Journal of Soft Computing and Information Technology (JSCIT)
Babol Noshirvani University of Technology, Babol, Iran
Journal Homepage: jscit.nit.ac.ir
Volume 9, Number 3, Fall 2020, pp. 255-270
Received:11/29/2019, Revised: 07/14/2020, Accepted: 08/06/2020

A New Flash Translation Layer with In-Block Update Capability

Reza Gholami Taghizadeh[1], Mohammadreza Binesh Marvasti[2], Seyyed Amir Asghari[3*] and Ramin Gholami Taghizadeh[4]
1- Department of Electrical and Computer Engineering, Kharazmi University, Tehran, Iran.
2- Department of Electrical and Computer Engineering, Kharazmi University, Tehran, Iran.
3*- Department of Electrical and Computer Engineering, Kharazmi University, Tehran, Iran.
4- Department of Electrical and Computer Engineering, Kharazmi University, Tehran, Iran.
[1]std_reza.taghizadeh@khu.ac.ir, [2]marvasti@khu.ac.ir, [3*]asghari@khu.ac.ir, and [4]Std_ramin.gholami.taghizadeh@khu.ac.ir

Corresponding author's address: Seyyed Amir Asghari, Faculty of Engineering, Kharazmi University of Tehran, Iran.

Abstract- The emergence of non-volatile NAND flash memory led to a new generation of Solid State Drives (SSD). One of the most important features of this type of drive is to update the data sectors out of place. In SSDs, a middleware called Flash Translation Layer (FTL) is used to hide such features from the operating system. The most important tasks consist of address translation, garbage collection, and wear leveling effect. Address translation has a great effect on the efficiency and read/write speed within SSDs. In this paper, we propose a new address translation scheme based on data compression, called *In Block Update FTL*. In the proposed scheme, the required memory of the address mapping table has been significantly reduced. Moreover, our extensive experimental results show that the proposed FTL scheme outperforms previous FLT schemes in the read and write operations under real workloads.

Keywords- NAND Flash Memory, Solid State Drive, SSD, Flash Translation Layer, Address Translation Unit, Data Compression in SSD

مجله علمی پژوهشی رایانش نرم و فناوری اطلاعات
دانشگاه صنعتی نوشیروانی بابل
صفحه مجله: jscit.nit.ac.ir
جلد ۹، شماره ۳، پائیز ۱۳۹۹ ، صفحه ۲۵۵-۲۷۰
دریافت: ۱۳۹۸/۰۹/۰۸، بازنگری: ۱۳۹۹/۰۴/۲۴، پذیرش: ۱۳۹۹/۰۵/۱۶

یک لایه ترجمه فلش جدید با قابلیت بروزرسانی درون بلوکی

رضا غلامی تقی زاده[۱]، محمدرضا بینش مروستی[۲]، سید امیر اصغری*[۳]، رامین غلامی تقی زاده[۴]
۱- دانشکده مهندسی برق و کامپیوتر- دانشگاه خوارزمی- تهران- ایران.
۲- دانشکده مهندسی برق و کامپیوتر- دانشگاه خوارزمی- تهران- ایران.
*۳- دانشکده مهندسی برق و کامپیوتر- دانشگاه خوارزمی- تهران- ایران.
۴- دانشکده مهندسی برق و کامپیوتر- دانشگاه خوارزمی- تهران- ایران.
[1]std_reza.taghizadeh@khu.ac.ir, [2]marvasti@khu.ac.ir, [3*]asghari@khu.ac.ir, [4]std_ramin.gholami.taghizadeh@khu.ac.ir

* نشانی نویسنده مسئول: سید امیر اصغری، تهران، خیابان شهید مفتح، دانشگاه خوارزمی تهران، دانشکده فنی و مهندسی.

چکیده- ظهور حافظه‌های پایدار نیمه هادی از نوع NAND فلش منجر به تولید نسل جدیدی از حافظه‌های جانبی به نام درایوهای حالت جامد (SSD) شد. یکی از مهمترین ویژگی‌های این نوع از درایوها به‌روزرسانی سکتورهای داده به صورت بیرون از مکان است. در SSDها برای مخفی کردن چنین ویژگی‌هایی از دید سیستم عامل، از یک بخش میان‌افزاری به نام لایه ترجمه فلش (FTL) استفاده می‌کنند. وظایف این بخش شامل ترجمه آدرس، زباله‌روبی و پخش فرسودگی است. در این مقاله یک طرح جدید برای لایه ترجمه فلش بر مبنای فشرده‌سازی داده‌ها به نام لایه ترجمه فلش با قابلیت به‌روزرسانی درون بلوکی (In Block Update FTL) پیشنهاد شده است. در این طرح پیشنهادی، حافظه مورد نیاز برای جدول نگاشت آدرس به میزان قابل توجهی کاهش یافته است. همچنین نتایج شبیه سازی با حجم‌کاری (Workload) واقعی نشان می دهد که سرعت خواندن و نوشتن روش مذکور نسبت به طرح مشابه قبلی به میزان قابل قبولی بهبود یافته است.

واژه‌های کلیدی: حافظه‌ی NAND فلش – درایو حالت جامد – لایه ترجمه فلش – واحد ترجمه آدرس – فشرده‌سازی داده‌ها در SSD.

۱- مقدمه

استفاده از فناوری‌های سیلیکونی در ساخت حافظه‌های غیر فرار منجر به ظهور نسل جدیدی از دستگاه‌های ذخیره‌ساز به نام درایوهای حالت جامد (SSD) شد. از آنجایی که SSDها از قطعات الکترونیکی تشکیل شده‌اند، دارای ویژگی‌هایی همچون اندازه کوچک، توان مصرفی پایین و مقاوم در برابر ضربات مکانیکی بوده و گزینه بسیار مناسبی برای استفاده در سیستم های نهفته و قابل حمل هستند. تراشه‌های حافظه NAND فلش یکی از متداولترین نوع حافظه‌های بکار رفته در درایوهای حالت جامد هستند. در هر تراشه حافظه فلش معمولا ۲ تا ۸ عدد Die وجود دارد و هر Die دارای سیگنال‌های کنترلی مخصوص به خود است. هر Die شامل چندین Plane بوده و هر کدام از این Planeها دارای یک یا دو عدد ثبات داده/حافظه نهان هستند. هر Plane نیز شامل هزاران بلوک فیزیکی بوده و هر بلوک معمولا از ۶۴، ۱۲۸ یا ۲۵۶ صفحه فیزیکی تشکیل شده است [۲-۱] . هر صفحه از دو بخش داده اصلی و داده یدک تشکیل شده است. اندازه داده اصلی در هر صفحه فیزیکی حافظه فلش شامل مقادیر ۲، ۴، ۸ و یا ۱۶ کیلوبایتی بوده و داده یدک بسته به کارخانه سازنده و نوع کاربرد می‌تواند حاوی اطلاعات کنترلی و کدهای تصحیح خطا باشد. در حافظه‌های فلش سه نوع عملیات خواندن، نوشتن و پاک‌کردن پشتیبانی می‌شوند. واحد

عملیات خواندن و نوشتن، صفحه است ولی واحد عملیات پاکسازی، بلوک می‌باشد [۴-۲].

حافظه‌های فلش، علیرغم مزایایی همچون سرعت بالا در خواندن و نوشتن درخواست‌ها، دارای یکسری محدودیت به دلیل ساختار درونی خود نیز هستند. اولین مشکل در این حافظه‌ها این است که عملیات خواندن و نوشتن در آن نامتقارن بوده و برای عملیات نوشتن مدت زمان بیشتری نسبت به عملیات خواندن نیاز است. دومین مشکل حافظه‌های فلش این است که نمی‌توان داده‌ها را بر روی یک صفحه از قبل نوشته شده، بازنویسی کرد. سومین مشکل حافظه‌های فلش، طول عمر محدود آن‌ها است. یک راهکار برای برطرف کردن محدودیت‌های حافظه‌های فلش، استفاده از یک سفت افزار میان سیستم فایل و حافظه فلش است که به آن لایه ترجمه فلش یا FTL می‌گویند. لایه ترجمه فلش شامل مولفه‌های مهمی همچون واحد ترجمه آدرس و واحد زباله‌روبی است [۶-۵].

برای بازنویسی یک صفحه در حافظه‌های فلش، ابتدا باید داده‌ی جدید در یک صفحه خالی نوشته شده و سپس صفحه قبلی به عنوان نامعتبر علامت‌گذاری شود. به این عملیات، بازنویسی بیرون از مکان می‌گویند. بازنویسی داده‌ها در حافظه‌های فلش، صفحات فیزیکی زیادی را نامعتبر و غیرقابل استفاده می‌کند. بلوک‌های فیزیکی حاوی صفحات نامعتبر، بلوک‌های آلوده نام دارند. هرگاه فضای خالی حافظه فلش به یک حد آستانه برسد، آنگاه لایه ترجمه فلش به کمک یک مولفه به نام واحد زباله‌روب، اقدام به پاکسازی بلوک‌های آلوده می‌کند. عملیات زباله‌روبی صفحات معتبر در بلوک‌های آلوده را به بلوک‌های خالی منتقل کرده و سپس بلوک‌های آلوده را پاک می‌کند. . فاکتور مهمی که بر عملیات زباله‌روبی تاثیر می‌گذارد، Over Provisioned Space (OPS) نام دارد. این فضا ظرفیت اضافی حافظه فلش است و همیشه آزاد نگه داشته می‌شود تا به کمک آن عملیات زباله‌روبی به طور موثر انجام گردد [۶].

سیستم فایل‌های متداول همچون FAT32 و NTFS، بر اساس ساختار دیسک‌های سخت کار می‌کنند. دیسک‌های سخت به صورت مجموعه‌ای از سکتورهای متوالی هستند که در آن قابلیت بازنویسی سکتورها وجود دارد. اما در حافظه‌های فلش، نمی‌توان یک صفحه فیزیکی را بدون پاکسازی بازنویسی کرد و برای حل این مشکل از یک مولفه به نام واحد «ترجمه آدرس» استفاده می‌شود. واحد ترجمه آدرس، سکتورهای منطقی داده را به صفحات فیزیکی حافظه فلش نگاشت می‌کند. این واحد نقش بسیار مهمی در حافظه های فلش ایفا می‌کند و نام FTL در واقع اشاره به واحد ترجمه آدرس است [۶-۵].

در حافظه‌های فلش، نوع معماری واحد ترجمه آدرس اثر بسیار زیادی در زمان تاخیر خواندن و نوشتن درخواست‌ها دارد. به همین دلیل بسیاری از تحقیقات گذشته، متمرکز بر معماری واحد ترجمه آدرس بوده است [۱۴-۵]. در کنار مسئله معماری واحد ترجمه آدرس، مسئله طول عمر مفید حافظه‌های فلش مطرح است که ناشی از محدودیت در تعداد عملیات‌های پاکسازی بلوک‌های حافظه‌های فلش است. یکی از راهکارهای پیشنهادی برای مدیریت فضای حافظه فلش و مسئله طول عمر مفید آن استفاده از روش فشرده-سازی داده‌ها در سطح حافظه فلش است. آن دسته از FTLهایی که تاکنون مطرح شده‌اند و از قابلیت فشرده‌سازی داده‌ها هم پشتیبانی می‌کنند، تماما نیاز به جداول نگاشت بسیار بزرگی برای ترجمه آدرس دارند [۱۶-۱۵].

هدف ما از این طرح پیشنهادی ، ارائه یک FTL جدید است که با استفاده از تکنیک فشرده‌سازی داده‌ها، طول عمر حافظه‌های فلش را بهبود داده و در عین حال یک واحد ترجمه آدرس جدید را معرفی می‌کند که جدول نگاشت آدرس بسیار کوچکتری در مقایسه با طرح‌های پیشین دارد و در فضای حافظه نگاشت صرفه‌جویی می-کند. و همچنین این طرح سرعت خواندن و نوشتن بهتری نسبت به طرح‌های پیشین دارد.

