به نام خدا

چکیده

مهاجرت زنده ماشین مجازی (VM) یک ویژگی ضروری برای فروشندگان ابری برای مدیریت زیرساخت های ابری خود است. از آنجایی که VM live migration مقدار زیادی از حافظه نوشتاری را برای همگام سازی حافظه بین ماشین های مجازی منتقل می کند، کاهش میزان انتقال حافظه تاثیر بسزایی در کارایی و موفقیت مهاجرت دارد. روش‌های مبتنی بر نرم‌افزار مختلفی برای کاهش انتقال حافظه پیشنهاد شده‌اند، اما آنها منابع CPU و حافظه قابل توجهی را مصرف می‌کنند و عملکرد را تحت بار سنگین کاهش می‌دهند. **ما پیشنهاد می‌کنیم که از یک محافظت از نوشتن صفحه فرعی مبتنی بر سخت‌افزار برای کاهش انتقال‌های غیرضروری حافظه با انجام تشخیص نوشتن ریز دانه** در مهاجرت VM زنده استفاده کنیم. با این حال، از آنجایی که حفاظت از نوشتن با تشخیص نوشتن متفاوت است، استفاده ساده از آن ممکن است هزینه زیادی را به همراه داشته باشد که بیشتر از مزایای آن باشد. در این مطالعه، ما علت اصلی سربار بزرگ را در یک پیاده سازی ساده لوح شناسایی کردیم و یک بهینه سازی برای جلوگیری از آن معرفی کردیم. آزمایش‌های عملکردی با انتقال شبه مبتنی بر شبیه‌ساز نشان داد که برای بسیاری از بارهای کاری، حفاظت از نوشتن صفحه فرعی می‌تواند زمان انتقال را به اندازه فشرده‌سازی دیفرانسیل ساخته شده در QEMU با مصرف CPU و حافظه کمتر کاهش دهد، و برای برخی از بارهای کاری، فقط موارد فرعی حفاظت از نوشتن صفحه می تواند مهاجرت را کامل کند. ما همچنین

نشان می دهیم که حداکثر سربار CPU را می توان با بهینه سازی ما ۲۵.۵ درصد کاهش داد. شرایط شاخص—مجازی سازی. VM live migration; حفاظت از نوشتن زیر صفحه؛ اینتل SPP

I. مقدمه

انتقال زنده ماشین مجازی (VM) تکنیکی برای انتقال ماشین مجازی در حال اجرا بر روی یک میزبان به میزبان دیگر بدون توقف VM است. انتقال زنده VM **برای متعادل کردن بار، نگهداری سخت افزار، مدیریت توان و تحمل خطا** [۱] مفید است و بنابراین به طور گسترده در مراکز داده ابری استفاده می شود. در انتقال زنده VM، روش پیش کپی [۲] معمولاً برای همگام سازی حافظه بین ماشین های مجازی مبدأ و مقصد استفاده می شود. در این روش، محتویات حافظه VM مبدا در حالی که VM منبع در حال اجرا است به VM مقصد منتقل می شود و سپس محتویات حافظه جدید نوشته شده در حین انتقال مجدداً منتقل می شود. این فرآیند تا زمانی تکرار می شود که مقدار انتقال حافظه در یک تکرار به اندازه کافی کم شود. بنابراین، **کاهش میزان انتقال حافظه برای کاهش زمان توقف، زمان مهاجرت، بار شبکه و مصرف انرژی بسیار مهم است** [۳]. علاوه بر این، **نرخ کاهش در مقدار انتقال حافظه در هر تکرار**، موفقیت یا عدم موفقیت مهاجرت را تعیین می‌کند. به منظور شناسایی ناحیه حافظه جدید نوشته شده در طول انتقال حافظه در هر تکرار، معمولاً از **صفحه‌بندی تودرتو** استفاده می‌شود. در

صفحه‌بندی تودرتو، زمانی که سیستم‌عامل geust در VM روی صفحه فیزیکی مهمان می‌نویسد، CPU بیت کثیف ورودی مربوطه را در جدول صفحه تودرتو تنظیم می‌کند. بنابراین، مانیتور ماشین مجازی (VMM) می تواند صفحات فیزیکی مهمان تازه نوشته شده را با اسکن بیت های کثیف در جدول صفحه تو در تو شناسایی کند. تشخیص نوشتن در صفحه‌بندی تودرتو در سخت‌افزار انجام می‌شود و در نتیجه زمان اجرای نرم‌افزار بسیار کمی دارد. با این حال، از آنجایی که تشخیص نوشتن بر اساس صفحه به صفحه انجام می‌شود، مناطق حافظه‌ای که واقعاً نوشته نشده‌اند ممکن است منتقل شوند. بسیاری از رویکردهای مبتنی بر نرم‌افزار برای تشخیص نوشتن با جزئیات دقیق‌تر از یک صفحه پیشنهاد شده‌اند [۴]-[۹]. یک رویکرد **محاسبه تفاوت از صفحاتی است** که قبلاً در تکرارهای قبلی منتقل شده اند [۴]، [۵]. **روش دیگر محاسبه مقدار هش حافظه** در واحدی کوچکتر از صفحه [۶]، [۷] است. **فشرده‌سازی صفحه [۸]، [۹] نیز تأثیری شبیه به تشخیص نوشتن ریزدانه دارد.** با این حال، این رویکردها به مقدار معینی از محاسبات بر اساس محتویات صفحه نیاز دارند، که می‌تواند تأثیر قابل‌توجهی بر عملکرد VM به دلیل سربار CPU و حافظه داشته باشد، به خصوص زمانی که میزبان به شدت بارگذاری شده است. چندین روش برای کاهش تعداد صفحات برای انتقال نیز پیشنهاد شده است [۱۰]-[۱۸]، اما آنها این را در نظر نمی گیرند.

اثر تشخیص نوشتن در دانه بندی دقیق تر از صفحه. در این مقاله، ما پیشنهاد می کنیم از یک ویژگی **حفاظت از نوشتن زیرصفحه مبتنی بر سخت افزار** برای تشخیص نوشتن ریز دانه در مهاجرت زنده VM استفاده کنیم. یک صفحه فرعی یکی از چندین پارتیشن با طول ثابت یک صفحه است. **به عنوان مثال، یک**

صفحه ۴ کیلوبایتی را می توان به ۳۲ قطعه از صفحات فرعی ۱۲۸ بایتی تقسیم کرد. محافظت از نوشتن صفحه فرعی یک ویژگی است که به صفحه بندی تو در تو اضافه شده است که امکان تنظیم حفاظت از نوشتن را برای هر صفحه فرعی فراهم می کند.

مزایای تشخیص نوشتن صفحه فرعی بر اساس حفاظت از نوشتن صفحه فرعی این است که سربار CPU کمتر به دلیل عدم محاسبه محتویات حافظه و سربار حافظه کمتر است زیرا فقط کمی نشان می دهد که محافظت از نوشتن فعال است باید در حافظه نگهداری شود. هر صفحه فرعی مشخص شده با این حال، حفاظت از نوشتن صفحه فرعی یک تابع بیت کثیف نیست که نوشته‌های موجود در جدول صفحه را در پس‌زمینه ضبط می‌کند، بلکه تابعی است که هر زمان که در یک صفحه فرعی محافظت شده از نوشتن نوشته می‌شود، یک رویداد خطای صفحه ایجاد می‌کند. بنابراین، زمانی که فرکانس نوشتن صفحه فرعی بسیار زیاد است، ممکن است دچار خطای بزرگ صفحه شود.

به منظور روشن شدن این سود و زیان، ما **رفتار نوشته‌های زیر صفحه** را در حین مهاجرت زنده VM برای بارهای کاری مختلف **مشاهده کردیم** و **کاهش انتقال حافظه** و سربار افزایش خطاهای صفحه را تحلیل کردیم. معماری CPU هدف برای اندازه‌گیری، Intel 64 با ویژگی مجوزهای نوشتن زیر صفحه (SPP) [۱۹] بود که در آخرین محصولاتی که اینتل ارسال کرده است گنجانده شده است.

در زمان نگارش، ما نتوانستیم یک CPU واقعی مجهز به SPP بدست آوریم. بنابراین، ما از Bochs [۲۰]، یک شبیه‌ساز IA32 که می‌تواند SPP را شبیه‌سازی کند، با برخی رفع اشکال توسط ما، به عنوان محیط آزمایشی خود استفاده کردیم.

ما همچنین تشخیص نوشتن را با استفاده از SPP در QEMU/KVM پیاده سازی کردیم. متأسفانه، شبیه ساز به طور قابل توجهی کندتر از CPU واقعی است و بنابراین استفاده از شبیه ساز به تنهایی منجر به بار کاری در طول مهاجرت می شود که دور از واقعیت است. بنابراین، **ما یک شبه مهاجرت ابداع کردیم** که شبیه ساز را با یک ماشین فیزیکی واقعی ترکیب می کند تا حجم کار را در یک محیط اجرایی نزدیک به واقعیت بازتولید کند.

از آزمایش‌های مشاهده‌ای خود، تأیید کردیم که تنها بخشی از یک صفحه ۴ کیلوبایتی به‌طور متوسط برای بسیاری از بارهای کاری نوشته می‌شود، و بنابراین، حفاظت از نوشتن صفحه فرعی احتمالاً میزان انتقال حافظه را به میزان قابل توجهی کاهش می‌دهد. ما همچنین تشخیص دادیم که بیشتر خطاهای صفحه اضافی ناشی از استفاده از محافظت از نوشتن صفحه فرعی به دلیل یک دستورالعمل تکراری در هسته مهمان بود. بنابراین، به منظور کاهش چنین خطاهای صفحه تکراری، ما یک بهینه سازی طراحی کردیم که صفحات فرعی را که در آینده نزدیک به آنها دسترسی خواهند داشت پیش بینی می کند و در صورت شناسایی چنین دستورالعملی، حفاظت از نوشتن را از قبل حذف می کند.

برای تعیین کمیت اثربخشی رویکردمان، آزمایش‌های عملکردی انجام دادیم و نشان دادیم که حفاظت از نوشتن زیرصفحه قادر است میزان انتقال حافظه را کاهش دهد و به همان سطحی از زمان مهاجرت دست یابد که روش فشرده‌سازی دیفرانسیل اجرا شده در QEMU، به نام Xor Binary Zero Run Length Encoding (XBZRLE) [۵]، با سربار CPU کمتر و مصرف حافظه کمتر. علاوه بر این، در آزمایش‌های بنچمارک SPEC CPU 2017، متوجه شدیم که فقط حفاظت از

نوشتن صفحه فرعی می‌تواند انتقال زنده VM را در یک حجم کاری کامل کند. در این حجم کاری، ما همچنین دریافتیم که با بهینه‌سازی ما برای کاهش خطاهای صفحه، سربار CPU در طول مهاجرت می‌تواند ۲۵.۵ درصد کاهش یابد (از ۸۰.۰٪ به ۵۴.۵٪).

مشارکت های این مقاله به شرح زیر است.

