一般社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

フラッシュ向け Linux スワップシステムの評価

丹 英之[†§] 緑川 博子[‡§]

†株式会社アルファシステムズ 〒211-0053 神奈川県川崎市中原区上小田中 6-6-1
‡成蹊大学理工学部 〒180-8633 東京都武蔵野市吉祥寺北町 3-3-1
§ JST CREST
E-mail: †hidetan@ci.seikei.ac.jp, ‡midori@st.seikei.ac.jp

あらまし 物理メモリサイズを超えるプロセスの実行を目的とし，フラッシュストレージ向けに提案された Linux スワップシステムの有効性について PCIe バス接続型フラッシュストレージを用い評価した．その結果，アプリケーションが拡張メモリとしてフラッシュストレージを利用する場合，スワップデバイスとして malloc(3)を用いるよりも，ファイルシステムをとしてファイル mmap(2)で用いる方が，現時点では性能を引き出せることがわかった．そこで，既存アプリケーションでも簡単にフラッシュ拡張メモリを利用できるようにするためのフレームワークを試作した．

キーワード Linux，仮想記憶，mmap，flash メモリ，不揮発性メモリ

The Evaluation of Flash-aware Linux Swap System

Hideyuki TAN[†§] Hiroko MIDORIKAWA[‡§]

† ALPHA SYSTEMS, INC. 6-6-1 Kami-kodanaka, Nakahara-ku, Kawasaki, 211-0053 Japan
‡ Seikei University 3-3-1 Kichijoji-Kitamachi, Musashino-shi, Tokyo, 180-8633 Japan
§ JST CREST

E-mail: †hidetan@ci.seikei.ac.jp, ‡midori@st.seikei.ac.jp

Abstract In order to use a PCIe-connected flash storage as extended memory beyond DRAM main memory, the newly proposed Linux swap system designed for a flash storage was evaluated. Results showed that using malloc(3) under the new swap system, where the flash storage was used as a swap device, brought us limited advantage, compared with using file mmap(2), where the flash storage was used as a ordinary file system. By these observations, we proposed a user-friendly framework to utilize a flash-based extended memory for existing application programs and users.

Keyword Linux, Virtual Memory, mmap, flash memory, non-volatile memory

1. はじめに

最先端の科学技術において，シミュレーションや解析などに用いられる高性能計算機は必要不可欠である．このようなハイパフォーマンスコンピューティングの需要はますます高まっており，システムの高性能化に向けた技術課題と研究開発ロードマップが公開されている[1]．その中で，プロセッサ性能に対するメモリ帯域・容量が低下する，メモリウォール問題が挙げられている．この課題を解決するため，NAND 型フラッシュメモリを始めとした，MRAM，ReRAM, FeRAM などの様々な特性を持つ不揮発性メモリを 1.5 次記憶装置として主記憶装置と二次記憶装置の間に用意することで，メモリに階層をつくり，システムメモリの大容量化と，キャッシュ・プリフェッチの手法でメモリバンド幅が不足する問題を解決することが検討されている．

我々は物理メモリサイズを超える問題サイズに対応するため，PCI バス接続型のフラッシュストレージを既存の UNIX API を用い，拡張メモリとして利用するための検討を行ってきた[2]．この検討では，スワップデバイスよりファイル mmap(2)でのマップトファイルを置くストレージとする手法がよい結果を得ている．

一方，我々と同様の目的で，フラッシュストレージ

向けのスワップシステムが Linux カーネルへのパッチとして提案された[3][4].

そこで，本稿ではこの最近提案されているフラッシュストレージ向けスワップシステムを評価すると共に，その結果を踏まえ新しく作成した，フラッシュストレージを拡張メモリとして扱うメモリアロケータについて述べる.