ما در بخش ۲ این مقاله کارهای پیشین مرتبط با ترجمه آدرس و فشرده‌سازی را بررسی کرده و سپس در بخش ۳، به معرفی طرح پیشنهادی خود به نام معماری نگاشت آدرس سکتور با بروزرسانی درون بلوکی می‌پردازیم. آنگاه در بخش ۴ نتایج شبیه‌سازی طرح پیشنهادی نسبت به طرح‌های گذشته بررسی شده و در پایان نتیجه‌گیری کلی بیان می‌شود.

۲- پیشینه تحقیق

عملیات نگاشت در واحد ترجمه آدرس را می‌توان به سه دسته‌ی نگاشت صفحه‌ای، نگاشت بلوکی و نگاشت ترکیبی تقسیم‌بندی کرد. در نگاشت صفحه‌ای، ورودی جدول نگاشت، یک LPN (شماره صفحه منطقی) بوده و محتوای هر مدخل یک PPN (شماره صفحه فیزیکی) است. عیب بزرگ این طرح این است که جدول نگاشت در آن بسیار بزرگ است. در نگاشت بلوکی، آدرس صفحه منطقی به یک LBN (شماره بلوک منطقی) و یک آفست صفحه تقسیم می-شود. از LBN برای پیدا کردن یک بلوک فیزیکی و از آفست صفحه برای نوشتن داده‌ها درون یک بلوک فیزیکی استفاده می‌شود. این طرح کوچکترین جدول نگاشت را دارد ولی عیب بزرگ آن بازنویسی مکرر تمام صفحات درون یک بلوک فیزیکی است [۸-۷].

درخواست‌های نوشتنی که در یک بازه زمانی برای حافظه فلش ارسال می‌شوند، شامل مجموعه‌ای از درخواست‌های نوشتن متوالی و تصادفی است. یک درخواست نوشتن متوالی، یک درخواست با اندازه بزرگ بوده که بعد از آخرین سکتور آن، درخواست‌های نوشتن دیگری به ترتیب وارد دستگاه ذخیره‌ساز فلش می‌شوند. اما یک درخواست نوشتن تصادفی، معمولا اندازه کوچکی داشته و بخش کوچکی از یک بلوک منطقی را شامل می‌شود. طرح‌های ترجمه آدرس ترکیبی، معمولا درخواست‌های نوشتن متوالی را به صورت نگاشت بلوکی و درخواست‌های نوشتن تصادفی را به‌صورت نگاشت صفحه‌ای ترجمه آدرس می‌کنند [۱۰]. از مشهورترین طرح‌های نگاشت آدرس ترکیبی می‌توان به طرح‌های FAST و LAST اشاره کرد [۱۱-۱۰].

از آنجایی‌که عیب بزرگ ترجمه آدرس صفحه‌ای، اندازه بزرگ جدول نگاشت است، در مقالاتی همچون DFTL و CDFTL پیشنهاد شده است که عملیات ترجمه آدرس مبتنی بر تقاضا باشد و جدول نگاشت تماما در صفحات فیزیکی حافظه فلش ذخیره شوند. سپس به هنگام نگاشت آدرس یک سکتور منطقی به یک صفحه فیزیکی، اطلاعات نگاشت آدرس متعلق به سکتور از صفحات فیزیکی فلش (موسوم به صفحات ترجمه) خوانده شده و درون حافظه نگاشت FTL بارگزاری می‌شود. چنین طرح‌هایی به حافظه نگاشت بسیار کمی نیاز دارند ولی مشکل عمده آن‌ها نیاز به خواندن و همچنین دوباره‌نویسی صفحات ترجمه در هنگام خواندن و نوشتن درخواست‌ها است و این باعث ایجاد سربار اضافی بر روی زمان تاخیر خواندن و نوشتن درخواست‌ها درون حافظه فلش می‌شود [۱۳-۱۲].

یک راهکار دیگر برای کاهش حافظه نگاشت آدرس، معرفی طرح CIAM بوده است. این طرح یک روش نگاشت آدرس مستقل از ظرفیت است و یک راهکار برای جداسازی فضای موردنیاز برای نگاشت آدرس‌ها از ظرفیت یک حافظه فلش، ارائه می‌کند. منطق طراحی این است که میزان فضای RAM مورد نیاز برای یک حافظه فلش بجای اینکه به ظرفیت حافظه فلش اشاره کند، باید به مجموعه داده‌های دردسترس توسط کاربر وابسته باشد. در این طرح، فضای RAM موردنیاز برای نگاشت آدرس نسبتا کوچک است و می تواند تقریبا مستقل از ظرفیت ذخیره‌سازی و مقدار داده‌های ذخیره شده باشد [۱۴] . عیب بزرگ این طرح استفاده از یک درخت جستجوی باینری نسبتا بزرگ برای ترجمه آدرس و عدم کارایی طرح در نوشتن موازی درخواست‌های ورودی به سیستم است. این عوامل سرعت خواندن و نوشتن داده‌ها را تا حدودی پایین می‌آورد.

در برخی از کارهای گذشته استفاده از ایده فشرده‌سازی داده‌ها در حافظه‌های فلش پیشنهاد شده است. از آنجایی‌که با فشرده‌سازی داده‌ها، بلوک‌های فیزیکی کمتری برای نوشتن درخواست‌ها مصرف می‌شوند و حجم بیشتری از بلوک‌های حافظه آزاد می‌ماند، در نتیجه نیاز کمتری به عملیات زباله‌روبی است و بدین ترتیب طول عمر حافظه فلش افزایش می‌یابد. مهمترین طرح‌هایی که از ایده فشرده-سازی داده‌ها در سطح حافظه فلش پشتیبانی می‌کنند، طرح‌های همچون ZFTL و LDC هستند [۱۶-۱۵].

طرح ZFTL [۱۵] [۱۷]، از نگاشت آدرس صفحه‌ای برای ترجمه آدرس استفاده می‌کند. در این طرح چندین صفحه منطقی قابل فشرده‌سازی درون یک بافر نوشتن قرار می‌گیرند. وقتی بافر نوشتن فضای کافی برای داده فشرده‌شده جدید نداشته باشد، آنگاه درون یک صفحه فیزیکی تخلیه می‌شود. سپس در جدول نگاشت صفحه، آدرس‌های صفحات منطقی فشرده شده موجود در بافر نوشتن، تماما به یک صفحه فیزیکی نگاشت می‌شوند. در این طرح برای فشرده-سازی داده‌ها، الگوریتم‌های فشرده‌سازی LZ77 [۱۸] و یا Zlib [۱۹] پیشنهاد شده است که پیاده‌سازی سخت‌افزاری کارایی برای آن‌ها وجود دارد. واحد فشرده‌سازی داده‌ها در این طرح ثابت و برابر با اندازه یک صفحه منطقی (4KB) است. از آنجایی که در طرح ZFTL چندین صفحه منطقی درون یک صفحه فیزیکی نوشته می‌شود، در نتیجه برای انجام عملیات زباله‌روبی نیاز به یک جدول نگاشت حالت به نام PST است. این جدول به ازای هر صفحه فیزیکی یک مدخل دارد و اندازه آن، تقریبا بزرگ بوده و حدود ۲۵ درصد جدول نگاشت صفحه (PMT) است. در هنگام عملیات زباله‌روبی، به کمک جدول PST صفحات منطقی معتبر درون هر صفحه فیزیکی شناسایی شده و سپس این صفحات منطقی معتبر پس از تجمیع درون بافر نوشتن در صفحات خالی یک بلوک فیزیکی جدید نوشته می‌شوند. یکی از معایب طرح ZFTL، اندازه بزرگ جدول نگاشت آدرس است که در مجموع حافظه نگاشت مورد نیاز در آن بسیار بیشتر از طرح‌های نگاشت صفحه‌ای است.

طرح LDC [۱۶] در حقیقت نسخه بهبود یافته طرح ZFTL است. این طرح برای پیش‌بینی قابلیت فشرده‌سازی داده‌ها از یک مولفه نرم‌افزاری به جای مولفه سخت‌افزاری (پیشنهاد شده در طرح ZFTL) بهره می‌برد. در طرح LDC از یک روش ابتکاری برای پیش-بینی نرخ فشرده‌سازی داده‌ها استفاده می‌شود که ارتباط میان آنتروپی و نرخ فشرده‌سازی داده‌ها را نشان می‌دهد. دومین بهبود ایجاد شده در این طرح مربوط به عملیات زباله‌روبی است که یک الگوریتم جدید برای زباله‌روبی پیشنهاد می‌دهد. ایده اصلی این زباله‌روبی در این طرح بر این مبنا است که بسیاری از داده‌های غیر قابل فشرده‌سازی، به یک میزان خاصی قابل فشرده‌سازی هستند و

با فشرده‌سازی این داده‌ها به هنگام عملیات زباله‌روبی می‌توان در تعداد صفحات فیزیکی که باید نوشته شوند صرفه‌جویی کرد. سربار عملیات زباله‌روبی در این طرح نسبت به طرح ZFTL بیشتر است ولی به دلیل آنکه این طرح حجم زیادی از داده‌های غیرقابل فشرده‌سازی را بهتر از طرح ZFTL شناسایی می‌کند، در مجموع کارایی بهتری در میانگین زمان خواندن و نوشتن نسبت به طرح ZFTL دارد. این طرح با وجود اینکه بهبود قابل قبولی در زمان پاسخگویی به درخواست‌ها نسبت به طرح ZFTL دارد، ولی مشکل اصلی حجم زیاد حافظه نگاشت در آن هنوز پابرجا است.

از طرح دیگری که در رابطه با فشرده‌سازی داده‌ها در سطح حافظه‌های فلش مطرح شده است، می‌توان EDC [۲۰] را نام برد. در این طرح سعی شده است که با توجه به سرعت و نرخ ورود داده‌ها به حافظه فلش، از سه نوع الگوریتم فشرده‌سازی متفاوت استفاده شود. اگر نرخ ورود داده‌ها بسیار زیاد باشد، از یک الگوریتم فشرده‌سازی استفاده می‌شود که سرعت بالایی در فشرده‌سازی دارد ولی قدرت فشرده‌سازی آن ضعیف است (همانند الگوریتم Lz4 و Lzf). اما اگر نرخ ورود داده‌ها کم باشد، از الگوریتمی برای فشرده‌سازی استفاده می‌شود که سرعت کمی دارد ولی قدرت فشرده‌سازی بالایی دارد (همانند الگوریتم‌های Bzip2 و Gzip). در این طرح برای حد وسط از الگوریتم فشرده‌ساز Zlib استفاده شده است. برخی از پژوهش‌های مرتبط پیشین، متمرکز بر ارزیابی و معرفی الگوریتم‌های فشرده‌سازی بهینه و کارا برای استفاده در حافظه‌های فلش بوده است. به عنوان مثال در طرح ZFTL کاربرد الگوریتم‌های فشرده‌سازی Zlib و LZ77 در فشرده‌سازی داده‌ها مورد ارزیابی قرار گرفته است. در طرح دیگری تحت عنوان BlueZIP [۲۱] اقدام به طراحی یک ماژول سخت افزاری برای فشرده‌سازی داده‌ها برای SSDهای مبتنی بر حافظه فلش شده است. همچنین در طرح دیگری تحت عنوان Modified LZ4 [۲۲] یک الگوریتم فشرده‌سازی داده پیشنهاد شده است که در SSDهای مبتنی بر حافظه فلش کاربرد دارد. از آنجایی که پیاده‌سازی سخت‌افزاری الگوریتم LZ4 بسیار سخت و پیچیده است، در نتیجه در این طرح از یک الگوریتم اصلاح شده برای پیاده‌سازی سخت‌افزاری آن استفاده شده است.

