌ی وظیفه‌ی تاخیر-کمینه در شرایط حضور چندین نوع وظیفه معرفی شد. همچنین چارچوب نرم‌افزاری جدیدی معرفی شد که قادر به محاسبه‌ی استراتژی تخلیه‌ی وظیفه‌ی بهینه و شبیه‌سازی آن می‌باشد. روش ارائه‌شده به طور جامع تحت آزمایش قرار گرفت و عملکرد آن با کمک شبیه‌سازی بررسی شد. روش ارائه‌شده این پتانسیل را دارد که نحوه‌ی زمان‌بندی وظایف در محیط‌های با وظایف گوناگون مانند اینترنت اشیاء را به طور قابل توجهی بهبود ببخشد. با این حال چندین محدودیت در پروژه‌ی فعلی وجود دارد، که رفع آنها نیاز به پژوهش بیشتر دارد. در ادامه به برخی از این موارد به طور مختصر اشاره می‌کنیم.

۷-۱ بهبود کارآیی الگوریتم

یک محدودیت اصلی در روش ارائه‌شده، افت کارآیی الگوریتم با افزایش تعداد صف‌ها می‌باشد. در فصل ۶ اشاره شد که این افت به دلیل انفجار فضای حالت مسئله می‌باشد. به عبارت دیگر با افزایش تعداد متغیرهای مسئله‌ی برنامه‌ریزی خطی $\mathcal{P}_2$ (رابطه‌ی ۴-۱۴.۶) زمان اجرای الگوریتم به صورت تصاعدی بالا می‌رود. در بخش ۴-۷-۱ روشی ارائه شد که تا حدی اندازه‌ی فضای حالت مسئله را کاهش می‌داد. با این حال همانطور که در نتایج شبیه‌سازی مشاهده شد، الگوریتم ارائه‌شده همچنان برای طول صف‌های بیشتر از ۵ کارآیی ندارد. در بخش فعلی دو ایده مختلف را ارائه می‌کنیم که با پژوهش بیشتر درباره آنها شاید بتوان عملکرد الگوریتم را بهبود ببخشید.

۷-۱-۱ حذف تک‌کنش‌ها

یک ایده برای التیام مشکل انفجار فضای حالت، حذف «تک کنش» ها از فضای مسئله ($|S| \times |A|$) می‌باشد. به طور دقیق‌تر کنش a را برای حالت τ یک **تک‌کنش** می‌نامیم اگر تنها کنش ممکن در

حالت τ باشد. با توجه به اینکه کنش `NoOperation` در همه حالت‌ها ممکن است، تک کنش‌ها همواره متناظر با `NoOperation` می‌باشند.

پیشتر در بخش ۴-۷-۱ به این موضوع اشاره شد که می‌توان متغیرهایی که متناظر با کنش‌های غیر ممکن هستند را از مسئله‌ی بهینه‌سازی $\mathcal{P}_2$ حذف کرد. با استدلالی مشابه این احتمال وجود دارد که بتوان متغیرهایی که متناظر با تنها کنش ممکن در یک حالت هستند را از الگوریتم برنامه‌ریزی خطی حذف کرد، زیرا احتمال انتخاب کنش‌های متناظر با چنین متغیرهایی همواره ۱ (قطعی) می‌باشد و مقدار آن متغیرها در تعیین استراتژی بهینه تاثیری ندارد.