2. フラッシュ向けスワップシステム

半導体ベースの記憶メディアは，HDD のような回転式メディアとは特性が全く異なることから，レガシなブロックデバイスでのアクセスではなく，新しいアクセス方法のためのインタフェースの提案を Fusion-io 社が行っている[4][5]. このプロジェクトの一つに，物理メモリサイズを超えるプロセスを動かす際のスワップ先にフラッシュメモリを用いる Flash-aware Linux swap がある[3][4]. Linux 3.6.0 を元に開発が行われており GitHub にて nvm-fast-swap というリポジトリで公開されている[6]. この nvm-fast-swap では，主にスワップシステムにあるグローバルロックの廃止や，スワップ情報参照における各種ロックをスワップパーティションや CPU ごとに割り当てることで，ロックの細粒度化が試みられている．特に，匿名メモリ割り当てツリー走査時の排他制御を Mutex からセマフォに変更した修正や，TLB フラッシュ時のロック細粒度化などは，Linux カーネルにも取り込まれている[7][8][9]. プロジェクトは不揮発性メモリに対応する新しいインタフェースの提案ではあるが，nvm-fast-swap は HDD をスワップデバイスとする場合と同様，フラッシュメモリをブロックデバイスとしてアクセスしている.

今回は，CentOS6.4 デフォルトの kernel である 2.6.32-358.el6(以下，2.6.32)及び，nvm-fast-swap(以下，fastswap)，元となった 3.6.0，そして最近リリースされ nvm-fast-swap での修正がいくつか取り込まれている 3.10.9 で，スワップシステムの性能を比較した.

3. フラッシュストレージの基本 I/O 性能

先ず，本研究で用いるフラッシュストレージの性能を，IO 性能ベンチマーク FIO2.1(Flexible IO test)[10]を用いて調査した.

これまで Fusion-io 社の PCIe バス型接続フラッシュストレージ ioDrive2 MLC を用いてきたが，今回は低コストで入手し易い Intel SSD 910(400GB, 1/2 Height PCIe 2.0, 25nm, MLC)を用いた．このデバイスも PCIe バスでマザーボードと接続される．PCIe バスから先は LSI SAS2008 SAS コントローラと SAS/NAND ASIC を経由して，NAND フラッシュモジュールへと繋がる．OS からは 2 個のブロックデバイスとして検出されるので，これらブロックデバイスをソフトウェア RAID でストライピング(RAID0)し，一つの仮想ブロックデバイスとして使用した．このブロックデバイスをファイル保存先として使用する場合はファイルシステム ext4fs で，スワップデバイスとして使用する場合はスワップ領域の形式 SWAPSPACE2 でフォーマットした．実験環境を表 1 に示す．FIO の計測では kernel2.6.32 を用いた.

表 1 実験環境

部位	概要
CPU	Xeon E5-2650 2.00GHz x1 (8cores)
Memory	DDR3-1600 ECC 8GiB x4 (32GiB)
Distribution	CentOS 6.4 (x86_64)
Kernel	- 2.6.32 (2.6.32-358.el6, CentOS6.4) - 3.6.0 - fastswap (3.6.0 + nvm-fast-swap) - 3.10.9
Compiler / option	gcc version 4.4.7 20120313 / -O3