- **ما یک روش جدید برای کاهش میزان انتقال حافظه در VM live migration با استفاده از ویژگی جدید CPU، حفاظت از نوشتن صفحه فرعی، برای تشخیص نوشتن ریزدانه پیشنهاد کرده‌ایم.**
- **ما یک شبه مهاجرت ابداع کرده‌ایم که یک شبیه‌ساز را با یک ماشین واقعی ترکیب می‌کند تا حجم کاری واقعی را در طول مهاجرت بدون یک CPU واقعی بازتولید کند.**
- **ما تشخیص داده‌ایم که بیشتر خطاهای صفحه اضافی در استفاده از محافظت از نوشتن صفحه فرعی ناشی از دستورالعمل‌های تکراری است و یک تکنیک بهینه‌سازی برای جلوگیری از آنها نشان داده‌ایم.**
- **ما میزان اثربخشی بهره برداری از حفاظت از نوشتن صفحه فرعی را تعیین کرده ایم و نشان داده ایم که می تواند به همان کاهشی که روش مبتنی بر نرم افزار استاندارد با مصرف منابع کمتر است، دست یابد.**

ادامه این مقاله به شرح زیر سازماندهی شده است. بخش دوم پیشینه مهاجرت زنده VM و تشخیص نوشتن ریزدانه را توضیح می دهد. بخش III نتایج آزمایش‌های مشاهده‌ای ما را ارائه می‌کند. بخش IV طراحی ما برای تشخیص نوشتن صفحه

فرعی با استفاده از حفاظت از نوشتن صفحه فرعی را توضیح می دهد. بخش V اجرای تشخیص نوشتن صفحه فرعی را با استفاده از Intel SPP نشان می دهد. بخش ششم اثربخشی حفاظت از نوشتن زیرصفحه را در انتقال زنده VM با آزمایش‌های مبتنی بر شبیه‌ساز نشان می‌دهد. بخش هفتم برخی از کارهای مرتبط را معرفی می کند و بخش هشتم این مقاله را خلاصه می کند.

II. زمینه

در این بخش، دانش پس زمینه در مورد VM را ارائه می دهیم

مهاجرت زنده ابتدا نحوه انتقال زنده VM را توضیح خواهیم داد

در بخش II-A کار می کند و سپس در مورد جزئیات نوشتن بحث می کند

تشخیص و انتقال حافظه در بخش II-B.

A. VM live migration

سه روش اصلی برای انتقال زنده VM وجود دارد:

روش پیش کپی [۲]، **روش پس از کپی** [۲۱]، و **روش ترکیبی** که هر دو را ترکیب می کند [۲۲]. در این مقاله، ما بر روی **روش پیش کپی** تمرکز می کنیم که متداول ترین روش مورد استفاده است.

روش پیش کپی به دو مرحله تقسیم می شود. **ابتدا در مرحله گرم کردن، کپی حافظه از VM مبدا به VM مقصد به صورت تکراری انجام می شود تا محتویات حافظه تقریباً همگام شوند. سپس در مرحله توقف و کپی، VM مبدا متوقف**

می شود، آخرین همگام سازی انجام می شود و در نهایت اجرا در VM مقصد از سر گرفته می شود. با کاهش زمان صرف شده برای مرحله توقف و کپی به چند صد میلی ثانیه، کاربران می توانند احساس کنند که VM به طور مداوم در حال کار است. از آنجایی که انتقال حافظه هزینه غالب در هر دو فاز است، کاهش میزان انتقال حافظه تأثیر قابل توجهی بر عملکرد انتقال زنده VM دارد. توجه داشته باشید که در مراکز داده ابری که مهاجرت VM زنده انجام می شود، ذخیره سازی بین ماشین های مجازی به اشتراک گذاشته می شود و نیازی به انتقال داده نیست. عملیات خاص در هر تکرار در **مرحله گرم کردن** به شرح زیر است. در اولین تکرار، **کل حافظه VM منبع به VM مقصد منتقل می شود.** در تکرار دوم و **بعدی، ناحیه حافظه ای که در تکرار قبلی نوشته شده است توسط VMM شناسایی می شود** و **فقط محتویات آن ناحیه حافظه منتقل می شود.** با تکرار این فرآیند، مقدار انتقال حافظه به تدریج کاهش می‌یابد و زمانی که **مقدار حافظه‌ای که باید انتقال داده شود به زیر یک آستانه** از پیش تعیین‌شده می‌رسد، **فاز به حالت توقف و کپی تغییر می‌کند.** ما به زمان از شروع اولین تکرار تا زمانی که ماشین مجازی در مقصد از سر گرفته شود به عنوان کل زمان مهاجرت. ما همچنین به کل **مقدار انتقال حافظه در مراحل گرم کردن و توقف و کپی به عنوان کل حافظه منتقل شده اشاره می کنیم.**

B. دانه بندی تشخیص نوشتن

همانطور که در بالا ذکر شد، روش پیش کپی نیاز به شناسایی ناحیه حافظه جدید نوشته شده در **حین انتقال حافظه در هر تکرار و انتقال مجدد حافظه در آن ناحیه در تکرار بعدی دارد.** از آنجایی که مکان و فرکانس نوشتن بسته به کاربرد و

حجم کاری متفاوت است، **VMM باید به درستی و کارآمد از ناحیه جدید نوشته شده عکس بگیرد تا میزان انتقال حافظه را به حداقل برساند.** کاهش میزان انتقال حافظه در هر تکرار منجر به کاهش کل حافظه انتقالی می شود که مستقیماً منجر به کاهش زمان مهاجرت و بار شبکه می شود. علاوه بر این، اگر با هر بار تکرار مقدار حافظه منتقل شده کاهش پیدا نکند، مقدار آن کاهش می یابد. **حافظه ای که باید در یک تکرار منتقل شود نمی تواند زیر آستانه قرار گیرد و انتقال نمی تواند کامل شود.**

در مکانیسم انتقال زنده VM که در حال حاضر مورد استفاده عملی است، **دانه بندی تشخیص نوشتن اساساً در هر صفحه است.** این به این دلیل است که تشخیص نوشتن معمولاً با استفاده از بیت های کثیف اجرا می شود که یکی از **ویژگی های کمکی صفحه بندی تو در تو CPU هستند**. با این حال، برنامه لزوماً در کل صفحه نمی‌نویسد. به عنوان مثال، حتی اگر فقط چند صد بایت از یک صفحه ۴ کیلوبایتی نوشته شده باشد، کل صفحه ۴ کیلوبایتی باید منتقل شود زیرا بیت های کثیف به تنهایی نمی توانند بایت های واقعی نوشته شده را شناسایی کنند. اگر تعداد نوشته های جزئی زیادی در یک صفحه وجود داشته باشد، مقدار انتقال حافظه در هر تکرار ممکن است چند صد تا چند هزار برابر بیشتر از مقدار حافظه واقعی نوشته شده باشد.

III. مشاهده

به منظور تایید اثربخشی تشخیص نوشتن صفحه فرعی بر کاهش انتقال حافظه در انتقال زنده VM، میانگین تعداد صفحات فرعی که در یک صفحه ۴ کیلوبایتی نوشته

شده بودند را اندازه گیری کردیم. در این آزمایش، ما تعداد دستورالعمل‌های اجرا شده در طول تکرار دوم انتقال زنده VM را بر روی یک ماشین فیزیکی محاسبه کردیم و تعداد صفحات فرعی نوشته شده را با بازتولید اجرا در Bochs اندازه‌گیری کردیم. بار کاری که استفاده کردیم در جدول I نشان داده شده است.

شکل ۱ نتیجه مشاهده را نشان می دهد. نوارهای خطا انحرافات استاندارد را نشان می دهند. ما متوجه شدیم که تنها یک صفحه فرعی در حدود ۱۳۰۰ صفحه ۴ کیلوبایتی نوشته شده است و تنها دو صفحه فرعی به طور متوسط در ۷۰۰ صفحه ۴ کیلوبایتی نوشته شده است. اگرچه انحراف استاندارد کمی نسبتاً بزرگ است زیرا تعداد آنها در بین بارهای کاری نسبتاً گسترده متفاوت است، روند کلی این بود که تنها بخشی از صفحات فرعی صفحات ۴-KiB نوشته شده است. بنابراین، با شناسایی و انتقال تنها چنین صفحات فرعی با استفاده از تشخیص نوشتن صفحه فرعی، انتظار می رود میزان حافظه قابل انتقال به میزان قابل توجهی کاهش یابد.

TABLE I
EXPERIMENTAL WORKLOAD

Name	Workload
idle	no specific task
kernel compile	Linux 5.5.7 compilation
redis+YCSB	Redis 5.0.3 [23] with six workloads selected from YCSB 0.17.0 [24], [25] (`workloada` to `workloadf`) [parameters] field size: 100 bytes number of fields: 10 record size: 1 KiB number of records: 256 K used memory: approximately 500 MiB
SPECCPU	13 workloads of SPEC CPU 2017 v1.0.1 (`500.perlbench_r`, `502.gcc_r`, `505.mcf_r`, `520.omnetpp_r`, `523.xalancbmk_r`, `531.deepsjeng_r`, `541.leela_r`, `557.xz_r`, `508.namd_r`, `511.povray_r`, `519.lbm_r`, `538.imagick_r`, `544.nab_r`)

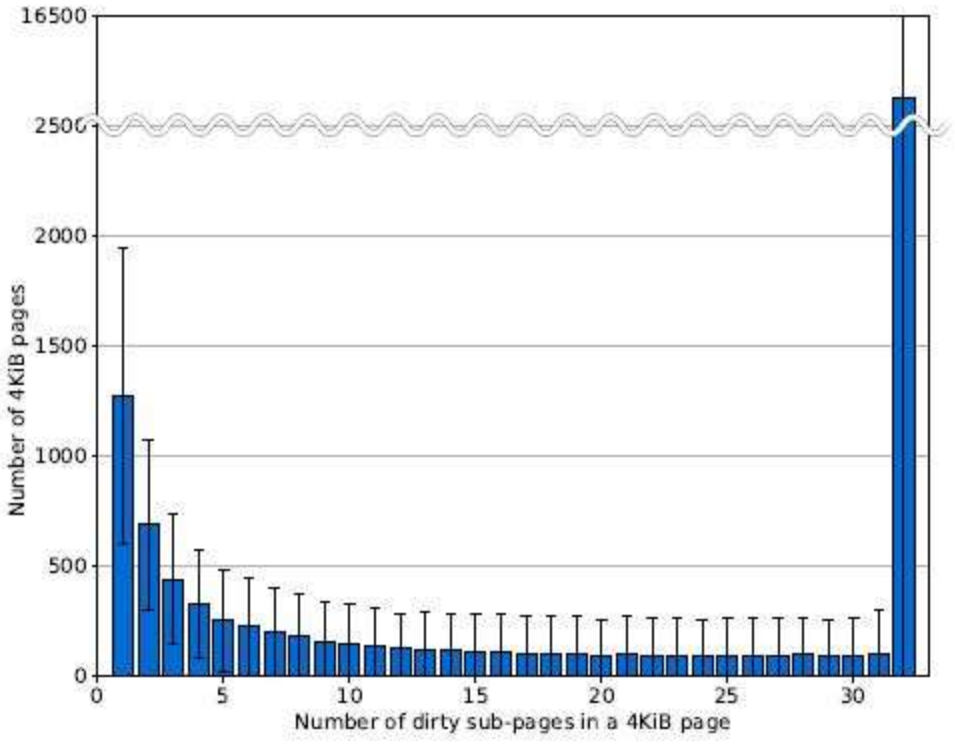

Fig. 1. Average number of sub-pages written in a 4-KiB page

از سوی دیگر، تقریباً ۱۶۰۰۰ مورد وجود داشت که تمام صفحات فرعی در یک صفحه ۴ کیلوبایتی نوشته شده بودند. ما مکان‌هایی را که چنین نوشته‌هایی در آنجا اتفاق می‌افتند شناسایی کردیم و بررسی کردیم که چه دستورالعمل‌هایی اجرا می‌شوند. در نتیجه، متوجه شدیم که یک مکان واحد بیشترین درصد دستورالعمل‌ها را در همه حجم‌های کاری به‌جز بی‌حرکتی، که در بسیاری از موارد بیش از ۹۰ درصد را شامل می‌شود، به خود اختصاص داده است. فهرست ۱ نسخه جداشده کد آن مکان را نشان می‌دهد. این تابعی در هسته لینوکس به نام clear_page_rep است که کل صفحه را با پر کردن آن با صفر پاک می کند. به طور خاص، این تابع ۸/۴۰۹۶ = ۵۱۲، تعداد تکرارها را به رجیستر ٪rcx با «movl$4096/8، %ecx» اختصاص می‌دهد، صفر را

به ثبات ٪rax 64 بیتی با «xorl %eax, %eax» اختصاص می‌دهد. "، و مقدار ٪rax(zero) را ۵۱۲ بار در ناحیه حافظه پیوسته با "repstosq" می نویسد. آدرس صفحه به طور ضمنی در ٪rdi مشخص شده است. بنابراین با شناسایی این کدها و اجتناب از خطاهای صفحه ناشی از آنها، انتظار می رود که تعداد خطاهای صفحه کاهش یابد.