۲-۱- انگیزه

دستگاه‌های ذخیره‌ساز داده معمولا جریان داده‌ها را از چندین منبع داده مختلف و به صورت درهم آمیخته دریافت می‌کنند. در میان درخواست‌های نوشتن، برخی ماهیت تصادفی و برخی دیگر ماهیت متوالی دارند. نوشتن‌های تصادفی سربار زیادی بر حافظه‌های فلش تحمیل می‌کنند و باعث انتقال صفحات زیادی در عملیات زباله‌روبی می‌شوند. FTLهای سطح صفحه در کاربردهایی که سطح دسترسی آن‌ها بیشتر از نوع نوشتن تصادفی بوده و شامل بروزرسانی‌های زیاد است توصیه می‌شوند. ولی عیب بزرگ آن‌ها اندازه بسیار بزرگ جدول نگاشت آدرس است. طرح‌هایی همچون DFL اگرچه حافظه نگاشت کوچکی دارند ولی عیب عمده آن، خواندن و بازنویسی مکرر صفحات ترجمه آدرس در حافظه فلش است که سربار بسیار زیادی به سیستم تحمیل می‌کند. ارزیابی‌ها نشان می‌دهد که تاخیر خواندن و نوشتن درخواست‌ها در DFTL کمتر از طرح‌هایی است که جداول نگاشت را تماما در حافظه موقت نگهداری می‌کنند.

در بسیاری از کاربردها (همانند کامپیوترهای شخصی) تعداد درخواست‌های تصادفی ارسالی برای حافظه فلش چندان زیاد نبوده و بیشتر درخواست‌های آن‌ها از نوع متوالی است. برای چنین کاربردهایی، FTLهای ترکیبی مناسب‌تر هستند. همچنین در کامپیوترهای شخصی بسیاری از درخواست‌های نوشتن متعلق به فایل‌های مالتی‌مدیا یا فایل‌های فشرده هستند که در آن‌ها تعداد کمی الگوی تصادفی وجود دارد [۱۱]. ویژگی مهم این فایل‌ها، این است که داده‌های آن‌ها غیرقابل فشرده‌سازی هستند [۱۵].

عملیات ترجمه آدرس پیشنهاد شده در FTLهای مبتنی بر فشرده‌سازی قبلی، تماما به صورت نگاشت صفحه‌ای است که نیازمند جدول نگاشت آدرس بسیار بزرگ است. یک راهکار برای کم کردن حافظه نگاشت در چنین FTLهایی استفاده از ترجمه آدرس ترکیبی است. این FTL ترکیبی باید بگونه‌ای باشد که هم دارای حافظه نگاشت کمی بوده و هم دارای تاخیر خواندن و نوشتنی در حد طرح‌های پیشنهادی قبلی باشد.

طرح پیشنهادی ما در حقیقت یک FTL با نگاشت آدرس ترکیبی مبتنی بر لاگ بلوک بوده که از قابلیت فشرده‌سازی داده‌ها پشتیبانی می‌کند و در SSDهای مورد استفاده در کامپیوترهای شخصی کاربرد دارد. در این طرح صفحات درون هر بلوک منطقی با استفاده از روش فشرده‌سازی داده‌ها، فشرده شده و در مجموع صفحات فیزیکی کمتری در هر بلوک فیزیکی نوشته می‌شوند. بنابراین در بلوک‌های فیزیکی حافظه فلش تعدادی صفحه آزاد باقی می‌ماند که از این صفحات آزاد، برای درخواست‌های بروزرسانی داده‌ها استفاده می‌شود. از آنجایی‌که در این طرح پیشنهادی، امکان بروزرسانی داده‌ها درون هر بلوک فیزیکی وجود دارد، در نتیجه به آن In Block Update FTL یا به اختصار IBU FTL گفته می‌شود.

۳- طرح پیشنهادی

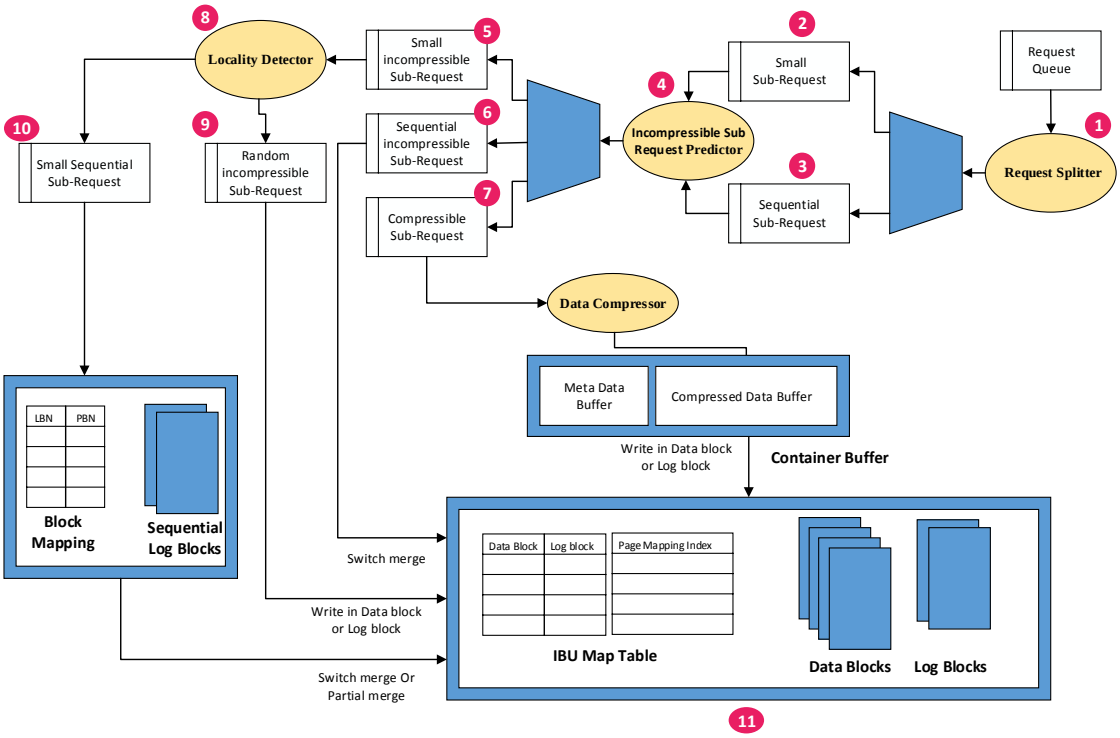
در این بخش، طرح پیشنهادی تشریح می‌گردد.

۳-۱- معماری کلی طرح

طرح پیشنهادی IBU FTL بر اساس ایده فشرده‌سازی داده‌ها در لایه FTL پایه‌گذاری شده است. ساختار کلی این طرح در شکل ۱ نشان داده شده است. از آنجایی‌که بسیاری از فایل‌های نوشته شده در حافظه‌های فلش، غیرقابل فشرده‌سازی هستند و این فایل‌ها معمولا اندازه بزرگی دارند، بنابراین درخواست‌های نوشتن آن‌ها بیشتر بصورت درخواست‌های متوالی تولید می‌شوند. یکی از ویژگی-های اصلی این طرح، نگاشت بلوکی درخواست‌های نوشتن متوالی غیرقابل فشرده‌سازی است. با توجه به شکل ۱ برای سرویس‌دهی به درخواست‌های نوشتن، ابتدا این درخواست‌ها با استفاده از مولفه Request Splitter (مرحله ۱) به تعدادی زیردرخواست متوالی و زیردرخواست کوچک (مرحله ۲ و ۳) دسته‌بندی می‌شوند. هر کدام از این زیردرخواست‌ها معادل با داده‌های یک بلوک منطقی هستند. طبق تعریف یک زیردرخواست متوالی شامل تمام صفحات یک بلوک منطقی است، در حالی‌که یک زیردرخواست کوچک فقط شامل بخش کوچکی از یک بلوک منطقی می‌شود.

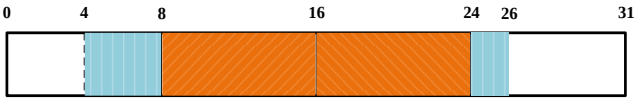
در این طرح قابلیت فشرده‌سازی زیردرخواست‌های نوشتن با استفاده از مولفه‌ای به نام Incompressible Request predictor پیش‌بینی شده (مرحله ۴) و به سه دسته زیردرخواست‌های قابل‌فشرده‌سازی و زیردرخواست‌های کوچک غیرقابل فشرده‌سازی و زیردرخواست-های متوالی غیرقابل فشرده‌سازی (مرحله ۵ و ۶ و ۷) تقسیم می-شوند. هر کدام از زیردرخواست‌های قابل فشرده‌سازی معادل با یک بلوک منطقی بوده و با فشرده‌سازی صفحات هر بلوک منطقی، چندین صفحه منطقی درون یکی از صفحات بلوک فیزیکی نوشته می‌شوند (مرحله ۱۱).

این طرح برای ترجمه آدرس از یک جدول نگاشت آدرس دو سطحی به نام IBU Map Table استفاده می‌کند و در آن هر بلوک منطقی به یک بلوک فیزیکی و یک لاگ بلوک اشتراکی (متعلق به ناحیه over provision) نگاشت می‌شود. پس از سرویس‌دهی به زیردرخواست قابل فشرده‌سازی معمولا تعدادی صفحه آزاد درون هر بلوک فیزیکی باقی می‌ماند. این صفحات آزاد برای سرویس‌دهی به درخواست‌های بروزرسانی رزرو می‌شوند و بدین ترتیب امکان بازنویسی زیردرخواست‌ها درون هر بلوک فیزیکی فراهم می‌شود. اگر زیردرخواستی از نوع متوالی غیرقابل فشرده‌سازی باشد، آنگاه تماما در یک بلوک فیزیکی جدید نوشته شده و آدرس‌های آن در IBU Map Table بروزرسانی می‌شود. مزیت زیردرخواست‌های متوالی غیرقابل‌فشرده‌سازی در این است که هزینه زباله‌روبی را به میزان بسیار زیادی کاهش می‌دهد.



شکل ۱: معماری کلی طرح IBU FTL

چنانچه تعدادی زیردرخواست کوچک غیرقابل فشرده‌سازی به‌گونه‌ای در امتداد همدیگر قرار گیرند که بتوانند یک بلوک منطقی را تماما پر کنند، آنگاه به هر کدام از آن‌ها یک زیردرخواست کوچک متوالی (زیردرخواست غیرقابل فشرده‌سازی) گفته می‌شود. این نوع از زیردرخواست‌ها، به کمک مولفه‌ای به نام Locality Detector تشخیص داده شده (مرحله ۸) و سپس با استفاده از یک جدول نگاشت بلوکی درون تعدادی لاگ بلوک به نام Sequential Log block تجمیع می‌شوند. پس از آنکه هر لاگ بلوک متوالی به طور کامل نوشته شد، آنگاه آدرس‌های نگاشت مرتبط با آن به IBU Map Table منتقل می‌شود (مرحله ۱۰). تجمیع این نوع از زیردرخواست‌ها درون لاگ بلوک‌های متوالی، هزینه عملیات زباله‌روبی را کاهش می‌دهد. اما اگر یک زیردرخواست کوچک غیرقابل فشرده‌سازی توسط مولفه Locality Detector به عنوان یک زیردرخواست متوالی تشخیص داده نشود، آنگاه به آن یک زیردرخواست تصادفی غیرقابل فشرده‌سازی گفته می‌شود و توسط IBU Map Table درون بلوک داده یا لاگ بلوک اشتراکی نوشته می‌شود (مرحله ۹). بدین ترتیب با تفکیک درخواست‌های نوشتن به لحاظ فاکتورهای نوع درخواست (متوالی یا تصادفی) و قابلیت فشرده‌سازی آن‌ها، امکان استفاده از یک مولفه ترجمه آدرس ترکیبی در طرح پیشنهادی فراهم می‌شود که حافظه نگاشت آن حدود یک چهارم طرح‌های پیشین است و به کارایی زمان پاسخ نوشتن درخواست‌ها، بهتر از طرح‌های پیشین عمل می‌کند. مولفه تقسیم کننده درخواست درخواست‌های نوشتنی که از سمت سیستم فایل برای یک دستگاه ذخیره‌ساز فلش ارسال می‌شود، با یک آدرس شروع و یک اندازه مشخص می‌شود. مولفه تقسیم کننده با تحلیل آدرس شروع و پایان و اندازه درخواست، آن را به تعدادی زیردرخواست متوالی و کوچک تقسیم می‌کند. هر زیردرخواست متعلق به یک بلوک منطقی است. در طرح پیشنهادی آن دسته از زیردرخواست‌های متوالی و زیردرخواست‌های کوچک با خاصیت متوالی که قابلیت فشرده‌سازی به اندازه کافی را ندارند، به صورت نگاشت بلوکی سرویس داده شده و از سربار هزینه زباله‌روبی کاسته می‌شود. در شکل شماره ۲، به عنوان نمونه یک درخواست با آدرس شروع ۴ و به اندازه ۲۲ صفحه منطقی وارد دستگاه ذخیره‌ساز فلش می‌شود که در آن اندازه هر بلوک منطقی برابر با ۸ صفحه است. مولفه تقسیم کننده این درخواست را به دو زیر درخواست کوچک و دو زیردرخواست متوالی تقسیم می‌کند. هر کدام از این زیردرخواست‌ها به ترتیب متعلق به بلوک‌های منطقی LBN0, LBN1, LBN2, LBN3 هستند.