با دقت در فضای حالت مسئله می‌توان دریافت که بسیاری از حالت‌های فضای مسئله دارای تک‌کنش می‌باشند. برای مثال هر حالتی که پردازنده و واحد ارسال هر دو در آن مشغول باشند از این نوع خواهد بود. فرآوانی چنین حالت‌هایی در فضای حالت مسئله بدین معنی است که حذف آنها می‌تواند کارآیی الگوریتم ارائه‌شده را به طور قابل توجهی بهبود بخشد. با این حال حذف تک‌کنش‌ها از لیست متغیرهای مسئله، بر خلاف بهینه‌سازی ۴-۷-۱ ساختار زنجیره‌ی مارکوف را دگرگون خواهد، بنابراین احتمالا نیازمند تغییر توابع انتقال و تغییر شروط رابطه‌ی ۴-۱۴.۶ خواهد بود. پژوهش بیشتر در این زمینه مشکل انفجار فضای حالت را به طور کامل حل نخواهد کرد، اما امید می‌رود که تعداد صف‌های قابل پشتیبانی در روش ارائه‌شده را به طور قابل توجهی افزایش دهد.

۷-۱-۲ اعمال جریمه برای اتلاف وظیفه

یک راه بنیادی‌تر برای رفع مشکل انفجار فضای حالت این است که کلا مدل حالت (τ) مسئله را اینگونه تغییر دهیم که هر صف ظرفیتی برابر با ۲ داشته باشد. طبیعتا اعمال چنین محدودیتی در مسئله فعلی ممکن نیست زیرا فرض کرده‌ایم که اتلاف صورت نمی‌گیرد، به این صورت که حضور در حالت‌های با صف پر ($q_i = Q$) را به منزله غیر کارامد بودن استراتژی در نظر گرفتیم (رجوع شود به بخش ۴-۲-۶). اما می‌توان تغییری در مدل اعمال کرد که هزینه اتلاف وظیفه را نیز در نظر بگیرد. به طور دقیق‌تر مدل برنامه‌ریزی خطی می‌تواند به ازای یک استراتژی تخلیه داده شده و نرخ ورود $\alpha_1 \cdots \alpha_k$ احتمال وقوع اتلاف وظیفه در صف با طول ۲ را در نظر بگیرد. در چنین حالتی می‌توان به

میزان مشخصی مانند Ω جریمه تاخیر در تابع هدف در نظر گرفت. بدین صورت حل کننده خطی در راستای کم کردن تاخیر، سعی خواهد کرد که احتمال وقوع اتلاف وظیفه را در سامانه پایین بیاورد. با استفاده از این تغییر فضای حالت مسئله به طور قابل توجهی کوچک خواهد شد به طوری که اگر در روش قدیمی $|S_1|$ حالت وجود داشته باشد، در روش جدید $\frac{|S_1|\cdot 2^k}{Q^k}$ حالت وجود خواهد داشت.

۷-۲ موضع‌گیری استراتژی تخلیه‌ی وظیفه

یک موضوع مهم که در پروژه‌ی فعلی به آن پرداخته نشد، نحوه‌ی موضع‌گیری (Deployment) استراتژی تخلیه‌ی وظیفه یا به عبارتی «سازوکار تنظیم استراتژی در دستگاه کاربر» است. همانطور که در شبیه‌سازی های انجام شده در فصل ۶ مشاهده شد، الگوریتم محاسبه‌ی استراتژی تخلیه‌ی بهینه به منابع محاسباتی زیادی نیاز دارد. طبیعتا انجام چنین محاسباتی بر روی دستگاه کاربر که موجودیتی مانند تلفن همراه یا اینترنت اشیاء است عملی نخواهد بود. چه بسا که در پروژه‌ی فعلی نیز برای تمام شبیه‌سازی‌ها از سروری قدرتمند با ۲۴ هسته پردازشی استفاده شد. در سطح بالا، یک راه حل برای موضع‌گیری روش پیشنهادی در پروژه، استفاده از معماری مشابه با شبکه‌های مبتنی بر نرم افزار[1] می‌باشد. برای مثال می‌توان در هر سرور رایانش لبه‌ای فرآیندی را اجرا کرد، که هر n ساعت یک بار، اطلاعات محیطی (مانند نرخ ورود هر نوع از وظایف) را از دستگاه‌های کاربر سرویس‌گیرنده بگیرد و متناسب با شرایط هر محیط، استراتژی تخلیه‌ی بهینه را محاسبه کرده و برای دستگاه کاربر مربوطه ارسال کند. با این وجود، پیاده‌سازی کارآمد چنین سازوکاری، نیازمند پژوهش بیشتر می‌باشد.