Listing 1. The Linux kernel code that writes to a whole page

```
1  SYM_FUNC_START(clear_page_rep)
2  movl $4096/8,%ecx
3  xorl %eax,%eax
4  rep stosq
5  ret
6  SYM_FUNC_END(clear_page_rep)
```

IV. تشخیص نوشتن در زیر صفحه

در این بخش، روش خود را برای تشخیص نوشتن صفحه فرعی برای پایگاه انتقال زنده VM در حفاظت از نوشتن صفحه فرعی شرح می‌دهیم. ما ابتدا مکانیسم تشخیص نوشتن صفحه فرعی مبتنی بر سخت افزار را در بخش IV-A توضیح می دهیم و سپس سیستم تشخیص نوشتن صفحه فرعی را برای مهاجرت زنده VM در بخش IV-B توضیح می دهیم.

الف. حفاظت از نوشتن صفحه فرعی

همانطور که در بالا توضیح داده شد، حفاظت از نوشتن صفحه فرعی اجازه می دهد تا **حفاظت از نوشتن برای هر صفحه فرعی تنظیم شود.** حفاظت از نوشتن معمولاً با

بیت هایی که در یک ساختار داده مرتبط با جدول صفحه در صفحه بندی تودرتو نگهداری می شوند نشان داده می شود. هنگامی که سیستم عامل مهمان در یک صفحه فرعی با فعال بودن محافظت از نوشتن می نویسد، CPU یک خطای صفحه ایجاد می کند. ما این خطای صفحه را خطای صفحه فرعی می نامیم.

در صفحه بندی معمولی، جدول صفحه علاوه بر محافظت از نوشتن، یک بیت کثیف برای هر صفحه دارد. با این حال، از آنجایی که حفاظت از نوشتن زیرصفحه عمدتاً به منظور حفاظت از نوشتن برای دستگاه های سخت افزاری در مجازی سازی معرفی شده است، عملکرد بیت های کثیف را ندارد. بنابراین، برای انجام تشخیص نوشتن، باید بیت های کثیف را با محافظت از نوشتن با همکاری VMM شبیه سازی کنیم.

یک خطای صفحه فرعی به مقدار مشخصی از چرخه های CPU برای سوئیچینگ متنی بین VM و VMM نیاز دارد. علاوه بر این، یک کنترل کننده خطای صفحه فرعی باید تعدادی عملیات را انجام دهد، مانند شناسایی نوع خطای صفحه فرعی، به دست آوردن آدرس محل وقوع خطای صفحه فرعی، و به روز رسانی تنظیمات حفاظت از نوشتن. بنابراین، تعداد زیادی از خطاهای صفحه فرعی، مقدار زیادی سربار را به همراه دارند.

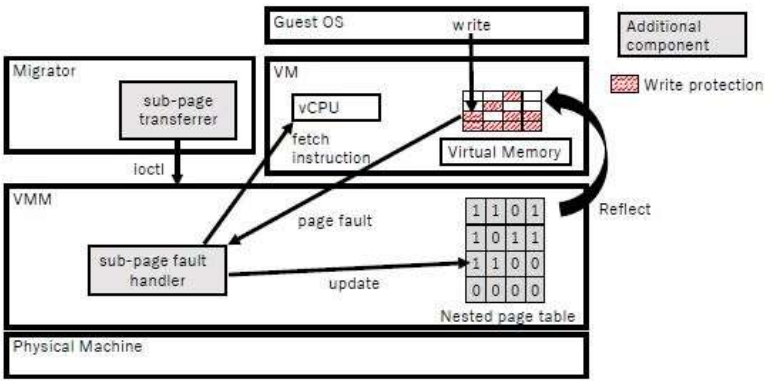

Fig. 2. The overview of our sub-page write detection system

ب. سیستم تشخیص نوشتن صفحه فرعی

شکل ۲ نمای کلی سیستم تشخیص نوشتن صفحه فرعی ما را با استفاده از مکانیسم حفاظت از نوشتن صفحه فرعی نشان می دهد. VMM جدول صفحه تودرتو را با تنظیمات حفاظت از نوشتن برای صفحات فرعی مدیریت می کند. در ابتدای هر تکرار انتقال زنده VM، VMM حفاظت از نوشتن را برای تمام صفحات فرعی در ناحیه اختصاص داده شده به حافظه فضای آدرس فیزیکی مهمان VM منبع فعال می کند. هنگامی که سیستم عامل مهمان در VM منبع به یک صفحه فرعی در فضای آدرس فیزیکی مهمان که محافظت از نوشتن برای آن فعال است می نویسد، CPU یک خطای صفحه فرعی ایجاد می کند و کنترل را به VMM می دهد. در کنترل کننده خطای صفحه فرعی، VMM آدرس صفحه فرعی که در حال نوشتن است را ثبت می کند، سپس محافظت از نوشتن را برای آن صفحه فرعی غیرفعال می کند و در نهایت اجرای VM را از سر می گیرد. با تکرار این عمل، VMM می تواند صفحات فرعی نوشته شده در ماشین مجازی را در هر تکرار ضبط کند. موازی با ضبط صفحات فرعی نوشته شده در کنترل کننده خطای صفحه فرعی، VMM تمام صفحات فرعی نوشته شده در تکرار قبلی را برشمرده است، به جز در تکرار اول که در آن فرض می شود که تمام صفحات فرعی نوشته شده اند. سپس، انتقال دهنده صفحه فرعی در migrator که جزء اضافی ما در VMM است، محتویات صفحات فرعی نوشته شده را از طریق ioctl دریافت کرده و به مقصد منتقل می کند. هنگامی که مقدار حافظه انتقالی به زیر آستانه معینی می‌رسد، سیستم وارد مرحله توقف و کپی می‌شود، جایی که انتقال حافظه نهایی انجام می‌شود.

به منظور کاهش سربار زمان اجرا ناشی از خطاهای صفحه فرعی، ما دو بهینه سازی را اتخاذ کردیم. ابتدا، در VM live migration، حتی اگر یک صفحه فرعی چندین بار در هر تکرار نوشته شود، فقط آخرین مطالب نوشته شده منتقل می شود. بنابراین، به جای فعال کردن محافظت از نوشتن بلافاصله پس از ثبت آدرس صفحه فرعی نوشته شده در خطای صفحه فرعی. کنترل کننده، حفاظت از نوشتن را در طول آن تکرار غیرفعال نگه می داریم و فعال کردن محافظت از نوشتن را تا شروع تکرار بعدی به تأخیر می اندازیم. به این ترتیب، می‌توانیم سربار خطاهای چند صفحه فرعی را که در همان صفحه فرعی در هر تکرار رخ می‌دهند، کاهش دهیم.

دومین بهینه سازی، پیش بینی ناحیه حافظه است که در آن خطای صفحه فرعی بعدی با بررسی دستورالعملی که باعث خطای صفحه فرعی شده است، رخ می دهد. به عنوان مثال، همانطور که در بخش III نشان داده شده است، اگر یک دستورالعمل به طور مکرر در مناطق متوالی حافظه بنویسد، انتظار می رود که مناطق حافظه بعد از صفحه فرعی که باعث خطای صفحه فرعی شده است نیز نوشته شود. بنابراین، به‌جای غیرفعال کردن محافظت از نوشتن هر بار که یک خطای صفحه فرعی رخ می‌دهد، می‌توانیم محافظت از نوشتن را برای صفحات فرعی متوالی از قبل زمانی که اولین خطای صفحه فرعی اتفاق می‌افتد غیرفعال کنیم و در نتیجه تعداد خطاهای زیر صفحه را کاهش دهیم.

در حال حاضر، ما فقط بهینه‌سازی را برای دستورالعمل‌های پاک کردن حافظه فرض می‌کنیم، زیرا عملکرد رضایت‌بخش است همانطور که بعداً نشان خواهیم داد، اما می‌توانیم با معرفی پیش‌بینی‌های پیشرفته‌تر عملکرد را بهبود بخشیم. توجه داشته باشید که حتی اگر پیش‌بینی اشتباه باشد و حفاظت از نوشتن بیش از حد لازم غیرفعال باشد، تنها اثر آن این است که صفحات فرعی که نوشته نشده‌اند، نوشته شده

در نظر گرفته می‌شوند. بنابراین، بر صحت مهاجرت تأثیری نخواهد داشت، اگرچه ممکن است مقدار انتقال حافظه را کمی افزایش دهد.

۷. پیاده سازی

در این مطالعه، ما از Intel 64 با SPP به عنوان معماری CPU استفاده کردیم که محافظت از نوشتن زیر صفحه را فراهم می کند. برای شبیه ساز CPU از Bochs استفاده کردیم. برای VMM، ما از یک KVM مبتنی بر وصله‌ای استفاده کردیم که پشتیبانی SPP [۲۶] را اضافه می‌کند و تشخیص نوشتن صفحه فرعی که در بخش IV-B توضیح داده شده است را پیاده‌سازی کردیم. ما همچنین برخی از عملکردهای مورد نیاز برای انتقال صفحات فرعی را در QEMU پیاده‌سازی کرده‌ایم. در این بخش، ابتدا به معرفی مختصری از Intel SPP (بخش V-A) می‌پردازیم و سپس تغییراتی را که در Bochs انجام داده‌ایم (بخش V-B) توضیح می‌دهیم. KVM (بخش V-C)، و QEMU (بخش V-D)، به ترتیب.

الف. Intel SPP

Intel SPP یک فرمت از جدول صفحه توسعه یافته (EPT) است که مکانیزم صفحه بندی تو در تو CPUهای اینتل است. این یک صفحه معمولی ۴ کیلوبایتی را به سی و دو صفحه فرعی ۱۲۸ بایتی تقسیم می‌کند و برای هر صفحه فرعی می‌توان مجوز نوشتن را تنظیم کرد. برای استفاده از SPP، ابتدا باید تابع SPP را در کنترل VM-execution ساختار کنترل ماشین مجازی (VMCS) فعال کنیم و سپس بیت SPP را در ورودی جدول صفحه برگ EPT روی ۱ قرار دهیم. SPP از ساختاری شبیه به

جدول صفحه تشکیل شده است. ما باید یک اشاره گر به جدول صفحه بالایی، به نام جدول SPP ریشه (SSPL4) در VMCS تنظیم کنیم. اگر بیت SPP ورودی EPT روی ۱ تنظیم شود، CPU یک صفحه پیاده روی از جدول صفحه SPP را شروع می کند. از SSPL4، برای شناسایی جدول برگ به نام جدول برداری (SSPL1) SPP.