شکل ۲: تقسیم یک درخواست به چندین زیردرخواست

۳-۲- مولفه پیش‌بینی درخواست‌های غیرقابل فشرده و الگوریتم فشرده‌سازی داده‌ها

در طرح پیشنهادی از مولفه Incompressible Request predictor برای کشف درخواست‌های غیرقابل فشرده استفاده می‌شود. بررسی‌ها نشان می‌دهد که نرخ فشرده‌سازی اکثر صفحات منطقی درون به یک درخواست نزدیک به هم بوده و پیش‌بینی نرخ فشرده‌سازی چند صفحه منطقی درون یک درخواست قابل تعمیم به بقیه صفحات درخواست نیز است. در این طرح برای پیش‌بینی، چند نمونه تصادفی از داده‌های موجود در هر زیردرخواست (به اندازه یک صفحه منطقی) را فشرده کرده و چنانچه میانگین نرخ فشرده‌سازی آن‌ها کمتر از ۵۰ درصد باشد، آنگاه احتمال داده می‌شود که آن درخواست غیرقابل‌فشرده‌سازی است.

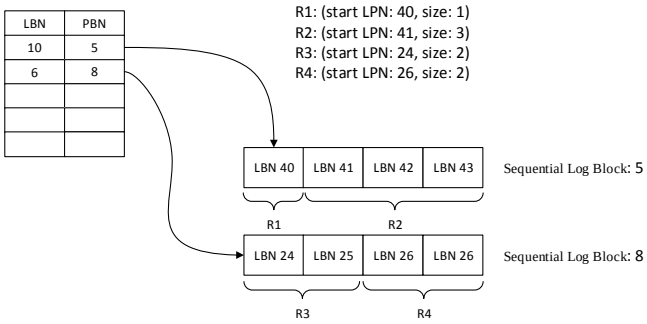
یکی از مهمترین فاکتورها در انتخاب الگوریتم فشرده‌سازی نرخ فشرده‌سازی و پیچیدگی پیاده‌سازی سخت افزاری یا نرم‌افزاری آن است. در طرح پیشنهادی از الگوریتم LZ77 که یکی از شناخته‌ترین الگوریتم‌های فشرده‌سازی است و پیاده‌سازی سخت افزاری و نرم‌افزاری بهینه‌ای برای آن وجود دارد، استفاده می‌شود. واحد فشرده‌سازی داده‌ها یک فاکتور مهم در نرخ فشرده‌سازی و سرعت آن است. اگرچه برای دستیابی به نرخ فشرده‌سازی بهتر، استفاده از واحد فشرده‌سازی بزرگ‌تر مناسب‌تر است، اما بررسی‌ها نشان می‌دهد که اختلاف معناداری میان نرخ فشرده‌سازی میان داده‌های 2KB تا 8KB وجود ندارد. در طرح پیشنهادی از یک واحد فشرده‌سازی با اندازه ثابت ۴ کیلوبایت (به اندازه سایز یک صفحه منطقی داده) استفاده می‌شود.

۳-۳- مولفه *Locality Detector* و نوشتن داده‌های غیرقابل فشرده‌سازی

در طرح پیشنهادی زیردرخواست‌های غیرقابل فشرده‌سازی شامل دو گروه زیردرخواست متوالی غیرقابل فشرده‌سازی و زیردرخواست کوچک غیرقابل فشرده هستند. زیردرخواست‌های متوالی غیرقابل فشرده‌سازی به طور کامل درون یک بلوک فیزیکی نوشته شده و سپس آدرس بلوک فیزیکی به IBU Map Table منتقل می‌شود.

از میان زیردرخواست‌های غیرقابل فشرده‌سازی کوچک، معمولا تعدادی زیردرخواست کوچک وجود دارند که در امتداد همدیگر قرار می‌گیرند. به هر کدام از این زیردرخواست‌ها یک زیردرخواست

کوچک متوالی گفته می‌شود و به کمک مولفه Locality Detector تشخیص داده می‌شوند. این مولفه با استفاده از یک الگوریتم تشخیص محلیت مکانی و به کمک یک جدول نگاشت آدرس بلوکی، زیردرخواست‌های کوچک غیرقابل فشرده‌سازی را درون تعدادی لاگ بلوک به نام Sequential Log block تجمیع می‌کند. پس از پر شدن هر کدام از این لاگ بلوک‌های متوالی، آدرس لاگ بلوک فیزیکی را با یک عملیات ادغام جابجایی به جدول نگاشت آدرس IBU Mapping منتقل می‌کند. ساختار جدول نگاشت آدرس Block Map Table در شکل ۳ نشان داده شده است. این جدول از دو فیلد LBN و PBN تشکیل شده است. در این جدول از فیلد LBN به عنوان کلید جستجو استفاده می‌شود و فیلد PBN اشاره به یک Sequential Log Block می‌کند که حاوی زیردرخواست‌های کوچک متوالی مرتبط با یک بلوک منطقی (LBN) هستند.



شکل ۳: مثالی از شیوه کار مولفه Locality Detector

الگوریتم تشخیص محلیت مکانی هر زیردرخواست کوچک بسیار ساده بوده و به صورت زیر است:

۱) اگر آدرس شروع یک زیردرخواست کوچک از ابتدای یک بلوک منطقی باشد، آنگاه این زیردرخواست یک زیردرخواست کوچک متوالی در نظر گرفته شده و آدرس بلوک منطقی آن به جدول Block Map Table افزوده شده و سپس زیردرخواست به ترتیب درون یک Sequential Log block نوشته می‌شود.

۲) اگر آدرس بلوک منطقی یک زیردرخواست کوچک از قبل به عنوان کلید جستجو در جدول Block Map Table موجود باشد و بتوان این زیردرخواست را در امتداد داده-های موجود در Sequential Log block مربوطه نوشت، آنگاه این زیردرخواست یک زیردرخواست کوچک متوالی خواهد بود.

۳) در غیر اینصورت زیردرخواست کوچک، به عنوان یک زیردرخواست تصادفی در نظر گرفته شده و مستقیما به کمک IBU Map Table درون بلوک‌های داده یا لاگ بلوک‌های اشتراکی نوشته می‌شود.

۳-۴- نوشتن داده‌ها در بلوک‌های داده و لاگ بلوک‌های اشتراکی

در طرح پیشنهادی برای زیردرخواست‌های قابل فشرده‌سازی، یک مولفه فشرده‌ساز داده وجود دارد که داده‌های صفحات منطقی زیردرخواست‌ها را فشرده‌سازی کرده و در در یک بافر موقت به نام Container Buffer تخلیه می‌کند. همانگونه که در شکل ۳ نشان داده شده است، ظرفیت این بافر به اندازه یک صفحه فیزیکی بوده و شامل دو بخش Compressed Data Buffer و Meta Data Buffer است. بخش Compressed Data Buffer محل نگهداری صفحات منطقی فشرده‌سازی شده بوده و بخش Meta Data Buffer برای نگهداری ابرداده‌های مورد نیاز برای عملیات‌های ترجمه آدرس و زباله‌روبی است. هرگاه ظرفیت Container Buffer به اندازه‌ای پر شود که صفحه منطقی فشرده شده بعدی نتواند درون آن قرار گیرد، آنگاه محتوای این بافر درون آخرین صفحه خالی یک بلوک فیزیکی نوشته می‌شود. سپس Container Buffer ریست شده و برای جایگذاری صفحات منطقی فشرده شده بعدی می‌شود.

در FTLهای مبتنی بر فشرده‌سازی، در هنگام عملیات زباله‌روبی و همچنین خواندن داده‌ها نیاز به تفکیک صفحات منطقی و بررسی اعتبار صفحات منطقی فشرده‌شده درون هر صفحه فیزیکی است. برای اینکار نیاز به ذخیره‌سازی یکسری اطلاعات اضافه است که به آن ابرداده گفته می‌شود. به هنگام فشرده‌سازی داده‌ها درون یک صفحه فیزیکی همیشه مقداری فضای فرگمنت باقی می‌ماند که از این فضا می‌توان برای نگهداری ابرداده‌های مورد نیاز استفاده کرد [۴] [۲۳]. پیشنهاد ما در طرح IBU FTL این است که از این فضای فرگمنت برای ذخیره‌سازی اینگونه ابرداده‌ها استفاده شود. این ابرداده‌ها شامل یک فیلد Number (یک بایت) که تعداد صفحات منطقی فشرده شده را نشان می‌دهد و آفست صفحات فشرده شده (برای هر صفحه یک بایت) و اندازه صفحات فشرده شده (برای هر صفحه دو بایت) است. به کمک این ابرداده‌ها به آسانی می‌توان صفحات منطقی فشرده‌شده معتبر را از یکدیگر تفکیک کرد. بخش Meta Data Buffer در بافر نگهدارنده برای ذخیره‌سازی این ابرداده-ها کاربرد دارد. جدول ترجمه آدرس طرح پیشنهادی، در شکل ۴ نشان داده شده است. این جدول دارای دو قسمت مجزا برای ترجمه آدرس است. بخش اول این جدول، آدرس‌های منطقی را به روش نگاشت بلوکی ترجمه کرده و شامل دو فیلد Data PBN و Log PBN به ازای هر بلوک منطقی است. فیلد Data PBN در حقیقت یک بلوک فیزیکی است که متعلق به فضای قابل دسترس حافظه

فلش است و تمام صفحات آن تنها به یک بلوک منطقی اختصاص می‌یابد.

بخش دوم جدول نگاشت آدرس، صفحات یک بلوک منطقی را به صورت نگاشت صفحه‌ای درون یک بلوک فیزیکی ترجمه آدرس می‌کند و شامل فیلدهایی برای نگهداری آفست فیزیکی داده نوشته شده درون یک بلوک فیزیکی است. تعداد این فیلدهای آفست به اندازه تعداد صفحات یک بلوک فیزیکی می‌باشد. به عنوان مثال اگر ۶۴ صفحه درون یک بلوک فیزیکی وجود داشته باشد، آنگاه ۶۴ فیلد آفست برای بخش دوم جدول نگاشت آدرس درنظر گرفته می‌شود.