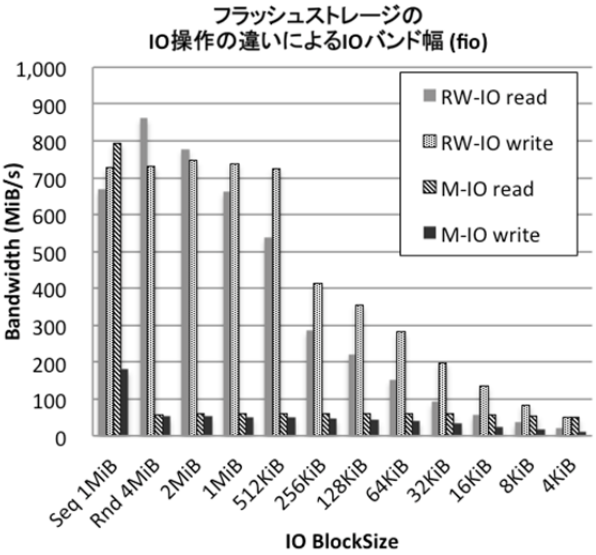

図 1 フラッシュストレージの IO バンド幅

FIO ベンチマークでは，発行する IO 操作やその IO サイズを指定することができる．IO 操作は，read(2)/write(2)+lseek(2)(以下，RW-IO 操作)，また，mmap(2)+memcpy(3)+msync(2)(以下, M-IO 操作)の組合せで計測した．M-IO 操作は，対象とするファイルを仮想メモリのアドレス空間にマップし，IO サイズ単位で当該アドレスのメモリを読み書きし，その後，ファイルへの同期命令を発行する手法であり，後述するフラッシュストレージをメモリとみなして使用する手段の一つである.

バッファ，キャッシュの影響を無くすため物理メモリサイズを超えた 40GiB のファイルを対象とし，RW-IO 操作と M-IO 操作にて，1 スレッドで計測した．図 1 は，IO サイズ 1MiB のシーケンシャルリード・ライトした場合(図左端)と，4KiB から 4MiB の IO サイズ

でランダムリード・ライトした場合の IO バンド幅である．

RW-IO 操作のランダムライトは，IO サイズ 512KiB までは 720MiB/s を超える IO バンド幅を計測したが，IO サイズ 256KiB では 512KiB の 57%と，IO バンド幅の低下が見られた．一方，ランダムリードは，IO サイズ 4MiB での IO バンド幅 862MiB/s をピークに IO サイズの縮小と共に低下したが，ランダムライトと同様，IO サイズ 256KiB を境に 512KiB の 53%と大きく低下した．これは，ストライピングのチャンクサイズを 256KiB にしていたため，256KiB 以下の IO サイズで発行された IO 操作を分割して実際の物理ブロックデバイスに発行できないことに起因するものと考えられる．

一方，M-IO 操作でのランダムリード・ライトの IO バンド幅は，10～50MiB/s と RW-IO 操作に比べ低い性能に留まっている．ioDrive2 を用いた計測でも，同様の傾向が得られている[2]．

M-IO 操作によるシーケンシャルライトでの IO バンド幅は，50～60MiB/s であったが，シーケンシャルリードでは 793MiB/s と，RW-IO 操作でのシーケンシャルリードより高い IO バンド幅を得た．これは，ファイルにマッピングしているページの先読みによる効果が出ているものと考えられる．

計測条件は異なるが，ioDrive2 の計測と比較すると SSD910 の IO バンド幅は平均して約 6 割程度，レイテンシは 2 倍程度であった．しかし，経済的コストの側面では数倍の差があり，コストパフォーマンスに優れたデバイスであると言える．

4. 物理メモリを超えた問題サイズ

物理メモリサイズを超えたデータサイズを指定したベンチマークを行うことで，前節で述べたフラッシュストレージを拡張メモリとして使用した場合の性能調査を行った．初めに，ストレージを拡張メモリとして使用する方法について，その後，各 kernel でのメモリ性能ベンチマーク STREAM[11]，姫野ベンチマーク[12]の計測結果について述べる．

4.1. フラッシュを拡張メモリとして使用する方法

ブロックデバイスであるフラッシュストレージを OS の拡張メモリとして使用するには 2 つの方法がある．

(一) メモリ領域を malloc(3)で確保し OS の仮想メモリのスワップデバイスとして使用する．

(二) ブロックデバイスにファイルシステムを載せ，ファイルを用意し mmap(2)で仮想メモリのアドレス空間にマッピングする．

(一)はアプリケーションより下のレイヤで処理が行われるので，既存アプリケーションを改変することなく使用できる．(二)の方法はマップするファイルの準備など前処理が必要になる．