[1] Software Defined Networks

واژه‌نامه فارسی به انگلیسی

احاطه‌گر . . . . . Wrapper

بی‌درنگ . . . . . Real-time

پردازش . . . . . Computation

تحرک‌پذیری . . . . . Mobility

تصادفی . . . . . Stochastic

چارچوب . . . . . Framework

دانه‌بندی . . . . . Granularity

ریسمان . . . . . Thread

سربار . . . . . Overhead

شگرد . . . . . Method

قطعی . . . . . Deterministic

کاربرد . . . . . Application

محلی . . . . . Local

مقیاس‌پذیری . . . . . Scaling

موازی‌سازی . . . . . Parallelism

موضع‌گیری . . . . . Deployment

هم‌کنش‌پذیری . . . . . Interoperability

پیوست ۱ - توابع انتقال حالت

تابع انتقال حالت به ازای کنش ورودی

```
fun getNextStateRunningAction(
   sourceState: UserEquipmentState,
   action: Action
): UserEquipmentState {
   return when (action) {
      is Action.NoOperation → {
         sourceState
      }
      is Action.AddToCPU → {
         getNextStateAddingToCPU(sourceState, action.queueIndex)
      }
      is Action.AddToTransmissionUnit → {
         getNextStateAddingToTU(sourceState, action.queueIndex)
      }
      is Action.AddToBothUnits → {
         getNextStateAddingToBothUnits(
            sourceState,
            action.cpuTaskQueueIndex,
            action.transmissionUnitTaskQueueIndex
         )
      }
   }
}
```

تابع انتقال حالت پایه

```
fun getNextStateAddingToCPU(
   sourceState: UserEquipmentState,
   queueIndex: Int
): UserEquipmentState {
   require(sourceState.cpuState == 0)
   require(sourceState.taskQueueLengths[queueIndex] > 0)

   val updatedLengths = sourceState.taskQueueLengths.decrementedAt(queueIndex)

   return sourceState.copy(
      taskQueueLengths = updatedLengths,
      cpuState = -1,
      cpuTaskTypeQueueIndex = queueIndex
   )
}
```

تابع انتقال حالت با کنش ارسال توسط واحد ارسال

```
fun getNextStateAddingToTU(
   sourceState: UserEquipmentState,
   queueIndex: Int
): UserEquipmentState {
   require(sourceState.tuState == 0)
   require(sourceState.taskQueueLengths[queueIndex] > 0)

   val updatedLengths = sourceState.taskQueueLengths.decrementedAt(queueIndex)

   return sourceState.copy(
      taskQueueLengths = updateLengths,
      tuState = 1,
      tuTaskTypeQueueIndex = queueIndex
```

```
    )
}
```

تابع انتقال حالت با کنش اجرا و ارسال به طور همزمان

```
fun getNextStateAddingToBothUnits(
    sourceState: UserEquipmentState,
    cpuQueueIndex: Int,
    tuTaskQueueIndex: Int
): UserEquipmentState {
    if (cpuQueueIndex == tuTaskQueueIndex) {
        require(sourceState.taskQueueLengths[cpuQueueIndex] > 1)
    } else {
        require(sourceState.taskQueueLengths[cpuQueueIndex] > 0)
        require(sourceState.taskQueueLengths[tuTaskQueueIndex] > 0)
    }
    return getNextStateAddingToCPU(
        getNextStateAddingToTU(sourceState, tuTaskQueueIndex),
        cpuQueueIndex
    )
}
```

