Persistent Memoryを用いたスクリプト言語のデータ永続化および復元処理の実装と評価

今村 智史[1,a)]　吉田 英司[1,b)]

概要：DRAM に比べ大容量であり不揮発性を有する新たなメモリデバイス Persistent Memory（以下，PMEM）の登場により，メインメモリ上での高速なデータ永続化が可能となった．しかしながら，PMEM の不揮発性を活用するアプリケーションを実装するには，PMEM に関する専門知識と高度なプログラミング技術が必要である．その一方，近年ではプログラミングが比較的容易なスクリプト言語を用いてアプリケーションを実装することが一般的になりつつある．そこで本論文では，煩雑な PMEM の管理機能をスクリプト言語である R と Python のバックエンドに実装し，PMEM を用いた高速なデータ永続化および復元処理を容易に行うための API を提供する．PMEM 実製品を搭載した実機サーバにおける実験を通して，R での統計分析処理や Python でのディープラーニングモデルの学習処理に本 API を適用することで SSD を用いた従来のデータ永続化および復元処理に比べその処理時間を最大約 1/8 に短縮できることを示す．

1. はじめに

PMEM とは，バイトアクセスが可能であり不揮発性を有する DIMM タイプの新たなメモリデバイスである．従来の DRAM と同様にメインメモリとして利用できる上，電源遮断時でもデータを保持し続けることが可能である．2019 年にインテルから世界初の PMEM 製品が発売され，様々な分野においてその活用が検討されている．特に，大容量のデータをメインメモリ上で管理するインメモリデータベースといったアプリケーションとの親和性が高く，SAP HANA では PMEM の活用によりデータベースの再起動を SSD に比べ 12.5 倍高速化したことが報告されている [7]．

しかしながら，PMEM をメインメモリとして使用しつつその不揮発性を活用するアプリケーションを実装するには，DRAM の場合とは異なるプログラミングモデルが必要である [8]．そのため，既存アプリケーションのソースコードを大幅に修正するか，もしくはフルスクラッチで新たに実装しなければならない．これには PMEM に関する専門知識と高度なプログラミング技術を要するため，PMEM のユースケース拡大の大きな足かせとなっている．

その一方，近年では比較的プログラミングが容易なスクリプト言語を用いてアプリケーションを実装することが一般的になりつつある．実際に，世界のプログラミング言語ランキングでは，Python，JavaScript，PHP，R といったスクリプト言語が上位 10 位以内に位置している [2]．特に，近年 AI やビッグデータ処理などによく利用される R と Python がその人気度を大きく伸ばしている．

R と Python ではメインメモリ上のデータをオブジェクトとして管理し，ブロックストレージ上のファイルを用いてオブジェクトを永続化する機能を有している．そのため，プロセスを再起動した場合でも，そのファイルからメインメモリ上にオブジェクトを復元し即座に再利用できる．

そこで，本論文では，PMEM を用いたデータ永続化／復元処理を R と Python に実装する．具体的には，煩雑な PMEM 上のオブジェクト管理機能をそれぞれのバックエンドに実装し，オブジェクトの永続化／復元処理を行うための API を提供する．これにより，R と Python のプログラマは，PMEM の不揮発性を活かすプログラミングを容易に行える．Intel® Optane™ DC Persistent Memory を搭載した実機サーバにおける R での統計分析処理や Python でのディープラーニングモデル学習処理に本 API を適用した結果，SSD を用いた従来のデータ永続化／復元処理に比べその処理時間を最大約 1/8 に短縮した．

2. 背景

2.1 Persistent Memory（PMEM）

PMEM は，DRAM と同様にロード／ストア命令によってメモリバスを介してバイトアクセスできる DIMM タイプの不揮発性メモリデバイスである．世界初の PMEM 製

[1] 株式会社富士通研究所 ICT システム研究所
[a)] s-imamura@fujitsu.com
[b)] yoshida.eiji-01@fujitsu.com

表 1　DRAM と PMEM の比較 [4]