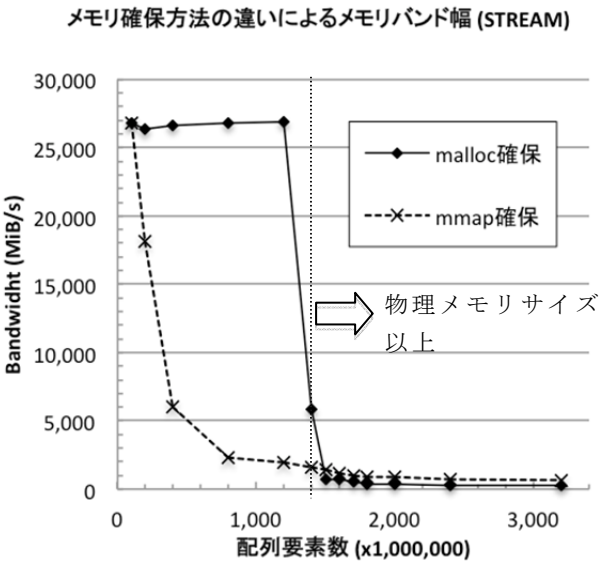

図 2 malloc/mmap 確保とメモリバンド幅(STREAM)

今回は，malloc(3)経由でスワップデバイスとして使う方法(以下，malloc 確保)，及び，各ベンチマークのメモリ確保処理に手を入れ，ファイルを mmap(2)する方法(以下，mmap 確保)で計測した．マップするファイルは，フラッシュストレージをマウントしているパス上に生成し，truncate(2)で確保したいメモリサイズと同じサイズに拡張した．

4.2. メモリ性能ベンチマーク STREAM

STREAM は，メモリバンド幅測定用のベンチマークである．double 型の 1 次元配列 a[N],b[N],c[N]に対し，OpenMP にて並列化された COPY(c[i]=a[i])，SCALE(b[i]=k*c[i])，ADD(c[i]=a[i]+b[i])，TRIAD(a[i]=b[i]+k*c[i])の操作を 1 セットとして繰り返し実行し，それぞれの操作でのメモリバンド幅を出力する．扱うデータ量に対し演算が少ないため，メモリアクセス速度が実行結果に影響する．

計測では，表 1 にある各 kernel，スレッド数 8 にて，配列操作のセットを 10 回繰り返し，配列要素数 N を 100M～3.2G 要素(データサイズ 2.2～71.5GiB)と変化させ，各サイズでのメモリバンド幅を計測した．

malloc 確保，mmap 確保での比較のため fastswap でのメモリバンド幅を図 2 に示す．malloc 確保では，配列要素数 1200M(データサイズ 26.8GiB)を超えると，ページイン・アウトが生じ，26.2GiB/s あったメモリバンド幅は急激に低下した．一方，mmap 確保では，物理メモリサイズを越える前から，マップトファイルへの同期のためメモリバンド幅が減衰する．しかし，ページキャッシュを利用できるため，物理メモリサイズを超えても malloc 確保よりメモリバンド幅は低下しない．例えば配列要素数 3.2G のとき，malloc 確保は

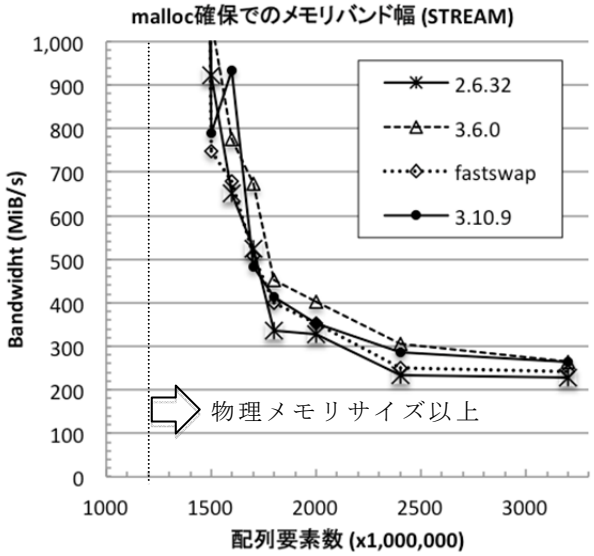

図 3 malloc 確保でのメモリバンド幅(STREAM)