SSPL1 شامل شصت و چهار ورودی ۶۴ بیتی (بردارهای SPP) است که هر کدام مربوط به یک صفحه فیزیکی مهمان ۴ کیلوبایتی است. هر بردار 64 SPP بیتی از سی و دو جفت ۲ بیتی تشکیل شده است و هر ۲ بیت مربوط به یک صفحه فرعی است. بیت بالای دو بیت، بیت اجازه نوشتن است. اگر بیت مجوز نوشتن روی ۰ تنظیم شود، زمانی که صفحه فرعی مربوطه نوشته می شود، نقض EPT (مرتبط با خطای صفحه) رخ می دهد. در این زمان، نرم افزار می تواند با بررسی بیت SPP ورودی EPT مربوطه تعیین کند که آیا خطای EPT در EPT رخ داده است یا در SPP.

ب. تغییرات در Bochs

اینتل SPP قرار است در Ice Lake، نسل دهم پردازنده Intel Core پیاده سازی شود. با این حال، CPU های نسل Ice Lake در حال حاضر فقط برای پلتفرم های رایانه های شخصی موبایل هستند و ما نتوانستیم یک CPU با SPP پیاده سازی شده به دست آوریم. بنابراین، ما از شبیه ساز Bochs استفاده کردیم که شبیه سازی SPP را پیاده سازی می کند. ما از Bochs نسخه r13850 استفاده کردیم. متاسفانه شبیه سازی SPP این نسخه از Bochs دارای اشکالات زیر بود.

Bochs به صورت داخلی TLB ها را پیاده سازی می کند. ورودی های جدول صفحه EPT را کش می کند. طبق کتابچه راهنمای توسعه دهندگان نرم افزار اینتل [۱۹]،

بردارهای SPP نیز در TLB ذخیره می شوند (به بخش ۲۸.۳.۱ "اطلاعاتی که ممکن است در حافظه پنهان ذخیره شوند" مراجعه کنید).

با وجود این، Bochs بردارهای SPP را کش نکرد. در نتیجه، اگر به یکی از سی و دو صفحه فرعی یک صفحه فیزیکی مهمان اجازه نوشتن داده شود، ورودی آن صفحه فیزیکی مهمان با مجوز نوشتن در TLB ذخیره می شود. بنابراین، کل صفحه مجاز به نوشتن در نظر گرفته می شود و حفاظت از نوشتن SPP کار نمی کند. برای رفع این مشکل، پیاده‌سازی TLB Bochs را تغییر دادیم تا اطلاعات SPP را نیز به‌درستی ذخیره کنیم تا حفاظت از نوشتن به درستی کار کند.

ج. تغییرات در KVM

ما از KVM در لینوکس ۵.۵.۷ با وصله پشتیبانی SPP به عنوان پایه VMM استفاده کردیم که تشخیص نوشتن صفحه فرعی را پیاده سازی می کند. این وصله به KVM اجازه می دهد تا SPP را فعال کند، جداول صفحه SPP را بسازد، و نقض EPT ناشی از رویدادهای مربوط به SPP را در هنگام وقوع رسیدگی کند. بر اساس این KVM، ما تابعی را برای واکشی دستورالعملی که باعث نقض EPT شده است، به منظور تحقق بهینه سازی بر اساس دستورالعمل ها، همانطور که در بخش IV-B توضیح داده شده است، پیاده سازی کردیم.

د. تغییرات در QEMU

ما تشخیص نوشتن صفحه فرعی را بر اساس QEMU 4.2.50 پیاده سازی کرده ایم. به منظور بازیابی محتویات صفحات فرعی نوشته شده توسط ماشین های مجازی از QEMU، مکانیزمی را برای بازیابی و تنظیم مقدار بیت مجوز نوشتن یک صفحه فرعی مشخص شده و محتویات صفحه فرعی از طریق ioctl با همکاری KVM پیاده سازی کردیم. علاوه بر این، از آنجایی که QEMU هنگام انتقال حافظه، آدرس‌های فیزیکی صفحاتی را که قرار است به همراه ابرداده منتقل شوند، ارسال می‌کند، ما یک مقدار ۱ بایت جدید به ابرداده اضافه کردیم تا آدرس‌های فیزیکی را در واحدهای زیرصفحه مشخص کنیم.

VI. ارزیابی

به منظور کمی کردن اثربخشی تشخیص نوشتن صفحه فرعی بر اساس حفاظت از نوشتن صفحه فرعی در انتقال زنده VM، عملکرد سه روش زیر را با هم مقایسه کردیم: (۱) روش مرسوم تشخیص نوشتن واحد صفحه KiB-۴ («تشخیص نوشتن صفحه ۴-KiB»)، (۲) روش ترکیبی از تشخیص نوشتن صفحه KiB-۴ و XBZRLE («تشخیص نوشتن صفحه XBZRLE-۴ + KiB»)، و (۳) روش تشخیص نوشتن صفحه فرعی با استفاده از Intel SPP («تشخیص نوشتن ۱۲۸ بایتی صفحه فرعی»)، که پیشنهاد ما است.

محیط آزمایشی به شرح زیر است. ما از دستگاهی با CPU Intel Core i7-4790K و حافظه ۱۶ گیگابایتی استفاده کردیم. ما یک سیستم‌عامل مهمان را روی QEMU/KVM روی Bochs اجرا کردیم، یعنی دو ماشین مجازی تو در تو هستند. ماشین مجازی که روی Bochs (L1 VM) اجرا می‌شد یک vCPU و ۲ گیگابایت

حافظه داشت و QEMU/KVM روی VM با Debian 10 (buster) و لینوکس ۵.۵.۷ با پچ SPP اجرا می‌شد. ماشین مجازی که روی QEMU/KVM (L2 VM) اجرا می‌شد، شامل یک vCPU و ۱ گیگابایت حافظه است که Debian 10 (buster) و Linux 5.5.7 را اجرا می‌کند. پهنای باند شبکه برای انتقال زنده 1 VM گیگابیت بر ثانیه در نظر گرفته شد. اندازه حافظه مورد استفاده توسط XBZRLE برای ذخیره سازی موقت صفحات منتقل شده روی ۵۱۲ مگابایت تنظیم شد. در ادامه، VM به L2 VM اشاره می کند، مگر اینکه خلاف آن مشخص شده باشد. در ادامه این بخش، ابتدا روش شبه مهاجرت را برای کاهش شبیه سازی کند CPU Bochs در بخش VI-A توضیح می دهیم. سپس نتایج سه دسته آزمایش را نشان می دهیم. ابتدا، بخش VI-B نتایج اندازه گیری اثر کاهش انتقال حافظه بر زمان مهاجرت کل را نشان می دهد. دوم، بخش VI-C نتایج مقایسه CPU و سربار حافظه را در حین انتقال حافظه نشان می دهد. در آخر، بخش VI-D نتایج اندازه‌گیری سربار زمان اجرا در تشخیص نوشتن صفحه فرعی را نشان می‌دهد. بارهای کاری مورد استفاده در جدول I نشان داده شده است.

الف. مهاجرت شبه

ما از Bochs برای شبیه سازی استفاده کردیم. با این حال، از آنجایی که Bochs یک شبیه ساز CPU مبتنی بر نرم افزار کامل است، عملکرد آن در مقایسه با یک ماشین فیزیکی واقعی بسیار کند است. بنابراین، به منظور تخمین عملکرد بر روی یک ماشین واقعی با دقت هر چه بیشتر در حین استفاده از شبیه ساز، ما یک شبه مهاجرت را همانطور که در روش زیر نشان داده شده است، طراحی کردیم.

۱) اندازه گیری عملکرد ماشین فیزیکی: اول از همه، برای تخمین تعداد دستورالعمل هایی که می توان در واحد زمان بر روی ماشین فیزیکی اجرا کرد، دستورات در ثانیه (IPS) را روی ماشین فیزیکی با استفاده از دستور perf از قبل اندازه گیری کردیم. از آنجایی که IPS به حجم کاری بستگی دارد، ما IPS را برای هر بار کاری اندازه گیری کردیم.

۲) اندازه گیری میزان انتقال حافظه روی ماشین مجازی: در مرحله بعد، برای تخمین زمان صرف شده برای هر تکرار، میزان انتقال حافظه در هر تکرار روی ماشین مجازی را اندازه گیری کردیم. برای اولین بار، مقدار کل حافظه فیزیکی مهمان را به عنوان مقدار انتقال حافظه در نظر گرفتیم. در تکرارهای بعدی، تعداد بایت‌های صفحات (صفحات فرعی) نوشته شده روی ماشین مجازی در تکرار قبلی را محاسبه کردیم. وقتی از XBZRLE استفاده شد، اندازه داده‌ها را پس از فشرده‌سازی دیفرانسیل به عنوان مقدار انتقال حافظه در نظر گرفتیم.

۳) اجرای دستورات روی ماشین مجازی: سپس تعداد دستورات قابل اجرا در هر تکرار روی ماشین مجازی را به صورت زیر محاسبه کردیم. ابتدا برای یافتن زمان لازم برای انتقال حافظه، مقدار انتقال حافظه به دست آمده در بالا را بر پهنای باند شبکه تقسیم کردیم. دوم، برای یافتن تعداد دستورالعمل‌هایی که در هر بار کاری می‌توان روی ماشین مجازی اجرا کرد، IPS از پیش اندازه‌گیری‌شده را در زمان انتقال حافظه به‌دست‌آمده در بالا ضرب کردیم. در نهایت، تعداد دستورالعمل های محاسبه شده در بالا را روی ماشین مجازی Bochs اجرا می کنیم.

۴) مراحل بالا را تکرار کنید: مراحل بالا را آنقدر تکرار کردیم تا مقدار انتقال حافظه به اندازه ای برسد که در عرض ۳۰۰ میلی ثانیه قابل انتقال باشد. با این حال، اگر تعداد تکرارها از ۲۰ تجاوز کرد، تشخیص دادیم که انتقال نمی‌تواند کامل شود.

ب. زمان کل مهاجرت

از آنجایی که اثر کاهش انتقال حافظه با استفاده از صفحات فرعی بر مهاجرت زنده VM ارتباط نزدیکی با حجم کار و رفتار برنامه در هر تکرار دارد، ما تأثیر آن را بر حسب زمان کل مهاجرت اندازه‌گیری کردیم. شکل ۳ کل زمان مهاجرت شبه مهاجرت را برای هر بار کاری نشان می دهد. نوارهای خطا خطاهای استاندارد را نشان می دهند. نوار بالاتر از خط موجی نشان می دهد که انتقال زنده VM کامل نشده است. از آنجایی که کل زمان مهاجرت تقریباً متناسب با مقدار کل انتقال حافظه بود، نتایج نشان دهنده اثربخشی کاهش انتقال حافظه در مهاجرت واقعی VM است.