پیوست ۲ - تابع ساخت شرط حداکثر توان مصرفی در برنامه‌ی خطی

تابع ساخت شرط حداکثر توان مصرفی

```
fun getEquation2(): EquationRow {
   val pLoc = systemConfig.pLoc
   val pTx = systemConfig.pTx
   val beta = systemConfig.beta
   val rhsEquation2 = systemConfig.pMax
   val coefficients = mutableListOfZeros(indexMapping.variableCount)

   indexMapping.coefficientIndexByStateAction.forEach { (stateAction, index) →
      val (state, action) = stateAction
      var coefficientValue = 0.0

      if (state.isTUActive()
      || (action is Action.AddToTransmissionUnit
      || action is Action.AddToBothUnits)) {
         coefficientValue += beta * pTx
      }

      if (state.isCPUActive()
      || (action is Action.AddToCPU)
      || (action is Action.AddToBothUnits)) {
         coefficientValue += pLoc
      }

      coefficients[index] = coefficientValue

   }

   return EquationRow(
      coefficients = coefficients,
      rhs = rhsEquation2,
      type = EquationRow.Type.LessThan
   )
}
```

پیوست ۳ - نحوه‌ی محاسبه‌ی ماتریس انتقال

در این بخش نحوه‌ی محاسبه‌ی درایه‌های ماتریس انتقال $\chi_{\tau,\tau'}$ به ازای حالت ورودی τ در قالب کد شرح داده شده است. همانطور که در بخش ۴-۲ گفته شد، درایه‌های ماتریس انتقال معادل «یال» های گراف زنجیره می‌باشند. بنابراین هدف ما پیدا کردن یال‌های گراف با مبدا τ به همراه وزن آنها می‌باشد. به این منظور ابتدا با کمک تابع زیر کنش‌های ممکن را برای حالت ورودی پیدا می‌کنیم:

تابع محاسبه‌ی کنش‌های ممکن به ازای حالت داده شده

```
override fun getPossibleActions(state: UserEquipmentState): List<Action> {
   val result = mutableListOf<Action>(Action.NoOperation)
   if (state.isCPUActive() && state.isTUActive()) return result
   val nonEmptyQueueIndices = state.taskQueueLengths.indices.filter {
      state.taskQueueLengths[it] > 0
   }
   if (!state.isCPUActive())
      for (queueIndex in nonEmptyQueueIndices) {
         if (config.limitation[queueIndex] ≠ StateManagerConfig.Limitation.OffloadOnly) {
            result.add(Action.AddToCPU(queueIndex))
         }
      }
   if (!state.isTUActive()) {
      for (queueIndex in nonEmptyQueueIndices) {
         if (config.limitation[queueIndex] ≠ StateManagerConfig.Limitation.LocalOnly) {
            result.add(Action.AddToTransmissionUnit(queueIndex))
         }
      }
   }
   if (!state.isTUActive() && !state.isCPUActive()) {
      for (i in nonEmptyQueueIndices) {
         for (j in nonEmptyQueueIndices) {
            if (i == j && state.taskQueueLengths[i] < 2) continue
            if (config.limitation[i] ≠ StateManagerConfig.Limitation.OffloadOnly
            && config.limitation[j] ≠ StateManagerConfig.Limitation.LocalOnly) {
               result.add(Action.AddToBothUnits(i,j))
            }
         }
      }
   }
   return result.sorted()
}
```

در مرحله بعد می‌بایست به ازای هر جفت حالت و کنش (τ, a)، مجموعه‌ی حالات ممکن در صورت حضور در حالت τ و انتخاب کنش a را محاسبه کنیم. به این منظور از تابع زیر استفاده می‌کنیم:

تابع محاسبه‌ی کنش‌های ممکن به ازای حالت داده شده

```
fun getTransitionsForAction(state: UserEquipmentState, action: Action): List<Transition> {
   checkStateAgainstLimitations(state)
   val stateAfterAction = getNextStateRunningAction(state, action).let {
      if (it.isCPUActive()) getNextStateAdvancingCPU(it) else it
   }
   checkStateAgainstLimitations(stateAfterAction)
   val transitions: MutableList<Transition> = mutableListOf()
   val notFullIndicesAfterAction = (stateAfterAction.taskQueueLengths.indices).filter {
      val queueLengths = stateAfterAction.taskQueueLengths[it]
      queueLengths < config.userEquipmentStateConfig.taskQueueCapacity
   }
   if (notFullIndicesAfterAction.isEmpty()) {
      if (stateAfterAction.isTUActive()) {
         transitions.add(Transition(
            source = state,
            dest = getNextStateAdvancingTU(stateAfterAction),
            transitionSymbols = listOf(listOf(action, ParameterSymbol.Beta))))
         transitions.add(Transition(state, stateAfterAction,
            listOf(listOf(action, ParameterSymbol.BetaC))))
      } else {
         transitions.add(Transition(state, stateAfterAction, listOf(listOf(action))))
      }
   } else {
      val taskArrivalMappings = getAllSubsets(notFullIndicesAfterAction.size)
      for (mapping in taskArrivalMappings) {
         val addTaskSymbols = mapping.mapIndexed { index, taskArrives →
            if (taskArrives)
               ParameterSymbol.Alpha(notFullIndicesAfterAction[index])
            else
               ParameterSymbol.AlphaC(notFullIndicesAfterAction[index])
         }
         val destState = getNextStateAddingTasksBasedOnMapping(
            stateAfterAction, mapping, notFullIndicesAfterAction
         )
         if (stateAfterAction.isTUActive()) {
            transitions.add(Transition(
               source = state,
               dest = getNextStateAdvancingTU(destState),
               transitionSymbols = listOf(
                  listOf(action, ParameterSymbol.Beta) + addTaskSymbols))
               )
            transitions.add(Transition(
               source = state,
               dest = destState,
               transitionSymbols = listOf(
                  listOf(action, ParameterSymbol.BetaC) + addTaskSymbols))
               )
         } else {
            transitions.add(Transition(state,destState, listOf(
               listOf(action) + addTaskSymbols)))
         }
      }
   }
   return transitions
}
```

در نهایت با ترکیب دو تابعی که تعریف شد می‌توانیم تابع سومی بنویسیم که تمام یال‌های با مبدا τ را پیدا کند:

تابع محاسبه‌ی یال‌های زنجیره به ازای حالت مبدا ورودی

```
fun getEdgesForState(state: UserEquipmentState): List<Edge> {
   return getPossibleActions(state)
   .map { action →
      getTransitionsForAction(state, action)
   }
   .flatten().map { it.toEdge() }
}
```

مراجع

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[2] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2018.

[3] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *2016 IEEE International Symposium on Information Theory (ISIT)*, 2016, pp. 1451–1455.

[4] A. Yousefpour, G. Ishigaki, R. Gour, and J. P. Jue, "On reducing iot service delay via fog offloading," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 998–1010, 2018.

[5] H. Tran-Dang and D.-S. Kim, "Frato: Fog resource based adaptive task offloading for delay-minimizing iot service provisioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2491–2508, 2021.

[6] J. Wang, J. Pan, F. Esposito, P. Calyam, Z. Yang, and P. Mohapatra, "Edge cloud offloading algorithms: Issues, methods, and perspectives," *ACM Computing Surveys*, vol. 52, no. 1, Feb 2019. [Online]. Available: https://doi.org/10.1145/3284387

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: *Association for Computing Machinery*, 2010, pp. 49–62. [Online]. Available: https://doi.org/10.1145/1814433.1814441

[8] G. Hu, Y. Jia, and Z. Chen, "Multi-user computation offloading with d2d for mobile edge computing," in *2018 IEEE Global Communications Conference (GLOBECOM)*, 2018, pp. 1–6.