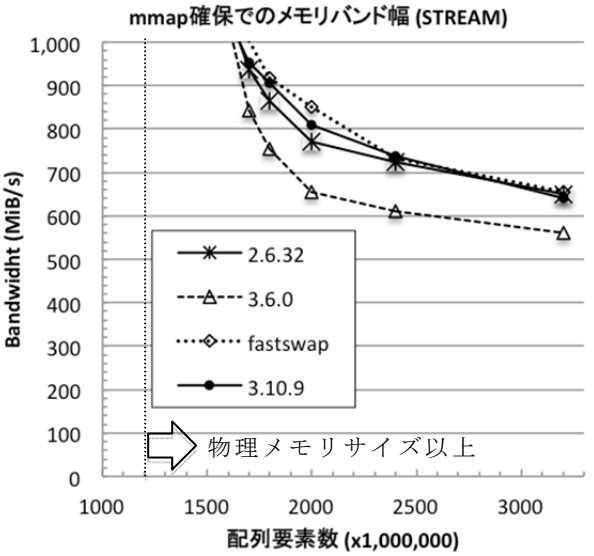

図 4 mmap 確保でのメモリバンド幅(STREAM)

241.4MiB/s だが mmap 確保では 655.6MiB/s と，約 2.7 倍近いメモリバンド幅が出ている．

各 kernel による比較のため，物理メモリサイズを超えた配列要素数での malloc 確保，mmap 確保でのメモリバンド幅をそれぞれ図 3，図 4 に示す．

malloc 確保で fastswap を 3.6.0 と比較すると，fastswap の方のメモリバンド幅が低下している．配列要素数 3.2G のとき，fastswap は 241.4MiB/s であったが，3.6.0 では 264.9MiB/s だった．nvm-fast-swap でのロック細粒度化により，並列化した配列操作は 3.6.0 よりも有利になると想定していたが，そうではなかった．3.10.9 との比較では，配列要素数 1.7G から 2.0G まで同程度のメモリバンド幅であった．

一方，mmap 確保では fastswap は 3.6.0 よりバンド幅が向上していた．配列要素数 3.2G のとき，fastswap は 655.6MiB/s であったが，3.6.0 は 560.4MiB/s だった．

図 2 から明らかなように，フラッシュメモリ向けのスワップシステムの kernel であっても，物理メモリサイズを超える場合は，malloc 確保でスワップデバイスとするよりファイル mmap(2)でのファイルストレージとする mmap 確保の方がよいことを確認できた．また，物理メモリサイズ内では mmap 確保のメモリバンド幅が低下することから，確保したいメモリ領域のサイズが物理メモリサイズを超えるとわかっている場合においてのみ，mmap 確保するのがよいと言える．

4.3. 姫野ベンチマーク

姫野ベンチマークは，非圧縮流体解析コードの性能評価のために作成されたベンチマークで，ポアッソン方程式解法をヤコビの反復法で解く場合の計算カーネルの処理速度を計測する．C-OpenMP 版を元に計算サイズ XXL(2048x1024x1024,データサイズ 112GiB)を用

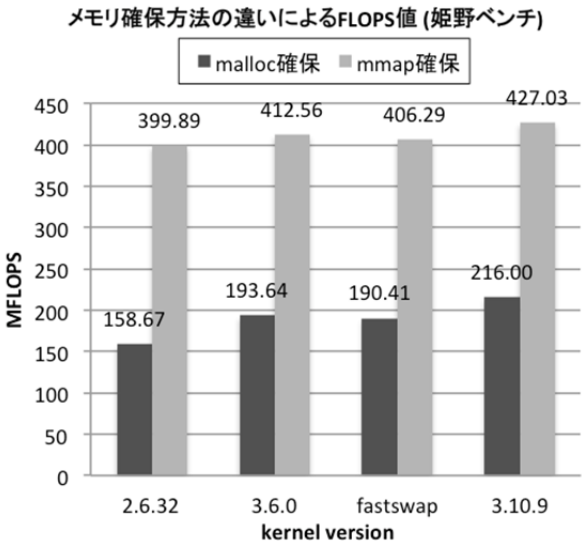

図 5 malloc/mmap 確保と FLOPS 値(姫野ベンチ)