	DRAM	PMEM
バイトアクセス	✓	✓
不揮発性		✓
DIMM 容量	<=128 GB	128, 256, 512 GB
ランダムリードレイテンシ	81 ns	305 ns
最大リードバンド幅	120 GB/s	39 GB/s
最大ライトバンド幅	80 GB/s	14 GB/s

品として，DDR4 互換であり DIMM スロットに接続できる Intel® Optane™ DC Persistent Memory が現在発売されている．

表 1 に DRAM と PMEM の簡単な比較をまとめる．現在一般的に利用可能な DRAM DIMM は最大 128 GB であるのに対し，PMEM DIMM は最大 512 GB である．現行の CPU には最大 6 枚の PMEM DIMM が接続できるため，CPU あたり最大 3 TB のメインメモリを構成できる．しかしながら，PMEM の基礎性能評価では，その性能が DRAM に比べ数倍低いことが報告されている [4]．たとえば，ランダムリードレイテンシは DRAM の約 4 倍大きく，最大リードバンド幅は DRAM の約 1/3，最大ライトバンド幅は約 1/6 である．

2.2　ハイブリッドメモリシステム

高性能な DRAM と大容量の PMEM をいずれも有効活用するために，両者を組み合わせてメインメモリを構成するハイブリッドメモリシステムが推奨されている．インテルは，こうしたシステムに対してメモリモードと App Direct モードと呼ばれる 2 種類のモードをサポートしている．メモリモードでは，PMEM が揮発性メモリとして利用され，DRAM はハードウェアによって管理される L4 キャッシュメモリとして動作する．そのため，アプリケーションを一切修正することなく PMEM をメインメモリとして利用できる．一方，App Direct モードでは，DRAM と PMEM をそれぞれ揮発性メモリと不揮発性メモリとして利用できる．この場合，PMEM はデバイスファイル（たとえば`/dev/pmem0`）として扱われ，その上にファイルシステムを構築できる．EXT4 や XFS といったファイルシステムでは DAX と呼ばれるオプションがサポートされており，このオプションを付与してマウントしたファイルシステム上のファイルをプロセスの仮想アドレス空間にメモリマップすることで PMEM へ直接アクセスできる．なお，本論文では App Direct モードを利用する．

2.3　PMEM プログラミング

PMEM をメインメモリとして利用しつつその不揮発性を活用するには，SNIA が定義する *NVM.PM.FILE* と呼ばれるプログラミングモデルに従う必要がある [8]．このモデルでは，前節で述べたように PMEM 上のファイルをプロセスの仮想アドレス空間にメモリマップすることで PMEM へのメモリアクセスを行う．また，PMEM 上のデータの永続性と一貫性を保証するために，CPU のキャッシュメモリから更新データを追い出すキャッシュフラッシュ命令や PMEM への書き込み順序を保つフェンス命令を実行する必要がある．さらに，PMEM 上で永続化したデータを再利用するには，それらのアドレス管理もアプリケーション内で行わなければならない．

```
size_t len;
void *head = pmem_map_file("/mnt/pmem/data",
                           32, 0, 0664, &len, NULL);
void *addr = head;

void **meta = (void**)addr;
addr = (void*)((size_t)addr + sizeof(void*)*2);

char *str1 = (char*)addr;
strcpy(str1, "hello");
meta[0] = (void*)((size_t)str1 - (size_t)head);
addr = (void*)((size_t)addr + strlen(str1) + 1);

char *str2 = (char*)addr;
strcpy(str2, "world!");
meta[1] = (void*)((size_t)str2 - (size_t)head);

pmem_persist(head, len);
```