در کنار هر کدام از فیلدهای آفست یک فیلد تک بیتی برچسب به نام Log قرار گرفته است و فعال بودن آن مشخص می‌کند که آفست فیزیکی نوشته شده متعلق به یک صفحه فیزیکی درون لاگ بلوک اشتراکی است. از آنجایی که چندین صفحه منطقی با فشرده‌سازی می‌توانند درون یک صفحه فیزیکی قرار گیرند، در نتیجه در هر سطر جدول نگاشت آدرس می‌توان چندین آفست صفحه فیزیکی یکسان برای تعدادی صفحه منطقی داشت.

برای ترجمه آدرس یک صفحه منطقی به یک صفحه فیزیکی در IBU Map Table بدین صورت عمل می‌شود که ابتدا شماره بلوک منطقی (LBN) با تقسیم شماره صفحه منطقی بر تعداد صفحات هر بلوک بدست آمده و سپس بلوک فیزیکی (بلوک داده یا لاگ بلوک) نگاشت شده به LBN مورد نظر از بخش ترجمه آدرس بلوکی جدول نگاشت استخراج می‌گردد. سپس آفست صفحه درون بلوک منطقی از طریق باقیمانده تقسیم شماره صفحه منطقی بر تعداد صفحات هر بلوک بدست آمده و با استفاده از این آفست و به کمک بخش ترجمه آدرس صفحه ای جدول نگاشت، آفست صفحه فیزیکی نگاشت شده به صفحه منطقی بدست می‌آید. آنگاه شماره صفحه فیزیکی نگاشت شده به صفحه منطقی به کمک شماره بلوک فیزیکی و آفست فیزیکی بدست آمده، محاسبه می‌گردد.

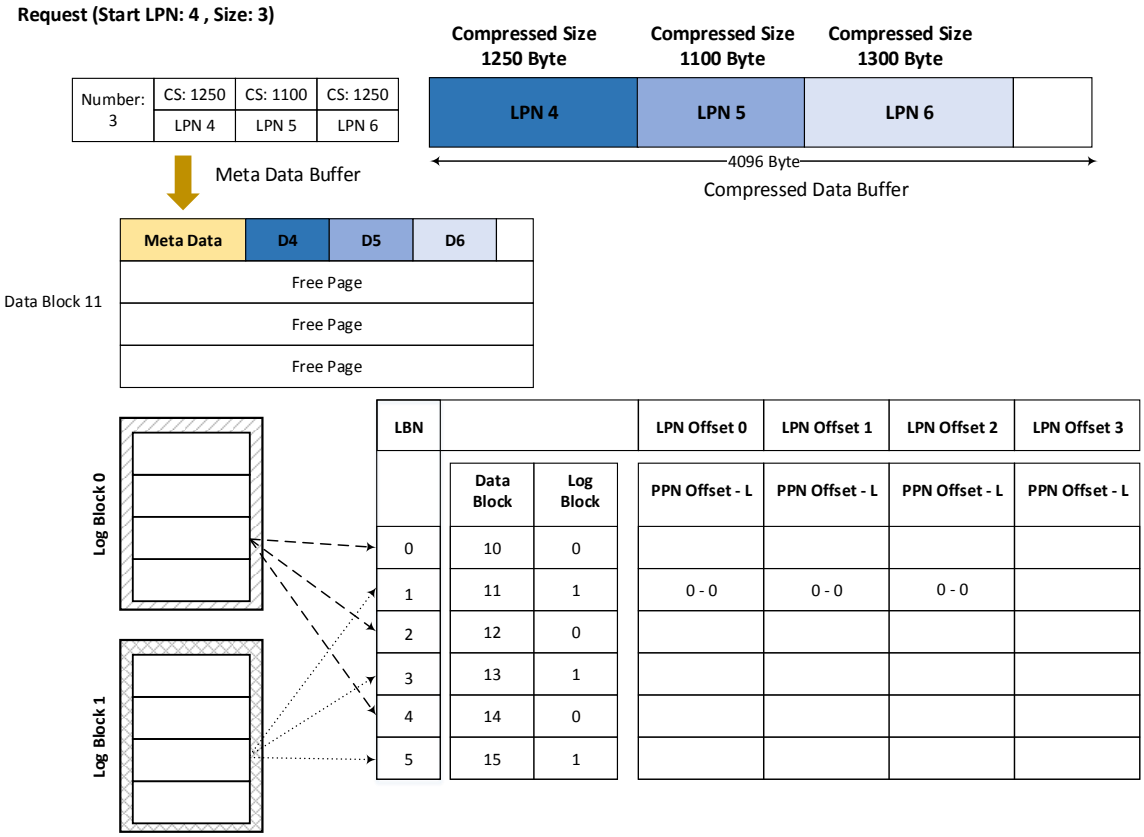
$$LBN = (LPN / Page\ per\ Block)$$

$$PBN = (Data\ Block\ or\ Log\ Block\ mapped\ to\ LBN)$$

$$LPN\ Offset = (LPN\ \%\ Page\ per\ Block)$$

$$PPN\ Offset = (PPN\ Offset\ mapped\ to\ LPN\ Offset)$$

$$PPN = (PBN \times Page\ per\ Block) + PPN\ Offset$$



شکل ۴: Container Buffer و Map Table در طرح IBU FTL

۳-۵- الگوریتم نوشتن داده‌ها

شبه‌کد شکل ۵ الگوریتم نوشتن درخواست‌های ورودی به دستگاه ذخیره‌ساز فلش را نشان می‌دهد. همانگونه که بیان شد، تمام درخواست‌های ورودی به سیستم ابتدا توسط مولفه تقسیم‌کننده درخواست به تعدادی زیردرخواست تقسیم می‌شوند. هر کدام از این زیردرخواست‌ها مرتبط با یک بلوک منطقی هستند. برای نوشتن هر زیردرخواست ابتدا قابلیت فشرده‌سازی آن بررسی می‌شود. اگر زیردرخواست قابلیت فشرده‌سازی داشته باشد، آنگاه نوع زیردرخواست از لحاظ متوالی بودن بررسی می‌گردد. اگر زیردرخواست از نوع متوالی قابل فشرده‌سازی باشد ولی بلوک داده نگاشت شده به زیردرخواست فضای کافی برای نوشتن صفحات فشرده شده زیردرخواست را نداشته باشد، در نتیجه یک بلوک داده جدید برای نوشتن صفحات فشرده شده زیردرخواست اختصاص می‌یابد و سپس با یک عملیات ادغام جابجایی و بدون سربار زمانی آدرس‌های بلوک داده جدید به جای بلوک داده قدیمی تعویض می‌شود.

به عنوان مثال در شکل ۶ زیردرخواست 0 مرتبط با LBN 0 بوده و از نوع یک زیردرخواست متوالی قابل فشرده‌سازی است. با توجه به جدول IBU Map Table بلوک داده نگاشت شده به این LBN (Data Block 10) فضای کافی برای نوشتن این زیردرخواست را ندارد. در نتیجه یک بلوک داده جدید به این زیردرخواست اختصاص داده شده (Data Block 16) و سپس صفحات منطقی این زیردرخواست فشرده شده و در صفحات فیزیکی با آفست ۰ و ۱ در بلوک فیزیکی جدید نوشته می‌شوند. در نهایت با یک عملیات ادغام جابجایی جدول IBU Map Table بروزرسانی می‌شود.

ولی اگر زیردرخواست از نوع متوالی قابل فشرده‌سازی باشد و بلوک داده نگاشت شده به زیردرخواست فضای کافی برای نوشتن صفحات فشرده شده را داشته باشد، آنگاه صفحات منطقی زیردرخواست، فشرده شده و به ترتیب درون آخرین صفحات بلوک داده نوشته می‌شوند.

به عنوان مثال در شکل ۶ زیردرخواست ۷ مرتبط با LBN 5 بوده و از نوع یک زیردرخواست متوالی قابل فشرده‌سازی است. بلوک داده نگاشت شده به این LBN (Data Block 15) دارای فضای کافی برای نوشتن زیردرخواست است. در نتیجه صفحات منطقی این زیردرخواست، فشرده شده و به ترتیب در صفحات فیزیکی با افست ۱ و ۲ در بلوک فیزیکی ۱۵ نوشته می‌شوند.

```
subReqList = divide the Request into several Sub-Requests
foreach (subReq in subReqList){
  if(subReq is compressible){
    if(typeof(subReq) == SEQUENTIAL_TYPE){
      if(data block have enough space to write the subReq){
        1. write multiple compressed logical pages
           in the Container Buffer
        2. write Container Buffer in the last Data Block
           or Log Block page;
        3. map the logical pages by IBU Map Table;
      }
      else{
        1. write multiple compressed logical pages
           in the Container Buffer
        2. Write Container Buffer in the new Data Block pages;
        3. switch merge new Data Block with old Data Block;
        4. map logical addresses by IBU Map Table;
      }
    }
    else{
      1. write multiple compressed logical pages in the Container
Buffer
      2. write Container Buffer in the last Data Block
         or Log Block page;
      3. map the logical pages by IBU Map Table
    }
  }
  else{
    if(typeof(subReq) == SEQUENTIAL_TYPE){
      write subReq in the new Data Block and switch merge
      with old Data Block;
      Map the logical pages by IBU Map Table;
    }
    else{
      // subReq is small incompressible
      if(subReq has the spatial locality){
        collect small incompressible sub-requests in a sequential
        Log Block;
        Merge sequential Log block with old Data Block
        (switch merge or partial merge);
        Map the logical pages by IBU Map Table;
      }
      else{
        write random incompressible sub-request in the Data Block
         or Log Block;
        Map the logical pages by IBU Map Table;
      }
    }
  }
}
```

شکل ۵: شبه‌کد الگوریتم نوشتن در طرح پیشنهادی

اگر زیردرخواست از نوع کوچک قابل فشرده‌سازی باشد، آنگاه صفحات منطقی زیردرخواست، فشرده شده و بنابر شرایط در بلوک داده یا لاگ بلوک نگاشت شده به زیردرخواست نوشته می‌شوند. به عنوان مثال در شکل ۶ زیردرخواست ۱ از نوع کوچک قابل فشرده‌سازی بوده و این زیردرخواست مرتبط با LBN1 است. بلوک داده نگاشت شده به این زیردرخواست (Data Block 11) فضای کافی برای صفحات فشرده شده زیردرخواست را دارا بوده و در نتیجه صفحات منطقی فشرده شده زیردرخواست در صفحه فیزیکی با آفست ۱ بلوک داده نوشته می‌شوند.

اگر زیردرخواست، غیرقابل فشرده‌سازی باشد، در ادامه متوالی بودن آن بررسی می‌شود. اگر زیردرخواست از نوع متوالی غیرقابل فشرده‌سازی باشد، آنگاه یک بلوک داده جدید برای آن انتخاب شده و تمام صفحات منطقی زیردرخواست به ترتیب در بلوک داده جدید نوشته می‌شوند. سپس با یک عملیات ادغام جابجایی و بدون سربار زمانی آدرس‌های بلوک داده جدید با بلوک داده قدیم در جدول IBU Map Table تعویض می‌شود. به عنوان مثال در شکل ۶ زیردرخواست ۲ از نوع متوالی غیرقابل فشرده‌سازی است. این زیردرخواست در ارتباط با LBN2 است. طبق این مثال برای زیردرخواست شماره ۲ یک بلوک داده جدید (Data Block 17) اختصاص می‌یابد و سپس تمام صفحات منطقی این زیردرخواست درون آن نوشته می‌شوند. در نهایت با یک عملیات ادغام جابجایی فیلدهای جدول IBU Map Table برای LBN 2 بروزرسانی می‌شود. اگر زیردرخواست از نوع کوچک غیرقابل فشرده‌سازی باشد، آنگاه محلیت مکانی زیردرخواست‌های کوچک مطابق با الگوریتم بخش ۳.۴ بررسی می‌شود. اگر این زیردرخواست‌ها دارای محلیت مکانی باشند، در نتیجه زیردرخواست کوچک غیرقابل فشرده‌سازی به صورت متوالی در نظر گرفته شده و صفحات منطقی آن به ترتیب درون یک لاگ بلوک متوالی نوشته می‌شود. هنگامی که لاگ بلوک متوالی پر شود، آنگاه با یک عملیات ادغام جابجایی (و تحت شرایط استثنا عملیات ادغام جزئی) با بلوک داده نگاشت شده به LBN زیردرخواست تعویض می‌شود.