意した．

各 kernel，スレッド数 8 にて計測した結果を図 5 に示す．malloc 確保と mmap 確保のどちらにおいても，2.6.32 が最低性能で，fastswap と 3.6.0 の比較では，むしろ fastswap のほうが，若干性能が低い結果となった．最も性能が高いのは 3.10.9 である．malloc 確保の場合，3.10.9 は，fastswap の 113%, 2.6.32 の 136%の性能を達成している．mmap 確保では，その差は小さいものの fastswap の 103%, 2.6.32 の 106%の性能になっている．malloc 確保と mmap 確保の比較では，すべての kernel において，mmap 確保が malloc 確保に比べ，2～2.5 倍高性能であることがわかった．

これらの結果より，フラッシュ向けスワップシステムの kernel であっても，現状では，スワップシステムを使用した malloc 確保に比べ，スワップシステムを用

いない mmap 確保のほうが，物理メモリサイズを超えるデータ処理を行う応用にとって，有利であることが明らかになった．

5. フラッシュ拡張メモリのための API

前節では flash 向けスワップシステムについてベンチマークを用いた評価を述べた．メモリ確保方法の違いによる結果から，フラッシュストレージを拡張メモリとして使用する場合には，malloc(3)経由でスワップデバイスとするよりも，mmap(2)でマップするファイルを置くストレージとして用いた方がよいことを確認できた．

そこで，本章では既存のアプリケーションの変更を最小限にしたまま，容易にフラッシュストレージを拡張メモリとして利用できるようにするための API を提案する．

5.1. フラッシュ拡張メモリアロケータ libflmalloc

図 2 からもわかる通り，物理メモリが十分にある状態では，malloc 確保が有利である．そこで，実行される環境に合わせ，動的に malloc 確保，mmap 確保を切り替え，メモリ帯域の減衰を避けつつ，広大なメモリ領域を利用できるアロケータを検討した．このフラッシュ拡張メモリアロケータは，一般的なアプリケーションでのメモリの動的確保・解放に用いられている malloc(3), calloc(3), free(3)と同じインタフェースを持つ，flmalloc(), flcalloc(), flfree()から成る．

- flmalloc(), flcalloc()

全てのメモリ確保を mmap 確保にしてしまうと，ファイル生成のオーバーヘッドが大きくなることから，ある一定サイズ(現実装では 10MiB)以下のメモリ確保では malloc(3)を使用することにした．また，メモリ確保したサイズを積算しておき，実行システムの物理メモリサイズに余裕がある場合には malloc(3)を使用するようにした．malloc(3)で利用可能な物理メモリサイズは，/proc/meminfo にある MemTotal の値に制限係数(現実装では 80%)を掛けた値を用いた．これら以外の条件を mmap 確保として，フラッシュストレージ上のパスにファイルを生成し，それを mmap(2)するようにした．

- flfree()

解放するアドレスがファイルにマップされている領域であれば，mmap 確保されているメモリ領域としてマップを解除しファイルを削除する．それ以外は malloc 確保されたメモリ領域として free(3)を用いるようにした．

libflmalloc のメモリ確保ポリシーは，一)物理メモリに載るサイズの範囲では malloc 確保を行う，二)物理メモリサイズを超えそうなメモリ確保から順次ファイル mmap(2)でフラッシュストレージを拡張メモリとして用いる，の 2 点である．ユーザは，メモリ確保に flmalloc()を用いることで，確保しようとするメモリのサイズが物理メモリサイズを越えてしまう場合には，フラシュ拡張メモリ，1.5 次記憶装置上に確保してもよい，という拡張メモリ利用許可を与えたことになる．