نتایج تجربی نشان می‌دهد که برای اکثر بارهای کاری،

«تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی» تقریباً به همان کاهشی در زمان مهاجرت دست یافت

"تشخیص نوشتن صفحه ۴ کیلوبایت + XBZRLE". علاوه بر این، برای چهار بار کاری SPECCPU (omnetdp_r.۵۲۰، deepsjeng_r۵۳۱.، xz_r۵۵۷.، و lbm_r.۵۱۹)، "تشخیص نوشتن صفحه KiB-۴" نمی تواند انتقال را کامل کند، در حالی که "KiB-۴ نوشتن صفحه" - detection+XBZRLE" می تواند دو مورد از آنها

را تکمیل کند (omnetdp_r,577.xz_r.۵۲۰)، و "تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی" می تواند یک انتقال دیگر را تکمیل کند (deepsjeng_r.۵۳۱). به عنوان یک مثال معمولی، ما نشان می دهیم در شکل ۴ انتقال در مقدار حافظه انتقال نیافته در هر تکرار بار کاری gcc_r.۵۰۲. در gcc_r.۵۰۲، اولین تکرار حدود ۴.۵ ثانیه و تکرار بعدی حدود ۴.۴ ثانیه طول کشید. با این حال، در تکرارهای بعدی، "تشخیص نوشتن صفحه KiB-۴" نمی تواند مقدار انتقال حافظه را به اندازه کافی کاهش دهد، در حالی که "تشخیص نوشتن صفحه KiB +XBZRLE-۴" و "تشخیص نوشتن زیر صفحه ۱۲۸ بایتی" قادر به کاهش است. برای کاهش تفاوت ها به اندازه کافی برای تکمیل انتقال در حدود ۵ تکرار. ما همچنین دریافتیم که "تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی" قادر است زمان انتقال را بیش از "تشخیص نوشتن صفحه KiB + XBZRLE-۴" برای برخی از بارهای کاری کاهش دهد. . به عنوان مثال، برای حجم کاری xz_r.۵۵۷، همانطور که در شکل ۵ نشان داده شده است، تفاوت در «تشخیص نوشتن ۱۲۸ بایتی صفحه فرعی» در ابتدا بزرگتر از «تشخیص نوشتن صفحه KiB + XBZRLE-۴» بود، اما بعداً معکوس شد و منجر به زمان مهاجرت کوتاه تر این احتمالاً به این دلیل بود که محتویات نوشته شده در حافظه به راحتی توسط XBZRLE فشرده نمی شدند. شکل ۶ نتیجه را در deepsjeng_r.۵۳۱ نشان می دهد. در این حجم کاری، تنها «تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی» توانست انتقال را کامل کند. همانطور که از نمودار مشاهده می شود، "تشخیص نوشتن صفحه KiB-۴" قادر به کاهش میزان انتقال حافظه حتی با پیشرفت تکرارها نبود.

حتی با "تشخیص نوشتن صفحه KiB + XBZRLE-۴"، مقدار انتقال حافظه به کمتر از ۱۹۲ مگابایت کاهش پیدا نکرد، که نشان می دهد انتقال در ۲۰ تکرار انجام نمی شود. از سوی دیگر، «تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی» توانست این انتقال را در

حدود ۱۷ تکرار کامل کند. این به این دلیل بود که deepsjeng_r.۵۳۱ خیلی مکرر روی حافظه می نوشت و مقادیر نوشته شده به سختی فشرده می شدند. در مقابل، «تشخیص نوشتن ۱۲۸ بایتی زیر صفحه» به طور پیوسته میزان انتقال حافظه را با هر تکرار، بدون توجه به مقدار نوشته شده، کاهش داد و در نهایت انتقال را تکمیل کرد.

متأسفانه، هنگام استفاده از "تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی" حجم کاری وجود داشت که در آن مقدار حافظه منتقل شده افزایش یافت. شکل ۷ نتیجه را در lbm_r.۵۱۹ نشان می دهد. از نمودار می بینیم که «تشخیص نوشتن ۱۲۸ بایتی صفحه فرعی» مقدار انتقال حافظه را کمی افزایش داد، در حالی که «تشخیص نوشتن صفحه ۴-KiB + XBZRLE» مقدار انتقال حافظه را در مقایسه با

"تشخیص نوشتن صفحه ۴ کیلوبایتی". این به این دلیل بود که این حجم کار روی کل صفحه ۴ کیلوبایتی به طور یکنواخت می نوشت، بنابراین با استفاده از صفحات فرعی، هیچ کاهشی در میزان انتقال حافظه وجود نداشت، در حالی که در حین انتقال، ابرداده به هر صفحه فرعی اضافه می شد و مقدار حافظه منتقل شده را افزایش می داد. با این حال، هیچ یک از روش‌های تشخیص نوشتن قادر به تکمیل انتقال این حجم کار نبودند. در مجموع، "تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی" به کاهش زمان مهاجرت یکسان یا بهتر از "نوشتن صفحه ۴ کیلوبایتی" دست یافت. تشخیص + XBZRLE، اگرچه تأثیر آن بسته به حجم کار متفاوت است.