به عنوان مثال در شکل ۶ زیردرخواست‌های ۳ و ۴ و ۵ همگی از نوع کوچک غیرقابل فشرده‌سازی هستند. ولی این زیردرخواست‌ها دارای ویژگی محلیت مکانی هستند و در مجموع می‌توانند یک زنجیره متوالی به اندازه یک بلوک منطقی را شامل شوند. مجموع این زیردرخواست‌ها متعلق به LBN3 هستند و پس از نوشته شدن در لاگ بلوک متوالی با PBN 18 با یک عملیات ادغام جابجایی PBN 18 به جای بلوک داده ۱۳ در جدول IBU Map Table جابجا می‌شود. ولی ممکن است که یک زیردرخواست کوچک غیرقابل فشرده‌سازی خاصیت محلیت مکانی را نداشته باشد، که در اینصورت به آن یک زیردرخواست تصادفی غیرقابل فشرده‌سازی می‌گوییم.

Sub-request 0 (Start LPN: 0 , Size: 4) Compression rate %45
Sub-request 1 (Start LPN: 4 , Size: 2) Compression rate %34
Sub-request 2 (Start LPN: 8 , Size: 4) Compression rate %85
Sub-request 3 (Start LPN: 12 , Size: 2) Compression rate %92
Sub-request 4 (Start LPN: 14 , Size: 1) Compression rate %92
Sub-request 5 (Start LPN: 15 , Size: 1) Compression rate %92
Sub-request 6 (Start LPN: 18 , Size: 1) Compression rate %92
Sub-request 7 (Start LPN: 20 , Size: 4) Compression rate %30

LBN			LPN Offset 0	LPN Offset 1	LPN Offset 2	LPN Offset 3
	Data Block	Log Block	PPN Offset - L	PPN Offset - L	PPN Offset - L	PPN Offset - L
0	10	0	0 - 0	1 - 0	2 - 0	3 - 0
1	11	1	0 - 0	0 - 0	0 - 0	
2	12	0	3 - 0	3 - 0	0 - 1	0 - 1
3	13	1	0 - 0	0 - 0	1 - 0	1 - 0
4	14	0	3 - 0	3 - 0	0 - 1	0 - 1
5	15	1	0 - 0	0 - 0	0 - 0	

Sub-request 0 : LPN(0,1) => PPN 64 , LPN(2,3) => PPN 65
Sub-request 1 : LPN(4,5) => PPN 45
Sub-request 2 : LPN(8) => PPN 68 , LPN(9) => PPN 69
LPN(10) => PPN 70 , LPN(11) => PPN 71
Sub-request 3 : LPN(12) => PPN 72 , LPN(13) => PPN 73
Sub-request 4 : LPN(14) => PPN 74
Sub-request 5 : LPN(15) => PPN 75
Sub-request 6 : LPN(18) => Log PPN 1
Sub-request 7 : LPN(20,21,22) => PPN 61 , LPN(23) => PPN 62

LBN			LPN Offset 0	LPN Offset 1	LPN Offset 2	LPN Offset 3
	Data Block	Log Block	PPN Offset - L	PPN Offset - L	PPN Offset - L	PPN Offset - L
0	**16**	0	**0 - 0**	**0 - 0**	**1 - 0**	**1 - 0**
1	11	1	**1 - 0**	**1 - 0**	0 - 0	
2	**17**	0	**0 - 0**	**1 - 0**	**2 - 0**	**3 - 0**
3	**18**	1	**0 - 0**	**1 - 0**	**2 - 0**	**3 - 0**
4	14	0	3 - 0	3 - 0	**1 - 1**	0 - 1
5	15	1	**1 - 0**	**1 - 0**	**1 - 0**	**2 - 0**

شکل ۶: مثال‌هایی از نوشتن درخواست‌ها در طرح

این نوع از زیردرخواست بنا بر شرایط درون بلوک داده یا لاگ بلوک اشتراکی نگاشت شده به زیردرخواست نوشته می‌شوند. به عنوان مثال در شکل ۶ زیردرخواست ۶ از نوع تصادفی غیرقابل قابل فشرده‌سازی بوده و مرتبط با LBN4 است. از آنجایی که بلوک داده نگاشت شده به این LBN (Data Block 14) کاملا پر شده است، در نتیجه صفحه منطقی این زیردرخواست در لاگ بلوک اشتراکی نگاشت شده به LBN4 (لاگ بلوک ۰) نوشته می‌شود.

۳-۶- عملیات زباله‌روبی در طرح پیشنهادی

به دلیل آنکه در حافظه‌های فلش قابلیت بازنویسی صفحات فیزیکی وجود ندارد، در نتیجه عملیات پاکسازی و زباله‌روبی بلوک‌های فیزیکی آلوده، به منظور ایجاد فضای خالی برای بازنویسی داده‌ها امری اجتناب ناپذیر است. در طرح‌هایی همانند ZFTL و LDC برای عملیات عملیات زباله‌روبی نیاز به یک جدول تعیین وضعیت صفحات فیزیکی (PST) است که با استفاده از آن صفحات منطقی معتبر درون هر صفحه فیزیکی شناسایی می‌شود. حجم جدول PST بزرگ بوده و اندازه آن حدود یک چهارم جدول PMT در طرح‌های مذکور است. طرح پیشنهادی IBU FTL برای عملیات زباله‌روبی هر بلوک فیزیکی نیازی به جدول PST ندارد و وضعیت صفحات منطقی معتبر درون هر صفحه فیزیکی مستقیما از طریق جدول نگاشت IBU Map Table و ابرداده‌های ذخیره شده در ناحیه فرگمنت آن صفحه قابل استخراج است. در این طرح به ازاء هر LPN ذخیره شده در ناحیه ابرداده یک صفحه فیزیکی، اگر نگاشتی به آن صفحه فیزیکی در جدول IBU Map Table وجود داشته باشد، آنگاه آن صفحه منطقی معتبر است و در عملیات زباله‌روبی باید منتقل شود.

یکی از مهمترین مزایای طرح پیشنهادی، تعداد بسیار زیاد عملیات-های زباله‌روبی از نوع ادغام جابجایی است. این نوع عملیات هیچگونه سربار زمانی برای انتقال داده‌ها به سیستم تحمیل نمی‌کند و فقط آدرس‌ها را در جدول نگاشت IBU Map Table بروزرسانی می‌کند. هرچه اندازه فایل‌های غیرقابل فشرده نوشته شده در سیستم بزرگتر باشد، آنگاه احتمال رخداد عملیات ادغام جابجایی نیز بیشتر است.

در طرح IBU FTL به ازاء هر بلوک منطقی در سیستم یک بلوک فیزیکی داده و یک لاگ بلوک اشتراکی تخصیص می‌یابد. هنگامی‌که یک لاگ بلوک اشتراکی به میزان کافی فضا برای ذخیره‌سازی یک زیردرخواست جدید را نداشته باشد، آنگاه نیاز به یک عملیات زباله-روبی برای آن لاگ بلوک اشتراکی است. از آنجایی که هر لاگ بلوک اشتراکی با چندین بلوک منطقی درگیر است (در این طرح هر لاگ بلوک میان ۸ بلوک منطقی به اشتراک گذاشته شده است)، در نتیجه برای پاکسازی یک لاگ بلوک اشتراکی نیاز به انجام چندین عملیات ادغام و زباله‌روبی میان لاگ بلوک اشتراکی و بلوک‌های داده است. این نوع از زباله‌روبی سربار زمانی برای انتقال داده‌های معتبر بر سیستم تحمیل می‌کند و وقوع آن نیز اجتناب ناپذیر است.

۴- نتایج شبیه‌سازی

از آنجایی که نرخ فشرده‌سازی در طرح پیشنهادی، تاثیر زیادی بر کارایی آن نسبت به طرح ZFTL و LDC دارد، بنابراین برای ارزیابی طرح IBU FTL از چندین نوع بارکاری متفاوت، مطابق با جدول ۱ و ۲ استفاده شده است.

جدول ۱: بارکاری‌های I/O و توزیع تعداد صفحات منطقی فشرده-شده‌ای که می‌توانند در یک صفحه فیزیکی ذخیره شوند

No. of pages	1	2	3	4	>= 5	Activity
Windows Workload	81%	11%	3%	3%	2%	Copy and Move windows files on Windows 7
Program Installation	21%	18%	28%	14%	19%	Install MySQL
IOZone	57%	20%	8%	10%	5%	IOZone workload execution

جدول ۲: مشخصات بارکاری‌های I/O

	Windows Workload	**Program Installation**	**IOZone**
No. of Requests	63500	22800	147450
Read Requests	35%	12%	19%
Write Requests	65%	88%	79%
Random Requests	36%	61%	41%
Sequential Requests	64%	39%	59%

در این شبیه‌سازی بارکاری ویندوزی، شامل مخلوطی از انواع فایل‌ها و درخواست‌هایی است که یک کاربر کامپیوتر شخصی ایجاد می‌کند. این بارکاری شامل درخواست‌های خواندن و نوشتن ناشی از کپی و ایجاد انواع مختلف فایل‌های پرکاربرد ویندوزی می‌شود. از جمله این موارد می‌توان به فایل‌های مالتی‌مدیا، فایل‌های اسنادی و

ماکروسافت آفیس ، فایل‌های فشرده شده همچون (rar, zip)، فایل‌های دانلود شده در استفاده‌های اینترنتی کاربر (web browsing) و همچنین فایل‌های ایجاد شده و مورد استفاده در یک محیط برنامه‌نویسی خاص (IntelliJ IDEA) می‌شود. ویژگی اصلی بارکاری‌های ویندوزی این است که تعداد زیادی از درخواست‌های نوشتن تولید شده توسط آن، از قبل فشرده شده‌اند و در نتیجه قابلیت فشرده‌سازی مجدد در سطح لایه ترجمه فلش را ندارند.

دومین بارکاری استفاده شده، بارکاری نصب برنامه است. این بارکاری شامل فرآیند نصب برنامه در محیط ویندوز می‌شود. برنامه نصب شده شامل فایل‌های اجرایی (exe) و فایل‌های DLL و انواع مختلف فایل‌های باینری استفاده شده در برنامه است. ویژگی مهم این بارکاری این است که اغلب درخواست‌های نوشتن تولید شده در آن، قابلیت فشرده‌سازی بسیار خوبی دارند. سومین بارکاری، متعلق به برنامه IOZone [۲۴] است. نرم‌افزار IOZone بارکاری‌های ایجاد شده توسط برنامه‌های OLTP را شبیه‌سازی می‌کند و انواع مختلفی از عملیات‌های I/O شامل (نوشتن، خواندن، بازنویسی تصادفی، بازنویسی متوالی و ...) را شامل می‌شود. OLTP به برنامه‌های تراکنش محوری گفته می‌شود که در هر تراکنش معمولا بخش کوچکی از داده‌ها را درون پایگاه داده درج، بروزرسانی و یا حذف می‌کنند. از سیستم‌های OLTP معمولا در تراکنش‌های مالی و سیستم‌های مدیریت ارتباط با مشتری (CRM) استفاده می‌شود. این سیستم‌ها معمولا دارای کاربران بسیار زیادی هستند که هر کدام تراکنش‌های کوتاهی را انجام می‌دهند [۲۶-۲۵].