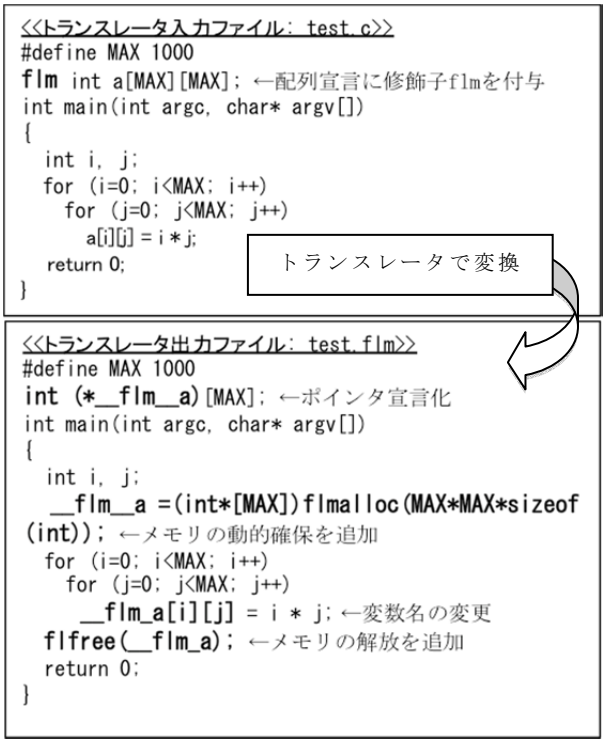

図 6 静的配列宣言の自動変換例

5.2. 既存アプリのフラッシュ拡張メモリ対応

アプリケーションのメモリを動的に確保している場合は，そのままソースにある malloc, calloc, free の文字列を置換処理し再ビルドすることで，フラッシュ拡張メモリ対応のアプリケーションとなる．これは対象とするアプリケーションのソースコードを入手でき，且つ動的に確保している場合，もしくは，アプリケーションの開発当初からフラッシュ拡張メモリに対応することを念頭に置いている場合の使い方に相当する．

数値計算系のアプリケーションでは，巨大な配列宣言を静的に行っていることがよく見られる．Fortran 由来のアプリケーションを移植したもの，特に f2c で得られたソースでは，配列宣言がそのまま静的な配列宣言に変換される．このままビルドしたロードモジュールを実行すると，プロセスのデータ領域が巨大に確保され，物理メモリサイズを超えた時点でページイン・アウトが生じることになる．このようなアプリケーションでフラッシュ拡張メモリを利用するには，静的に確保される配列を動的に確保するように変更する必要がある．しかし，複数の配列を用いているアプリケーションでは変更に手間を要する．

そこで，専用コンパイラを用い，静的領域にメモリ

を確保している部分を変換してビルドする仕組みを用意した．ユーザは，ソースにある静的配列変数の宣言部分に修飾子“flm”を付与する．この修飾子付与例を図 6 のトランスレータ入力ファイルに示す．これを専用コンパイラでビルドする際，トランスレータによって変数宣言部分が展開され，フラッシュ拡張メモリ対応ソースとしてビルドされる．トランスレータは，配列の型をポインタで参照できる型に変換して宣言し，ブロック開始時に flmalloc()を用いたメモリの動的確保処理を追加する．更に，配列参照部分をポインタ参照に置換し，ブロック終了時のメモリ解放処理を追加する．つまり，ユーザの修飾子付与は，flmalloc でのメモリ確保と同様，該当配列の領域確保に拡張メモリ利用許可を与えたことになる．

また，ソースの無いアプリケーションでもフラッシュ拡張メモリを利用できるよう libflmalloc にラッパをかぶせたダイナミックリンクライブラリも用意した．アプリケーション実行時，環境変数 LD_PRELOAD の指定で用意したライブラリを事前に読み込んでおく．これにより，アプリケーション内部の malloc(3),calloc(3),free(3)をフックし，フラッシュ拡張メモリアロケータの API を呼び出すことで，アプリケーションを改変しなくてもフラッシュ拡張メモリ対応となる．