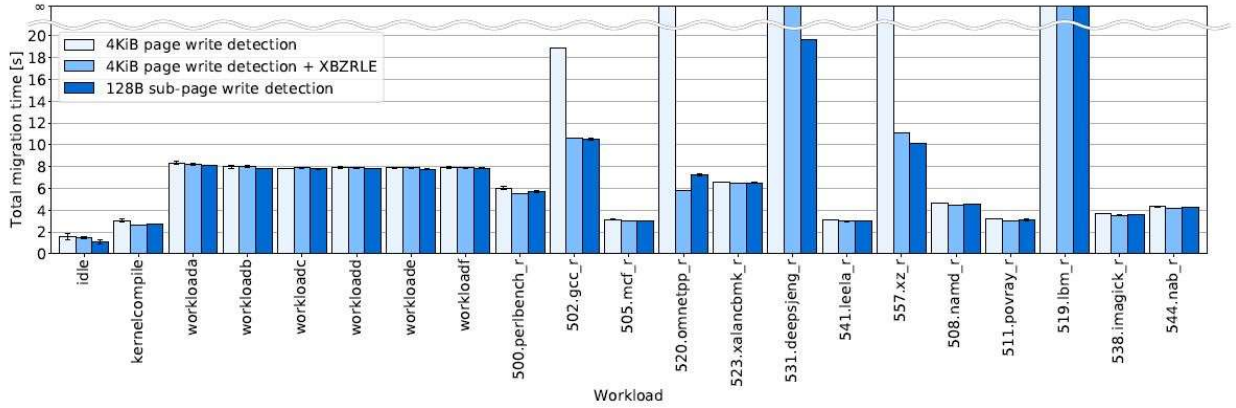

Fig. 3. Total migration time

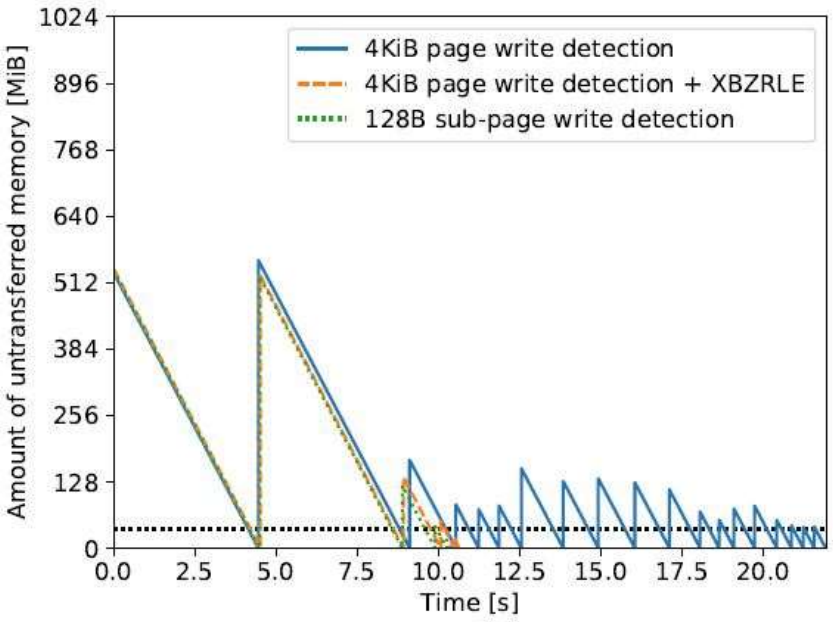

Fig. 4. Transition of the untransferred memory in `502.gcc_r`

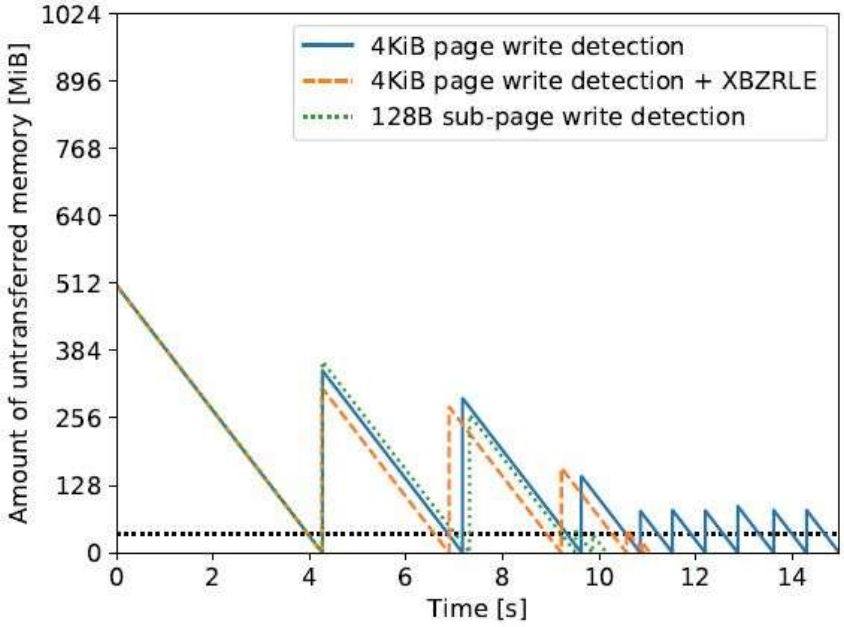

Fig. 5. Transition of the untransferred memory in `557.xz_r`

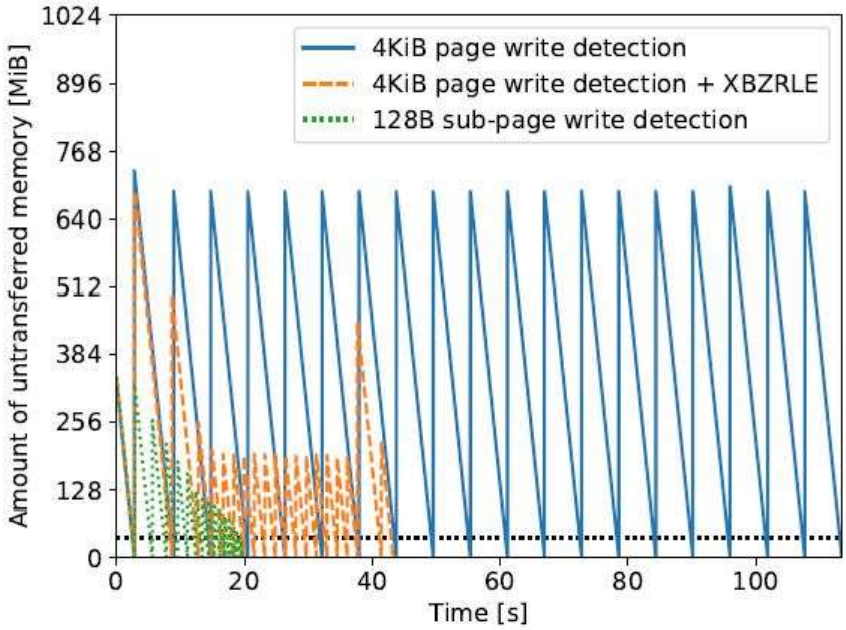

Fig. 6. Transition of the untransferred memory in `531.deepsjeng_r`

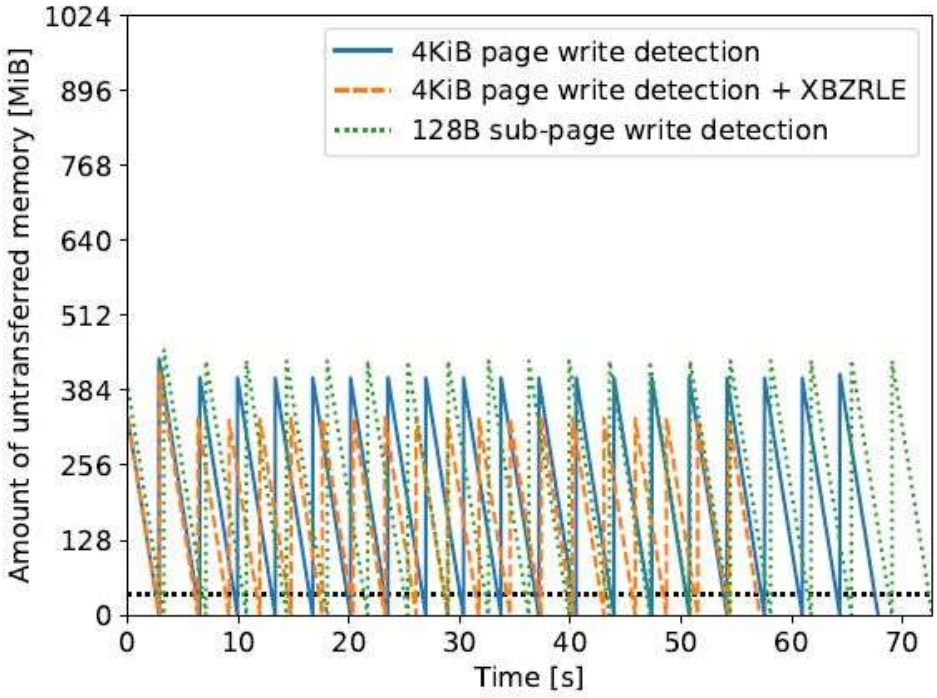

Fig. 7. Transition of the untransferred memory in `519.lbm_r`

ج. سربار در انتقال حافظه

برای تخمین سربار انتقال حافظه در هر تکرار، زمان قالب‌بندی داده‌ها برای انتقال حافظه و میزان حافظه مصرف‌شده در VMM را اندازه‌گیری کردیم.

۱) سربار CPU: برای تخمین سربار CPU در انتقال حافظه، تعداد دستورالعمل های مورد نیاز برای قالب بندی یک صفحه ۴ کیلوبایتی در QEMU را اندازه گیری کردیم. در «تشخیص نوشتن صفحه ۴-KiB»، تعداد دستورالعمل ها با اجرای ioctl برای دریافت اطلاعات صفحات نوشته شده، تعیین اینکه آیا صفحه تماماً صفر است، و قالب بندی کل صفحه به فرمت مشخص شده اندازه گیری شد. در "تشخیص نوشتن صفحه ۴-XBZRLE + KiB"، تعدادی دستورالعمل برای ذخیره صفحات در ذخیره سازی موقت و فشرده سازی صفحات نیز گنجانده شده است. در "تشخیص نوشتن ۱۲۸ بایتی صفحه فرعی"، مشابه "تشخیص نوشتن صفحه ۴-KiB" بود، با این تفاوت که VMM به جای صفحات، صفحات فرعی را به دست آورد. شکل ۸ نتایج را نشان می دهد. نوارهای خطا نشان دهنده خطاهای استاندارد هستند. در مقایسه با "تشخیص نوشتن صفحه ۴ کیلوبایت"،

تعداد دستورالعمل های مورد نیاز برای "تشخیص نوشتن صفحه ۴- + KiB XBZRLE" با ضریب ۱.۳۳ به ۱۷۵.۳۸ افزایش یافت. این به این دلیل بود که XBZRLE به تعداد زیادی دستورالعمل برای فشرده سازی از جمله مقایسه مقدار بایت به بایت نیاز دارد. تعداد دستورالعمل‌های «تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی» نیز با ضریب ۱.۲۲ به ۱۵.۸۴ افزایش یافته است که دلیل آن

افزایش پردازش هر صفحه برای مدیریت صفحات فرعی است، اما تعداد دستورالعمل‌ها کمتر از «۴» نگه داشته می‌شود. تشخیص نوشتن صفحه KiB + XBZRLE.

۲) سربار حافظه: برای تخمین سربار حافظه در انتقال حافظه، استفاده از حافظه کل ماشین میزبان (Bochs) را در شروع دومین تکرار انتقال زنده VM با استفاده از دستور free اندازه گیری کردیم. شکل ۹ نتایج را نشان می دهد. نوارهای خطا خطاهای استاندارد را نشان می دهند. نتایج نشان می دهد که "تشخیص نوشتن صفحه ۴-KiB + XBZRLE" مصرف حافظه دستگاه میزبان را ۱.۲۱ تا ۱.۸۵ برابر در مقایسه با "تشخیص نوشتن صفحه ۴-KiB" افزایش داده است. این به این دلیل است که ما حداکثر مقدار حافظه را برای ذخیره‌سازی موقت صفحه مورد استفاده در فشرده‌سازی دیفرانسیل XBZRLE روی ۵۱۲ مگابایت تنظیم کرده‌ایم، بنابراین VMM برای ذخیره حداکثر حافظه انتقال‌یافته، حافظه مصرف می‌کند. از سوی دیگر، "تشخیص نوشتن صفحه فرعی ۱۲۸ بایتی" به سختی استفاده از حافظه را افزایش داد.

D. سربار زمان اجرا

برای تخمین سربار زمان اجرا «تشخیص نوشتن ۱۲۸ بایتی صفحه فرعی» به دلیل خطاهای زیرصفحه و تأثیر بهینه‌سازی تشخیص دستورالعمل بدون شفافیت، تعداد دستورالعمل‌های اجرا شده در VMM را با ضرب تعداد مربوط به SPP محاسبه کردیم. وقایع در طول مهاجرت شبه بر اساس تعداد دستورالعمل های مورد نیاز برای مدیریت یک رویداد. شکل ۱۰ سربار نسبی را در مقایسه با

مورد بدون SPP نشان می دهد. نوارهای خطا نشان دهنده خطای استاندارد است. در حجم کاری بیکار، سربار به ۲۰۰.۷۴ درصد می رسید، اما این سربار مشکلی نداشت زیرا VM هیچ کاری انجام نمی داد. برای بارهای کاری دیگر، متوجه شدیم که سربار بارهای کاری که «تشخیص نوشتن صفحه KiB-۴» قادر به تکمیل انتقال برای آنها نبود، نسبتاً زیاد است. این بدان معناست که این بارهای کاری فرکانس نوشتن بسیار بالایی دارند. جالب اینجاست که محافظت از نوشتن زیرصفحه باعث کاهش سرعت فقط دستورالعمل‌های نوشتن می‌شود و در نتیجه سرعت نوشتن را در این نوع حجم کاری کاهش می‌دهد. در نتیجه، حفاظت از نوشتن صفحه فرعی، نرخ موفقیت مهاجرت را نه تنها از طریق تشخیص دقیق نوشتن، بلکه از طریق توانایی تنظیم سرعت نوشتن، بهبود می‌بخشد. همانطور که برای بهینه‌سازی تشخیص دستورالعمل بدون شفافیت، ما دریافتیم که حداکثر سربار شناسایی ساده نوشته ها در واحدهای صفحات فرعی ۱۲۸ بایتی ۸۰.۰ درصد بود، در حالی که با بهینه سازی می توان آن را به ۵۴.۵ درصد کاهش داد. از این نتیجه می‌توان گفت که بهینه‌سازی تشخیص دستورالعمل صفر شفاف مؤثر بود. این سربار را می توان با بهبود قابلیت پیش بینی کاهش داد، اما باید با اثر کاهش سرعت نوشتن، که کار آینده ما است، متعادل شود.

VII. کار مرتبط

یکی از روش‌های تشخیص نوشته‌ها با دانه‌بندی دقیق‌تر از صفحه، محاسبه تفاوت از صفحاتی است که قبلاً منتقل شده‌اند. سوارد و همکاران [۴] روشی را

برای ذخیره موقت بخشی از صفحات منتقل شده به عنوان کش و کاهش میزان انتقال حافظه با فشرده سازی دیفرانسیل پیشنهاد کرد. در این روش از XOR Binary RLE (XBRLE) به عنوان الگوریتم فشرده سازی تفاضلی استفاده می شود که در آن فشرده سازی طول اجرا پس از XORing بین صفحات انجام می شود. شریبمن و همکاران [۵] یک نسخه بهبود یافته از XBRLE، رمزگذاری طول صفر دودویی Xor (XBZRLE) را پیشنهاد کرد که در QEMU استفاده می شود. این الگوریتم با انجام فشرده‌سازی طول اجرا فقط در ناحیه‌ای که پس از XOR کردن بین صفحات صفر می‌شود، کارایی فشرده‌سازی را بهبود می‌بخشد. با این حال، از آنجایی که این روش های فشرده سازی دیفرانسیل نیاز به ذخیره بخشی از صفحات اصلی دارند، مصرف حافظه با افزایش تعداد صفحات نوشته شده افزایش می یابد. علاوه بر این، از آنجایی که نسبت فشرده سازی به مقادیر موجود در حافظه بستگی دارد، مواردی وجود دارد که فشرده سازی حتی اگر تفاوت کم باشد، چندان مؤثر نیست.