برای ارزیابی طرح پیشنهادی از یک نسخه توسعه یافته از شبیه‌ساز SSDSim [۲۷] استفاده شده است. شبیه‌ساز SSDSim یک ابزار استاندارد برای شبیه‌سازی لایه ترجمه فلش در دستگاه‌های ذخیره‌ساز SSD است. شبیه‌ساز SSDSim به‌گونه‌ای توسعه داده شده است که شامل مولفه‌های مورد نیاز برای پشتیبانی فشرده‌سازی در سطح FTL و یک مولفه جدید ترجمه آدرس مطابق با طرح پیشنهادی باشد. مهمترین پارامترهای تنظیم شده در شبیه‌ساز SSDSim بر مبنای یک تراشه MLC NAND Flash و یک پردازنده ۲۰۰ مگاهرتزی توکار در یک SSD تجاری بوده و در جدول ۳ نشان داده شده است [۱۶].

از آنجایی‌که طبق تحقیقات گذشته، واحد فشرده‌سازی 4KB دارای نرخ فشرده‌سازی خوبی است، در نتیجه ما اندازه صفحات منطقی و فیزیکی موجود در شبیه‌سازی را برابر با 4KB در نظر می‌گیریم. این اندازه برای صفحات فیزیکی حافظه‌های فلش بسیار متداول است.

جدول ۲: پارامترهای تنظیم شده در شبیه‌ساز SSDSim برای حافظه فلش SLC NAND

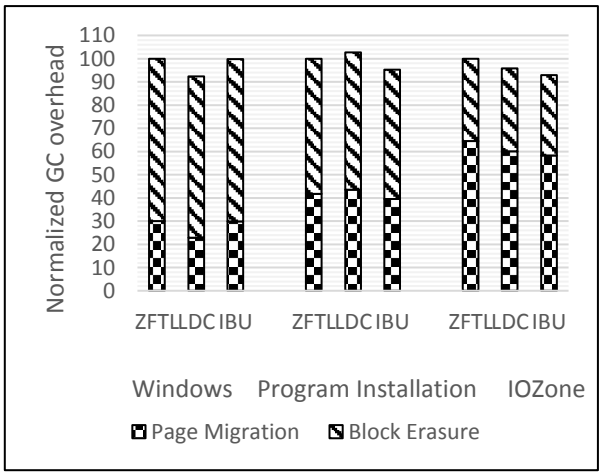
Parameter	Configuration
Page Size	4KB
Pages per Block	64
NAND Flash Memory Size	8GB
Over Provision Space	12.5%
SSD Processor Frequency	200MHz
Page Write Time	200 µs
Page Read Time	25 µs
Block Erase Time	1500 µs
Time Overheads of Compressing a 4KB Page	136 µs
Time Overheads of Decompressing a 4KB Page	33 µs

همچنین در این آزمون حدود 1GB فضای Over Provision علاوه بر فضای داده حافظه فلش (8GB) به دستگاه ذخیره‌ساز فلش تخصیص یافته است که از آن به عنوان فضای کمکی در عملیات زباله‌روبی در طرح ZFTL و همچنین به عنوان فضای لاگ بلوک‌های متوالی و لاگ بلوک‌های اشتراکی در طرح پیشنهادی استفاده می‌شود. در طرح پیشنهادی به ازاء هر ۸ عدد بلوک منطقی یک لاگ بلوک اشتراکی تخصیص داده شده است، که به‌طور مشترک از فضای صفحات خالی آن استفاده می‌کنند. لاگ بلوک‌ها در طرح‌های FTL ترکیبی باید از فضای Over provision انتخاب شوند. در محصولات صنعتی حافظه‌های فلش، فضای اضافی Over provision معمولا بین ۱۰ تا ۱۵ درصد از فضای بلوک‌های داده است. در این مقاله، یک حد وسط ۱۲.۵ درصدی برای فضای Over provision انتخاب گردید. این عدد ۱۲.۵ درصد، یک نسبت ۱ به ۸ را میان لاگ بلوک‌ها و بلوک‌های داده (و به تبع آن بلوک‌های منطقی) ایجاد می‌کند. در این شبیه‌سازی، بیشتر داده‌های بارکاری ویندوزی غیرقابل فشرده‌سازی هستند. اما اکثریت درخواست‌های بارکاری نصب برنامه، با نرخ مناسبی قابل فشرده‌سازی هستند. در بارکاری IOZone نرخ داده‌های قابل‌فشرده‌سازی با داده‌های غیرقابل فشرده‌سازی تقریبا برابر است. مهمترین معیارهای ارزیابی FTLهایی که از قابلیت فشرده‌سازی داده‌ها پشتیبانی می‌کنند، عبارتند از : سربار زباله‌روبی، میزان حافظه مصرفی برای جداول نگاشت آدرس، میانگین زمان تاخیر نوشتن درخواست‌ها و میانگین زمان خواندن درخواست‌ها.

۴-۱- ارزیابی سربار زباله‌روبی

نمودار شکل ۷ سربار زمانی ناشی از عملیات زباله‌روبی را نشان می‌دهد. این سربار شامل زمان پاک‌کردن بلوک‌های فیزیکی و زمان

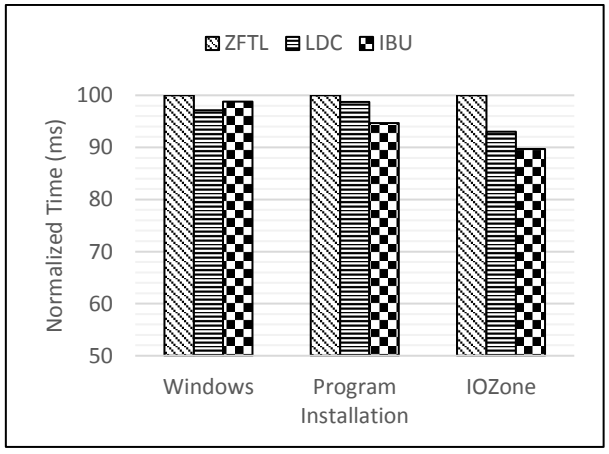
مهاجرت صفحات معتبر است. سربار عملیات زباله‌روبی در طرح LDC برای بارکاری ویندوزی (که نرخ فشرده‌سازی بیشتر داده‌های آن کم می‌باشد) نسبت به طرح ZFTL حدود ۷.۵ درصد و نسبت به طرح IBU حدود ۷.۳ درصد کمتر است. دلیل این امر این است که طرح LDC در بارکاری‌هایی با نرخ فشرده‌سازی کم، به هنگام عملیات زباله‌روبی بسیاری از داده‌هایی که قبلا فشرده نشده‌اند (به دلیل نرخ فشرده‌سازی کم) را فشرده کرده و آن‌ها را در فضای فرگمنت داخلی صفحات فیزیکی می‌نویسد. در طرح LDC به هنگام زباله‌روبی، بسیاری از فضاهای فرگمنت داخلی صفحات فیزیکی حذف می‌شود و در نتیجه صفحات فیزیکی کمتری نوشته می‌شوند. ولی در طرح‌های ZFTL و IBU فضای فرگمنت داخلی صفحات فیزیکی باقی می‌مانند. اما در یک بارکاری، هرچه نرخ فشرده‌سازی داده‌ها بیشتر شود، آنگاه طرح پیشنهادی IBU ، عملکرد بهتری در عملیات زباله‌روبی دارد. نتایج نمودار شکل ۷ نشان می‌دهد که سربار زباله‌روبی در طرح IBU برای بارکاری نصب برنامه، حدود ۷.۴ درصد نسبت به طرح LDC و حدود ۴.۷۵ درصد نسبت به طرح ZFTL کاهش داشته است. همچنین برای بارکاری IOZone ، این میزان کاهش در طرح IBU نسبت به طرح LDC حدود ۴.۸۹ درصد و نسبت به طرح ZFTL حدود ۷.۱۴ درصد است. دلیل اصلی بهبود عملیات زباله‌روبی در طرح IBU برای بارکاری‌های نصب برنامه و IOZone، نوشتن درون بلوکی درخواست‌های بروزرسانی برای داده-های قابل فشرده‌سازی و همچنین رخداد عملیات‌های ادغام جابجایی بسیار زیاد در چنین بارکاری‌هایی است. این دو عامل تعداد عملیات‌های زباله‌روبی را کاهش داده و سربار ناشی از مهاجرت صفحات داده را کم می‌کند.



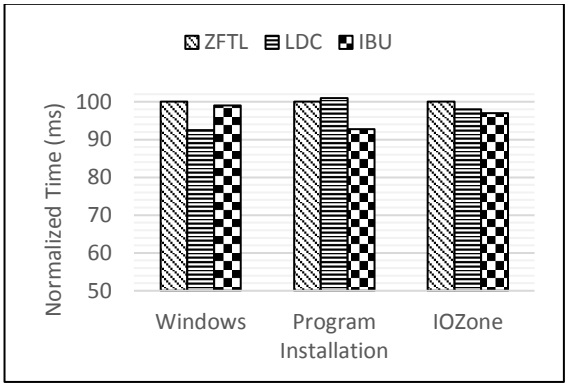
شکل ۷: سربار عملیات زباله‌روبی (نرمال شده به ۱۰۰)

۴-۲- تاخیر نوشتن و خواندن درخواست‌ها

نمودار شکل ۸ و شکل ۹ مقایسه تاخیر نوشتن و خواندن را در سه طرح IBU Mapping ، ZFTL و LDC نشان می‌دهد. برای بارکاری ویندوزی، تاخیر نوشتن در طرح IBU Mapping نسبت به طرح ZFTL حدود ۱.۲ درصد کاهش و نسبت به طرح LDC حدود ۱.۷ درصد افزایش دارد. ولی تاخیر نوشتن طرح پیشنهادی در بارکاری Program Installation نسبت به طرح LDC حدود ۴ درصد و در بارکاری IOZone حدود ۳.۳ درصد کاهش داشته است. به طور کلی طرح پیشنهادی در بارکاری‌های Program Installation و IOZone نسبت به طرح‌های ZFTL و LDC تاخیر نوشتن کمتری دارد، ولی در بارکاری ویندوزی تفاوت چشمگیری دیده نمی‌شود. به‌طور کلی نمودار شکل ۸ نشان می‌دهد که هر چه نرخ فشرده‌سازی داده‌ها در یک بارکاری بیشتر باشد، آنگاه طرح پیشنهادی IBU Mapping تاخیر نوشتن کمتری نسبت به طرح‌های دیگر دارد. این میزان کاهش بیشتر ناشی از جداسازی درخواست‌های نوشتن متوالی از تصادفی و نگاشت بلوکی درخواست‌های متوالی است. نگاشت بلوکی درخواست‌های متوالی باعث می‌شود که عملیات‌های زباله‌روبی کمتری حین نوشتن درخواست‌های جدید اتفاق بیفتد و بدین ترتیب میانگین کلی تاخیر نوشتن درخواست‌ها کاهش یابد. با توجه به نمودار شکل ۸، دلیل کاهش تاخیر نوشتن طرح IBU Mapping نسبت به طرح LDC در بارکارهای Program Installation و IOZone ، سربار کمتر عملیات زباله‌روبی حین نوشتن درخواست‌ها است. همچنین طرح ZFTL ، بیشترین تاخیر نوشتن را دارد که دلایل اصلی آن ناشی از سربار زمانی فشرده‌سازی داده‌های غیرقابل فشرده و انجام عملیات‌های زباله‌روبی بیشتر حین نوشتن درخواست‌های جدید است.