5.3. libflmalloc を用いた姫野ベンチマーク

前節で行った姫野ベンチマークと同様の計測を kernel3.10.9 にて行った．その結果，453.57MFLOPS であった．これはメモリ確保全てを mmap 確保した場合と比較すると，106%性能が向上したことになる．libflmalloc では，メモリ確保ポリシーにより物理メモリサイズの一定レベルに収まる分までは malloc 確保，それ以降は mmap 確保を行う．姫野ベンチマークでは 7 つの配列分のメモリを確保するが，libflmalloc を用いると，計算サイズが XXL の場合では，配列の宣言順に 3 配列分 24GiB を malloc 確保で物理メモリに，残り 4 配列分の 88GiB を mmap 確保でフラッシュストレージ上に確保する．

物理メモリが malloc 確保で消費されると，mmap 確保で生成したマップトファイルのページキャッシュ分が確保できなくなることが懸念される．今回の計測では，FLOPS 値が向上していることから，物理メモリサイズの 80%という制限係数は，姫野ベンチマークの計算サイズ XXL にとって，ある程度の効果のある数値であったと考えられる．

しかし，どの程度まで malloc 分を確保しページキャッシュを残すかの割合に関しては，今後評価が必要である．

6. おわりに

フラッシュ向けスワップシステム nvm-fast-swap を評価した．その結果，少なくとも，今回用いた環境，ベンチマークでは，既存 kernel に比べ，大きな優位性が見られなかった．現時点での既存技術でフラッシュストレージを拡張メモリとして簡単に利用するには，スワップデバイスとしてではなく，ファイル mmap(2)で用いる手法がよいことを確認した．

また，この手法を既存アプリケーションに適用する枠組み libflmalloc を試作し，予備実験で若干の性能向上を得ることができた．試作した libflmalloc のメモリ確保ポリシーは，物理メモリ上への確保サイズ積算値が，検出した実行環境の物理メモリサイズの一定レベルを超えない範囲に収めるようにするだけであった，今後は，実行環境から取得できる他の情報も考慮したメモリ確保ポリシーの検討を進める．

今回はベンチマークソフトを評価に用いたが，実際に使用されているアプリケーションに適用した際の挙動について検証・評価を行いたい．

現在，様々な不揮発性メモリのデバイス，ソフトウェア，標準化インタフェースなどが活発に提案・開発されている．したがって，libflmalloc の実装方式も適宜，利用可能な最新技術を用い変更・拡張していく予定である．

文　献

[1] HPCI 技術ロードマップ白書, 戦略的高性能計算システム開発(SDHPC), 2012 年 3 月

[2] 緑川博子, 丹英之, "メモリサイズを超えるデータ処理を目的としたバス接続型 SSD の性能評価", 情報処理学会, ハイパフォーマンス研究会 Vol.2013- HPC-140, No.44, pp.1-6, (2013, 8)

[3] Shaohua Li, Improve Linux swap for High Speed Flash Storage, LinuxCon Japan 2013

[4] Nisha Talagala, Creating Flash-Aware Applications, (203-B, G-31, I-31), Flash Memory Summit 2013

[5] OpenNVM, http://opennvm.github.io, Oct.1,2013.

[6] nvm-fast-swap, https://github.com/opennvm/nvm-fast-swap/, Oct.1,2013

[7] Ingo Molnar, mm/rmap: Convert the struct anon_vma::mutex to an rwsem, Dec.2,2012
http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=5a505085f043e8380f83610f79642853c051e2f1

[8] Ingo Molnar, mm/rmap, migration: Make rmap_walk_anon() and try_to_unmap_anon() more scalable, Dec.2,2012
http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=4fc3f1d66b1ef0d7b8dc11f4ff1cc510f78b37d6

[9] Shaohua Li, smp: make smp_call_function_many() use logic similar to smp_call_function_single(), Feb.22,2013
http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=9a46ad6d6df3b547d057c39db13f69d7170a99e9

[10] FIO, http://freecode.com/projects/fio, Oct.1, 2013.

[11] John D. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, http://www.cs.virginia.edu/stream/, Oct. 1, 2013.

[12] 姫野ベンチマーク, http://accc.riken.jp/2145.htm, Oct.1,2013.