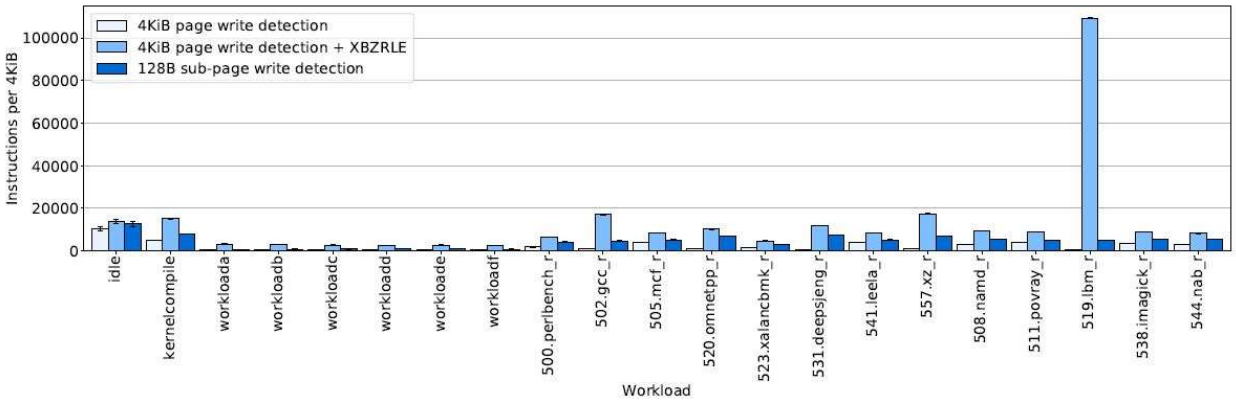

Fig. 8. The number of instructions to prepare a 4-KiB page in memory transfer

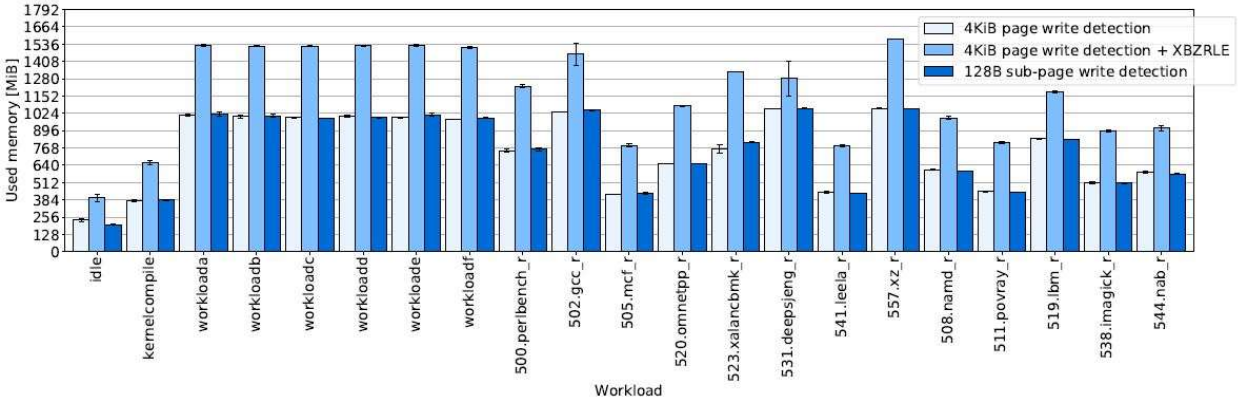

Fig. 9. The amount of memory consumed in the VMM in memory transfer

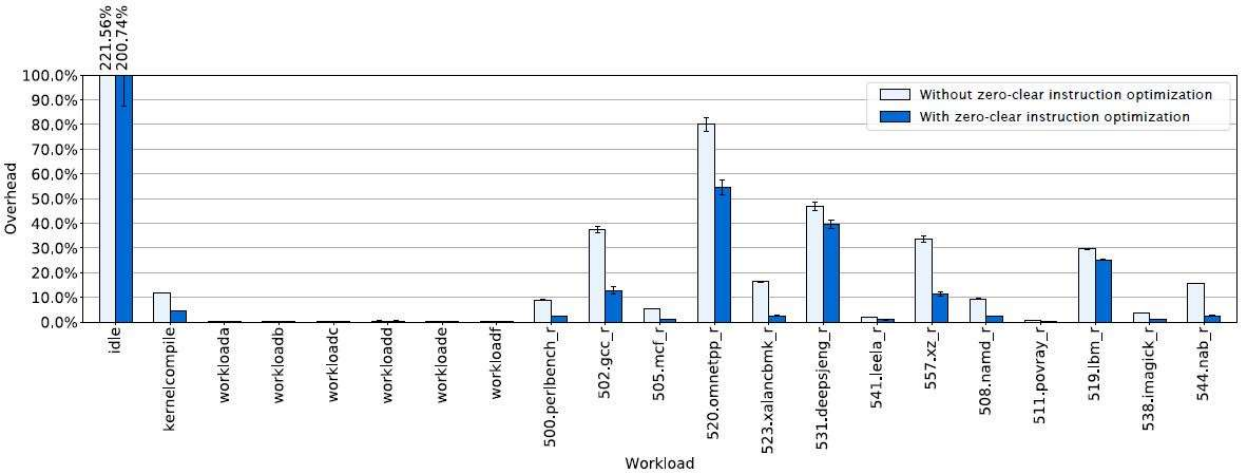

Fig. 10. The runtime overhead due to the SPP handling

روش دیگر محاسبه مقدار هش محتویات حافظه برای تشخیص ناحیه نوشته شده با دانه بندی خوب است.

وود و همکاران [۶] از روشی برای تقسیم یک صفحه به چهار صفحه فرعی و محاسبه مقادیر هش آن‌ها برای کاهش میزان انتقال حافظه برای انتقال زنده VM در شبکه‌های WAN استفاده کرد. دشپنده و همکاران [۷] روشی را برای محاسبه مقادیر هش صفحات فرعی برای کپی برداری در بین ماشین های مجازی هنگام مهاجرت چند ماشین مجازی به طور همزمان در یک محیط خوشه ای پیشنهاد کرد. لیت آل. [۱۰] روی نوشتن-نه-کثیف (نوشتن-نه-تغییر) تمرکز کرده و روشی را برای جلوگیری از انتقال چنین صفحات کثیف جعلی که کثیف هستند اما مقادیر آنها با هش اصلاح نمی‌شوند، پیشنهاد می‌کند. ژانگ و همکاران [۲۷] فناوری اثر انگشت مبتنی بر هش را برای شناسایی صفحات حافظه یکسان و مشابه معرفی کرد و از الگوریتم رمزگذاری RLE برای انجام کپی برداری استفاده کرد. انتظار می‌رود روش‌های مبتنی بر هش انتقال حافظه را کاهش دهند، اما زمانی که اندازه صفحه فرعی کاهش می‌یابد، حافظه بیشتری مصرف می‌کنند و برای مقابله با برخورد هش، نیاز به مقایسه بایت به بایت دارند.

اگر ماشین میزبان منابع کافی داشته باشد، این روش‌های مبتنی بر نرم‌افزار ممکن است بتوانند محاسبات و سربار حافظه را از طریق پردازش موازی پنهان کنند. با این حال، در موارد استفاده در دنیای واقعی، ماشین میزبان همیشه منابع کافی ندارد. به عنوان مثال، Birke و همکاران. [۲۸] تجزیه و تحلیلی را روی یک ابر خصوصی انجام داد و دریافت که بیش از ۵۰ درصد از ماشین‌های فیزیکی تقریباً تمام حافظه واقعی خود را به ماشین‌های مجازی اختصاص می‌دهند. بنابراین، اگر ماشین میزبان حافظه بیشتری را برای انتقال زنده VM مصرف کند زمانی که ماشین‌های مجازی تقریباً حافظه را اشغال می‌کنند، ممکن است منجر به کاهش عملکرد VM شود. فشرده‌سازی حافظه یک روش موثر

برای کاهش میزان حافظه انتقال حافظه در VM live migration است. و همکاران [۸] روشی را پیشنهاد کرد که یک الگوریتم فشرده سازی مؤثر را بر اساس محتوای صفحه ای که قرار است منتقل شود انتخاب می کند. لی و همکاران [۹] روشی را برای تغییر پویا نسبت فشرده سازی با توجه به پهنای باند شبکه پیشنهاد کرد. فشرده سازی حافظه با استفاده از Zlib در QEMU/KVM نیز پیاده سازی شده است. با این حال، فشرده‌سازی حافظه معمولاً با افزایش نسبت فشرده‌سازی، زمان CPU بیشتری مصرف می‌کند و نسبت فشرده‌سازی بسته به محتویات و الگوریتم فشرده‌سازی بسیار متفاوت است. فشرده سازی حافظه را می توان همراه با تشخیص نوشتن صفحه فرعی استفاده کرد. با این حال، اگر اندازه داده ها خیلی کوچک باشد، نسبت فشرده سازی کاهش می یابد. بنابراین، لازم است روشی مانند فشرده سازی چندین صفحه فرعی به طور همزمان ابداع شود. پیش‌بینی الگوهای دسترسی به نوشتن حافظه و نادیده گرفتن انتقال صفحاتی که اغلب نوشته می‌شوند رویکرد دیگری است. کلارک و همکاران [۲] و لین و همکاران. [۱۱] الگوریتمی را پیشنهاد کرد که صفحاتی را که در هر دو تکرار متوالی کثیف شده اند را از انتقال در تکرارهای بعدی حذف می کند.

هو و همکاران [۱۲] و شی و همکاران. [۱۳] روشی را برای پیش‌بینی دسترسی‌های حافظه در چندین تکرار با استفاده از الگوریتم پیچیده‌تری پیشنهاد کرد. علمداری [۲۹] مفهوم فاصله استفاده مجدد را برای کاهش تعداد انتقال صفحات کثیف اتخاذ کرد. این روش ها همچنین ممکن است برای تشخیص نوشتن زیر صفحه اعمال شوند. حذف نواحی غیر ضروری از حافظه VM از قبل نیز در کاهش مقدار کل حافظه منتقل شده موثر است. هاینز و

همکاران [۱۴] روشی را برای حذف دینامیک حافظه بلااستفاده با استفاده از بالون کردن حافظه پیشنهاد کرد. مائت آل. [۱۵] روشی را برای به دست آوردن صفحات اختصاصی مجازی واقعی بر اساس اطلاعات سیستم عامل مهمان و انتقال تنها آن صفحات پیشنهاد کرد. کوتو و همکاران [۳۰] از انتقال صفحات نرم، مانند صفحات رایگان یا صفحات کش فایل اجتناب می کند. این روش‌ها ممکن است همراه با تشخیص نوشتن صفحه فرعی استفاده شوند. موتور محاسباتی Google (GCE) مهاجرت زنده [۳۱] روش‌های پیش‌کپی و پس‌کپی را ترکیب می‌کند. برای اکثر موارد از روش پیش کپی استفاده می‌کند، اما زمانی که ماشین‌های مجازی دارای مجموعه کثیف زیاد و سرعت نوشتن بالا هستند، به روش پس‌کپی تغییر می‌کند. این رویکرد نیز متعامد با رویکرد ما است. یک رویکرد جالب دیگر کاهش غیرمستقیم مقدار انتقال حافظه با محدود کردن حجم کار VM است. Clark et al. [۲] روشی را برای کاهش تعداد صفحات کثیف با توقف فرآیندی که بیشترین خطاهای صفحه را در سیستم عامل مهمان ایجاد می کند، پیشنهاد کرد. جین و همکاران [۱۶] روشی را برای کاهش زمان اجرای vCPU پیشنهاد کردند، و Yqiu و همکاران. [۱۷] روشی را برای افزودن وظایف خواب به صف وظایف سیستم عامل مهمان پیشنهاد کرد.

QEMU/KVM همچنین روشی را برای محدود کردن فعالیت vCPU پیاده‌سازی می‌کند [۱۸]. حفاظت از نوشتن صفحه فرعی این ویژگی را دارد که با افزایش فرکانس نوشتن حافظه، تعداد خطاهای صفحه فرعی افزایش می یابد. بنابراین، می توان آن را برای تنظیم خودکار فرکانس نوشتن حافظه اعمال کرد. چک پوینت و پخش مجدد [۳۲]، [۳۳] میزان انتقال حافظه را با رویکردی که وضعیت VM را به طور کامل بازیابی نمی کند، بسیار کاهش می دهد. در این

رویکرد، حالت VM با چک پوینت از قبل منتقل می شود و سپس تنها حداقل مقدار گزارش برای پخش مجدد رفتار ماشین مجازی در مقصد ارسال می شود. با این حال، محتویات حافظه VM ممکن است به طور کامل بازتولید نشود، که ممکن است باعث مشکلات سازگاری شود. حافظه مشترک توزیع شده (DSM) [۳۴] یک روش کلاسیک برای همگام سازی حافظه در چندین ماشین است. [۳۵] یک DSM را با استفاده از تشخیص نوشتن با مجوزهای نوشتن EPT پیشنهاد کرد تا متوجه شود ماشین های مجازی در حال اجرا در چندین ماشین فیزیکی توزیع شده اند. زکائوسکاس و همکاران [۳۶] با اصلاح کامپایلر، تشخیص نوشتن ریزدانه را پیشنهاد کرد و تا ۴۴ درصد کاهش در انتقال داده به دست آورد.Schoina و همکاران. [۳۷] اجازه نوشتن ریز دانه را با تغییر رابط حافظه و درج کد قبل از دستورالعمل‌های نوشتن حافظه پیاده‌سازی کرد. برخی از این تکنیک‌های DSM ممکن است برای مهاجرت زنده VM اعمال شوند.