[9] J. Kwak, Y. Kim, J. Lee, and S. Chong, "Dream: Dynamic resource and task allocation for energy minimization in mobile cloud systems," *IEEE Journal on Selected Areas in Communications*, vol. 33, no. 12, pp. 2510–2523, 2015.

[10] G. Feng, X. Li, Z. Gao, C. Wang, H. Lv, and Q. Zhao, "Multi-path and multi-hop task offloading in mobile ad hoc networks," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 6, pp. 5347–5361, 2021.

[11] X. Meng, W. Wang, and Z. Zhang, "Delay-constrained hybrid computation offloading with cloud and fog computing," *IEEE Access*, vol. 5, pp. 21 355–21 367, 2017.

[12] Y. He, N. Zhao, and H. Yin, "Integrated networking, caching, and computing for connected vehicles: A deep reinforcement learning approach," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 1, pp. 44–55, 2018.

[13] A. Shakarami, M. Ghobaei-Arani, M. Masdari, and M. Hosseinzadeh, "A survey on the computation offloading approaches in mobile edge/cloud computing environment: A stochastic-based perspective," *Journal of Grid Computing*, vol. 18, no. 4, pp. 639–671, Dec 2020. [Online]. Available: https://doi.org/10.1007/s10723-020-09530-2

[14] A. Samanta and Z. Chang, "Adaptive service offloading for revenue maximization in mobile edge computing with delay-constraint," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 3864–3872, 2019.

[15] Z. Jiang and S. Mao, "Energy delay tradeoff in cloud offloading for multi-core mobile devices," *IEEE Access*, vol. 3, pp. 2306–2316, 2015. [Online]. Available: https://doi.org/10.1109/ACCESS.2015.2499300

[16] W. Zhang, Y. Wen, K. Guan, D. Kilper, H. Luo, and D. O. Wu, "Energy-optimal mobile cloud computing under stochastic wireless channel," *IEEE Transactions on Wireless Communications*, vol. 12, no. 9, pp. 4569–4581, 2013.

[17] A.-E. M. Taha, N. A. Ali, and H. S. Hassanein, *Frame Structure, Addressing and Identification*, ser. LTE, LTE-Advanced and WiMAX: Towards IMT-Advanced Networks. *Wiley*, 2011, pp. 59–73. [Online]. Available: http://ieeexplore.ieee.org/document/8045017

[18] Wikipedia contributors, "Glop — Wikipedia, the free encyclopedia," 2022, [Online; accessed 2-July-2022]. [Online]. Available: https://en.wikipedia.org/wiki/GLOP

Abstract:

Edge computing is a distributed computing paradigm that seeks to provide users with lower response times, lower power consumption, and mobility management by bringing computing resources closer to the network edge. Since its introduction, edge computing and its standard implementations, such as Multi-access Edge Computing, have faced one important challenge: How to design efficient task offloading policies? Furthermore, with the rapid growth of the smartphone and IoT industry, many new types of applications have been introduced to the internet, each having different resource needs. Thus, taking into account the heterogeneity of user tasks becomes an essential factor when designing task offloading policies for edge computing environments. This paper introduces a method for finding the delay-optimal task offloading policy under the power consumption constraint. The method consists of two steps. First, the offloading system is modeled using Discrete-time Markov Chains. Then, an algorithm based on linear programming is used to find the optimal task offloading policy for the created model. In addition to discussing the problem mathematically, we introduce a new software framework, written in the Kotlin language, which allows users to find the optimal task offloading policy for a given system. This framework can also benchmark the optimal policy's effectiveness using simulation.

Keywords: Task Offloading, Edge Computing, Markov Chains, Linear Programming, Cloud Computing

Iran University of Science and Technology

Computer Engineering Department

A multi-user computation offloading policy to minimize average latency for IoT devices

Bachelor of Computer Engineering Final Project

By:

Mohammadmobin Dariushhamedani

Supervisor:

Dr. Reza Entezari-Maleki

July 2022