شکل ۸: میانگین تاخیر نوشتن درخواست‌ها (نرمال شده به ۱۰۰)



شکل ۹: میانگین تاخیر خواندن درخواست‌ها (نرمال شده به ۱۰۰)

به طورکلی طرح LDC بارکاری ویندوزی بیشترین کاهش تاخیر خواندن را نسبت به طرح‌های دیگر دارد، اما طرح پیشنهادی IBU برای بارکاری‌های نصب برنامه و IOZone بهترین عملکرد را داشته است. به عنوان مثال تاخیر خواندن در طرح IBU Mapping نسبت به طرح LDC حدود ۸ درصد در بارکاری Program Installation و حدود ۲ درصد در بارکاری IOZone کاهش نشان می‌دهد. تحلیل نمودارهای تاخیر خواندن و تاخیر نوشتن و همچنین نمودار سربار زمانی زباله‌روبی نشان می‌دهد که در بارکاری‌های سنگین، هرچقدر سربار زمانی عملیات زباله‌روبی در یک طرح کمتر باشد، آنگاه تاخیر خواندن و نوشتن داده‌ها نیز به میزان مناسبی کاهش می‌یابد.

۴-۳- میزان حافظه نگاشت

مهمترین ویژگی طرح پیشنهادی، کاهش قابل‌توجه مقدار حافظه مورد نیاز برای جدول نگاشت آدرس نسبت به طرح ZFTL و LDC است. از آنجایی که طرح LDC به لحاظ ساختار نگاشت آدرس همانند طرح ZFTL است و از الگوریتم ترجمه آدرس صفحه‌ای استفاده می‌کند، در نتیجه جداول نگاشت آدرس در هر دو طرح به لحاظ میزان مصرف حافظه یکسان هستند و در محاسبات فقط اشاره به طرح ZFTL شده است. مقدار حافظه مورد نیاز برای جداول نگاشت در طرح ZFTL و LDC از رابطه زیر بدست می‌آید :

$$Page\ Map\ Table\ size\ (ZFTL) = \left((Flash\ memory\ size/page\ size) \times (physical\ page\ address\ size)\right)$$

$$Page\ State\ Table\ size\ (ZFTL) = \left((Flash\ memory\ size/page\ size) \times 1\ byte\right)$$

$$Page\ Map\ Table\ size\ (ZFTL) = (8\ GB/4\ KB) \times (4\ byte) = 8\ MB$$

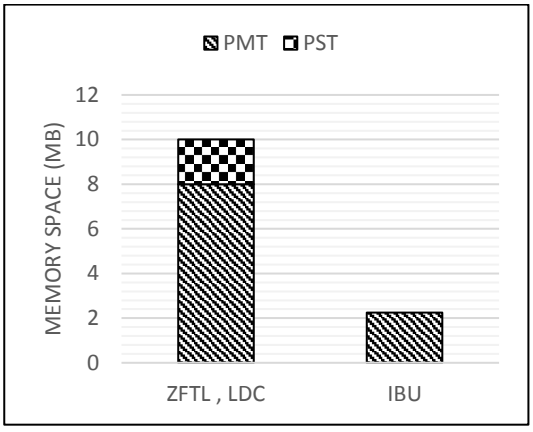
$$Page\ State\ Table\ size\ (ZFTL) = (8\ GB) \times (4\ KB) = 2\ MB$$

همچنین مقدار حافظه مورد نیاز برای طرح پیشنهادی IBU FTL از رابطه زیر بدست می‌آید:

$$Map\ Table\ size\ (IBU) = (Flash\ memory\ size/block\ size) \times [(data\ block\ field\ address\ size + log\ block\ field\ address\ size) + (page\ per\ block\ number\ *\ page\ offset\ size)]$$

$$Map\ Table\ size\ (IBU) = (8\ GB/256\ KB) \times [(4\ byte + 4\ byte) + (64 \times 1)] = 2.25\ M1B$$

همانگونه که در نمودار شکل ۱۰ نشان داده شده است، مجموع حافظه نگاشت در طرح ZFTL و LDC شامل مقادیر مورد نیاز برای جداول Page Map Table و Page State Table است. با توجه به این نمودار و محاسبات انجام شده این مقدار برای طرح‌های ZFTL برابر با 10 MB است، در صورتی که حافظه نگاشت برای طرح IBU FTL برابر با 2.25 MB ارزیابی می‌شود. در نتیجه مقدار حافظه نگاشت در طرح پیشنهادی حدود ۷۷ درصد کاهش می‌یابد. این مقدار کاهش قابل توجه در طرح پیشنهادی، به‌دلیل تغییر در ساختار جدول نگاشت و ترکیبی بودن آن است.



شکل ۱۰: مقدار حافظه مورد نیاز برای جداول نگاشت آدرس

۵- نتیجه‌گیری

حافظه‌های NAND فلش به دلیل ویژگی‌های ذاتی‌شان، دارای طول عمر محدودی بوده و صفحات فیزیکی آن‌ها قابلیت بروزرسانی درون مکان را ندارند. برای پوشش این محدودیت‌ها از یک سفت‌افزار، به نام لایه ترجمه فلش استفاده می‌شود. از میان مولفه‌های درون FTL، مولفه‌های ترجمه آدرس و زباله‌روبی اثر بسیار زیادی بر کارایی حافظه‌های فلش دارند و بسیاری از تحقیقات گذشته، بر بهبود این دو مولفه متمرکز شده‌اند. یکی از روش‌های پیشنهادی برای افزایش طول عمر حافظه‌های فلش، استفاده از روش فشرده‌سازی داده‌ها در سطح لایه ترجمه فلش است که اندازه بسیار زیاد جداول نگاشت

آدرس در این طرح‌های قبلی و نیاز به عملیات زباله‌روبی پیچیده، یکی از مهمترین معایب آن‌ها است. در این مقاله یک لایه ترجمه فلش جدید (IBU FTL) برای حافظه‌های NAND فلش پیشنهاد شده است که با استفاده از روش فشرده‌سازی داده‌ها در سطح FTL و پیشنهاد یک واحد ترجمه آدرس ترکیبی جدید برای آن، مقدار حافظه مورد نیاز برای جدول نگاشت آدرس را به میزان قابل توجهی کاهش می‌دهد (٪۷۷ کاهش نسبت به طرح قبلی). ارزیابی طرح پیشنهادی توسط بارکاری‌های استاندارد، نشان می‌دهد که تاخیر نوشتن در طرح IBU FTL نسبت به طرح مشابه قبلی کاهش قابل قبولی داشته است. مهمترین برتری طرح IBU FTL نسبت به طرح‌های مشابه، در مقدار حافظه مورد نیاز برای جدول نگاشت آدرس است که آن را مناسب استفاده در SSDهای External و SSDهای مخصوص کامپیوترهای شخصی می‌کند. به دلیل نگهداری ابرداده‌های جداول نگاشت در حافظه RAM، مسئله قطع ناگهانی منبع تغذیه در حافظه‌های فلش از اهمیت بالایی برخوردار بوده و ارائه روشی بسیار کارا در طرح IBU برای پشتیبان‌گیری از ابرداده‌ها و بازیابی سریع سیستم پس از قطع ناگهانی منبع تغذیه می‌تواند به عنوان یکی از کارهای آینده در رابطه با حافظه‌های NAND فلش مطرح گردد.

مراجع

[1] S. Mittal, J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537-1550, 2016.

[2] A. Tavakkol , M. Arjomand, Hamid Sarbazi-Azad, "Network-on-SSD: A Scalable and High-Performance Communication Design Paradigm for SSDs," IEEE Computer Architecture Letters, v.12 n.1, p.5-8, January 2013.

[3] C. W. Tsao, Y. H. Chang, M. C. Yang, P. C. Huang, "Efficient victim block selection for flash storage devices", *IEEE Transactions on Computers*, vol. 64, no. 12, pp. 3444-3460, Dec 2015.

[4] D. Kim et al., "Exploiting compression-induced internal fragmentation for power-off recovery in SSD," *IEEE Trans. Comput.*, vol. 65, no. 6, pp. 1720-1733, Jun. 2016.

[5] M.-C. Yang, Y.-H. Chang, T.-W. Kuo, P.-C. Huang, "Capacity-independent address mapping for flash storage devices with explosively growing capacity," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 448-465, Feb 2016.

[6] F. Chen , T. Luo , Xi. Zhang, "CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," *Proceedings of the 9th USENIX conference on File and storage technologies*, p.6-6, February 15-17, 2011, San Jose, California.

[7] D. Ma , J. Feng , G. Li, "A survey of address translation technologies for flash memories," *ACM Computing Surveys (CSUR)*, Volume 46 Issue 3, January 2014.

[8] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of Flash Translation Layer," *Journal of Systems Architecture*, vol. 55, pp. 332-343, 23 March 2009.

[9] A. S. Ramasamy, P. Karantharaj, "RFFE: A buffer cache management algorithm for flash-memory-based SSD to improve write performance", *in Canadian Journal of Electrical and Computer Engineering*, vol. 38, no. 3, pp. 219-231, 2015.

[10] Park Dong-Joo, Choi Won-Kyung, Lee Sang-Won, "FAST: A Log Buffer Scheme with Fully Associative Sector Translation for Efficient FTL in Flash Memory", ACM Transactions on Embedded Computing Systems (TECS), Volume 6 Issue 3, July 2007.

[11] S Lee, D. Shin, Y-J Kim; "LAST: locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Operating Systems*, Volume 42 Issue 6, October 2008.

[12] Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings", ACM SIGARCH Computer Architecture News, Volume 37 Issue 1, March 2009.

[13] Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, "A Two-Level Caching Mechanism for Demand-Based Page-Level Address Mapping in NAND Flash Memory Storage Systems", *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, Chicago, IL, USA*, 12 May 2011.

[14] Ming-Chang Yang, Student Member, "Capacity-independent Address Mapping for Flash Storage Devices with Explosively Growing Capacity", 2015.

[15] Y. Park and J-S. Kim, "zFTL: Power-efficient data compression support for NAND flash-based consumer electronics devices," *IEEE Transactions on Consumer Electronics*, 57(3):1148–1156, AUG 2011.

[16] C. Ji, L.-P. Chang, L. Shi, C. Gao, C. Wu, Y. Wang, and C. J. Xue, "Lightweight Data Compression for Mobile Flash Storage," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 1–18, 2017.

[17] T. Park, J.-S. Kim, "Compression support for flash translation layer", *Proceedings of the International Workshop on Software Support for Portable Storage*, pp. 19-24, Oct. 2010.

[18] J. Ziv and A. Lempel, "A universal algorithm for sequential data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[19] J.-L. Gailly and M. Adler, "Zlib general purpose compression library (version 1.2.5)," Apr. 2010.

[20] B. Mao, S. Wu, H. Jiang, Y. Yang, and Z. Xi, "EDC: Improving the performance and space efficiency of flash-based storage systems with elastic data compression," IEEE Trans. Parallel Distrib. Syst., vol. 29, no. 6, pp. 1261–1274, Jun. 2018.

[21] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim, "Improving Performance and Lifetime of Solid-State Drives Using Hardware-Accelerated Compression", *IEEE Trans. Consumer Electronics*, vol. 57, no. 4, pp. 1732-1739, 2011.

[22] W. Liu, F. Mei, C. Wang, M. O'Neill and E. E. Swartzlander, "Data compression device based on modified LZ4 algorithm", *IEEE Trans. Consum. Electron.*, vol. 64, no. 1, pp. 110-117, Feb. 2018.

[23] M. Nazari, R. Taghizadeh, S. A. Asghari, M. B. Marvasti, and A. M. Rahmani, "FRCD: Fast recovery of compressible data in flash memories," *Computers & Electrical Engineering*, vol. 78, pp. 520–535, 2019.

[24] http://www.iozone.org/docs/IOzone_msword_98.pdf

[25] S.-P. Lim, S.-W. Lee, B. Moon, "FASTer FTL for enterprise-class flash memory SSDs", *Proc. Int. Workshop Storage Netw. Archit. Parallel I/Os (SNAPI)*, pp. 3-12, May 2010.

[26] "OLTP Performance Comparison: Solid-state Drives vs. Hard Disk Drives", Test report, Jan. 2009.

[27] Y. Hu, H. Jiang, D. Feng, L.Tian, H.Luo, Sh. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands," *Proceedings of the ICS'11*, pp. 96-107, May-Jun. 2011.