هشتم. خلاصه

در این مطالعه، ما اثربخشی تشخیص نوشتن زیر صفحه را با استفاده از مکانیسم حفاظت از نوشتن صفحه فرعی در مهاجرت زنده VM برآورد کردیم. حفاظت از نوشتن صفحه فرعی تابعی است که می تواند محافظت از نوشتن را برای هر صفحه فرعی تنظیم کند، و می تواند تشخیص نوشتن را در واحدهای صفحات فرعی با مصرف حافظه کمتر متوجه شود، در حالی که هزینه خطای صفحه را برای هر دسترسی نوشتن به صفحه فرعی متحمل می شود. با حفاظت از نوشتن. ما معماری CPU را با Intel SPP هدف قرار دادیم که اولین موردی است که از

محافظت از نوشتن زیر صفحه پشتیبانی می کند، تشخیص نوشتن زیر صفحه و انتقال شبه را در QEMU/KVM پیاده سازی کرده است، بارهای کاری مختلفی را روی Bochs اجرا کردیم که می تواند Intel SPP را شبیه سازی کند. و اثر کاهش کل زمان مهاجرت و سربار را بر روی CPU و حافظه تخمین زد. نتایج تجربی نشان می‌دهد که تشخیص نوشتن صفحه فرعی با محافظت از نوشتن صفحه فرعی به همان سطح کاهش انتقال حافظه و کاهش کل زمان مهاجرت دست می‌یابد. روشی ترکیبی از تشخیص نوشتن صفحه ۴ کیلوبایتی و XBZRLE در QEMU، در حالی که فقط مقدار کمی مصرف حافظه را افزایش می‌دهد. علاوه بر این، برای SPECCPU 531.deepsjeng_r، فقط حفاظت از نوشتن زیر صفحه قادر به تکمیل انتقال بود. از سوی دیگر، متوجه شدیم که برخی از بارهای کاری وجود دارد که حتی تشخیص نوشتن صفحه فرعی نیز نمی تواند انتقال را کامل کند. برای کارهای آینده، باید عملکرد یک ماشین واقعی مجهز به CPU را ارزیابی کنیم که از نوشتن صفحه فرعی پشتیبانی می کند. حفاظت. به منظور کاهش سربار خطاهای صفحه ناشی از SPP، باید بهینه سازی های بیشتری مانند اجتناب از خطاهای صفحه با پیش بینی نوشتن در هر صفحه فرعی را در نظر بگیریم. علاوه بر این، ما نیاز به ارزیابی و کاهش هزینه حفظ سازگاری داده‌ها برای حفاظت از نوشتن صفحه فرعی در چندین هسته CPU داریم، که برای صفحات معمولی نیز مشکل است، اما انتظار می‌رود که برای صفحات فرعی حتی بیشتر افزایش یابد. در نهایت، حفاظت از نوشتن صفحه فرعی ممکن است برای تنظیم سرعت نوشتن برای مدیریت بارهای کاری که با روش‌های معمولی تکمیل نمی‌شوند، استفاده شود.

REFERENCES

[1] T. Le, “A survey of live Virtual Machine migration techniques,” Computer Science Review, vol. 38, pp. 1–17, Nov. 2020.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live migration of virtual machines,” in Proceedings of the 2nd Symposium on Networked Systems Design & Implementation, ser. NSDI ’05, 2005, pp. 273–286.

[3] S. Akiyama, T. Hirofuchi, and S. Honiden, “Evaluating impact of live migration on data center energy saving,” in Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science, ser. CloudCom 2014, Dec. 2014, pp. 759–762.

[4] P. Sv ¨ ard, B. Hudzia, J. Tordsson, and E. Elmroth, “Evaluation of delta compression techniques for efficient live migration of large virtual machines,” in Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE ’11, Mar. 2011, pp. 111–120.

[5] A. Shribman and B. Hudzia, “Pre-copy and post-copy vm live migration for memory intensive applications,” in Proceedings of the 18th International Conference on Parallel Processing Workshops, ser. Euro-Par 2012, Aug. 2012, pp. 539–547.

[6] U. Deshpande, X. Wang, and K. Gopalan, “Live gang migration of virtual machines,” in Proceedings of the 20th International Symposium on High Performance Distributed Computing, ser. HPDC ’11, Jun. 2011, pp. 135–146.

[7] T. Wood, K. K. Ramakrishnan, P. Shenoy, J. Van Der Merwe, J. Hwang, G. Liu, and L. Chaufournier, “Cloudnet: Dynamic pooling of cloud resources by live wan migration of virtual machines,” IEEE/ACM Transactions on Networking, vol. 23, no. 5, p. 1568–1583, Oct. 2015.

[8] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, “Live virtual machine migration with adaptive memory compression,” in Proceedings of 2009 IEEE International Conference on Cluster Computing and Workshops, ser. CLUSTER 2009, 31 Aug.–4 Sep. 2009, pp. 1–10.

[9] C. Li, D. Feng, Y. Hua, W. Xia, L. Qin, Y. Huang, and Y. Zhou, “Bac: Bandwidth-aware compression for efficient live migration of virtual machines,” in Proceedings of the 36th IEEE International Conference on Computer Communications, ser. INFOCOM 2017, May 2017, pp. 1–9.

[10] C. Li, D. Feng, Y. Hua, and L. Qin, “Efficient live virtual machine migration for memory write-intensive workloads,” Future Generation Computer Systems, vol. 95, pp. 126–139, Jun. 2019.

[11] C. Lin, Y. Huang, and Z. Jian, “A two-phase iterative pre-copy strategy for live migration of virtual machines,” in Proceedings of the 8th International Conference on Computing Technology and Information Management, ser. ICCM 2012, Apr. 2012, pp. 29–34.

[12] B. Hu, Z. Lei, Y. Lei, D. Xu, and J. Li, “A time-series based precopy approach for live migration of virtual machines,” in Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed

Systems, ser. ICPADS 2011, Dec. 2011, pp. 947–952.
[13] B. Shi and H. Shen, “Memory/disk operation aware lightweight vm live migration across data-centers with low performance impact,” in In Proceedings of the 38th IEEE Conference on Computer Communications, ser. INFOCOM 2019, 29 Apr.–2 May 2019, pp. 334–342.
[14] M. R. Hines and K. Gopalan, “Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning,” in Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE ’09, Mar. 2009, p. 51–60.
[15] Y. Ma, H. Wang, J. Dong, Y. Li, and S. Cheng, “Me2: Efficient live migration of virtual machine with memory exploration and encoding,” in Proceedings of the 2012 IEEE International Conference on Cluster Computing, ser. CLUSTER 2012, Sep. 2012, pp. 610–613.
[16] H. Jin, W. Gao, S. Wu, X. Shi, X. Wu, and F. Zhou, “Optimizing the live migration of virtual machine by CPU scheduling,” Journal of Network and Computer Applications, vol. 34, no. 4, pp. 1088–1096, Jul. 2011.
[17] F. Yiqiu, Z. Chen, and G. Junwei, “Improvement on live migration of virtual machine by limiting the activity of CPU,” in Proceedings of the 2017 International Conference on Computer Systems, Electronics and Control, ser. ICCSEC 2017, Dec. 2017, pp. 1420–1424.
[18] QEMU/KVM. Autoconverge Live Migration. [Online]. Available: https://wiki.qemu.org/Features/AutoconvergeLiveMigration
[19] Intel, Intel 64 and IA-32 Architectures Software Developer’s Manual. Intel, Nov. 2020. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm
[20] Bochs. [Online]. Available: http://bochs.sourceforge.net/
[21] M. R. Hines, U. Deshpande, and K. Gopalan, “Post-copy live migration of virtual machines,” ACM SIGOPS Operating Systems Review, vol. 43, no. 3, p. 14–26, Jul. 2009.
[22] S. Sahni and V. Varma, “A hybrid approach to live migration of virtual machines,” in Proceedings of the 2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), Oct. 2012, pp. 1–5.
[23] Redis. [Online]. Available: https://redis.io/
[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in Proceedings of the 1st ACM Symposium on Cloud Computing, ser. SoCC ’10, Jun. 2010, pp. 143–154.
[25] YCSB. [Online]. Available: https://github.com/brianfrankcooper/YCSB
[26] QEMU/KVM. SPP-Patch. [Online]. Available: https://lore.kernel.org/kvm/20200516125507.5277-1-weijiang.yang@intel.com/
[27] X. Zhang, Z. Huo, J. Ma, and D. Meng, “Exploiting data deduplication to accelerate live virtual machine migration,” in 2010 IEEE International Conference on Cluster Computing, 2010, pp. 88–96.
[28] R. Birke, A. Podzimek, L. Y. Chen, and E. Smirni, “Virtualization in the private cloud: State of the practice,” IEEE Transactions on Network and Service Management, vol. 13, no. 3, pp. 608–621, Sep. 2016.
[29] J. F. Alamdari and K. Zamanifar, “A reuse distance based precopy

approach to improve live migration of virtual machines," in 2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing, 2012, pp. 551–556.
[30] A. Koto, H. Yamada, K. Ohmura, and K. Kono, "Towards unobtrusive vm live migration for cloud computing platforms," in Proceedings of the Asia-Pacific Workshop on Systems, ser. APSYS '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online].
Available: https://doi.org/10.1145/2349896.2349903
[31] A. Ruprecht, D. Jones, D. Shiraev, G. Harmon, M. Spivak, M. Krebs, M. Baker-Harvey, and T. Sanderson, "Vm live migration at scale," SIGPLAN Not., vol. 53, no. 3, p. 45–56, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296975.3186415
[32] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu, "Live migration of virtual machine based on full system trace and replay," in Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, ser. HPDC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 101–110. [Online]. Available: https://doi.org/10.1145/1551609.1551630
[33] T. Knauth and C. Fetzer, "Vecycle: Recycling vm checkpoints for faster migrations," in Proceedings of the 16th Annual Middleware Conference, ser. Middleware '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 210–221. [Online]. Available: https://doi.org/10.1145/2814576.2814731

[34] K. Li, "Ivy: A shared virtual memory system for parallel computing," in Proceedings of the 1988 International Conference on Parallel Processing, ser. ICPP 1988, 1988, pp. 94–101.
[35] J. Zhang, Z. Ding, Y. Chen, X. Jia, B. Yu, Z. Qi, and H. Guan, "GiantVM: A type-ii hypervisor implementing many-to-one virtualization," in In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ser. VEE '20, Mar. 2020, pp. 30–44.
[36] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, "Software write detection for a distributed shared memory," in Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation, ser. OSDI '94, Nov. 1994.
[37] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," in Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS VI, Nov. 1994, pp. 297–306.