図 1　PMEM 上でデータを永続化する C 言語のソースコード例

インテルが開発する Persistent Memory Development Kit（PMDK）[6] に含まれる `libpmem` ライブラリを用いて PMEM 上のデータを永続化する C 言語のソースコード例を図 1 に示す．この例では，PMEM 上のファイルをメモリマップした領域に 2 つの文字列を書き込み，それらを再利用するためのオフセットアドレスをメタデータとして保存する．また，`pmem_persist` 関数を用いて更新箇所に対してキャッシュフラッシュ命令とフェンス命令を実行する．ここでは対象のデータを 2 つの文字列に限定しているため単純な配列をメタデータとして使用しているが，様々な種類のデータを多数扱う場合にはより複雑なデータ構造をメタデータとして使用し各データの型やサイズも記録する必要がある．PMDK にはより抽象度の高いプログラミング向けのライブラリも含まれているが，いずれのライブラリを用いる場合でも PMEM 上でのアドレス管理やデータの永続性および一貫性を意識したプログラミングが必要である．

3.　PMEM を用いたデータ永続化／復元処理

本節では，まず R と Python にて使用可能な従来のデータ永続化／復元処理について説明する．その後，R と Python のバックエンドに実装する PMEM 管理機能と PMEM を用いたデータ永続化／復元処理を容易に行うための API について述べる．

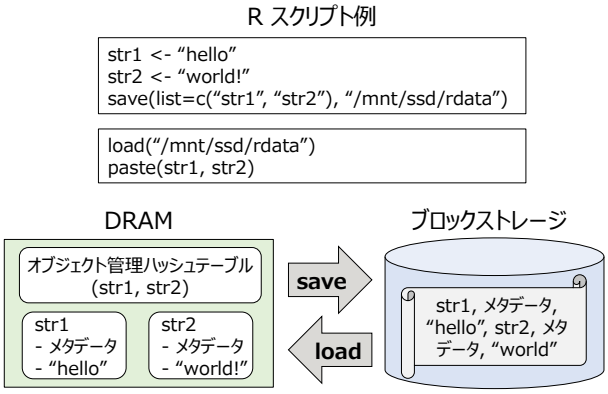


図 2 R のオブジェクト管理と従来の永続化／復元処理の例

3.1 R と Python の従来データ永続化／復元処理

R と Python では，DRAM 上のデータがその型やサイズを格納するメタデータと併せてオブジェクトという単位で管理される．作成されたオブジェクトは，同じく DRAM 上のオブジェクト管理用データ構造（R ではハッシュテーブル，Python では辞書リスト）を用いて管理され，そこに登録されたオブジェクトがプロセス中で使用できる．

また，R では，DRAM 上の指定オブジェクトをシリアライズ化しブロックストレージ上のファイルに書き出して永続化する `save` 関数とファイルのデータを読み出しデシリアライズ化することで DRAM 上にオブジェクトを復元する `load` 関数が利用できる．Python では，標準モジュール *pickle* に同様のデータ永続化／復元処理を行うための `dump` 関数と `load` 関数が実装されている．

R でのオブジェクト永続化／復元処理の例を図 2 に示す．ここでは，図 1 の例と同様に文字列オブジェクトを 2 つ作成し，`save` 関数を用いてそれらをファイル上で永続化している．また，R のプロセスを再起動した際には，このファイルを指定して `load` 関数を呼び出すことで両オブジェクトを復元し即座に再利用できる．なお，Python のオブジェクト管理と pickle モジュールを用いた永続化／復元処理に関しても同様に図示できるためここでは割愛する．

3.2 バックエンド PMEM 管理機能

本論文では，図 2 に例示した DRAM 上のオブジェクト管理と同様に，PMEM 上のオブジェクト管理機能を R と Python のバックエンドにそれぞれ実装する．R では，プロセスにメモリマップされた PMEM 上のファイル（以下，PMEM マップファイルと呼称）内に PMEM 上のオブジェクトを管理するためのハッシュテーブルを新たに構築する．このハッシュテーブルには，同ファイル上に作成された各オブジェクト（つまり，PMEM 上のオブジェクト）へのポインタがエントリとして登録される．また，PMEM マップファイルからのメモリ割り当てはその先頭領域から順に行い，割り当て済み領域の末尾アドレスを割り当てたサイズ分増加する．この割り当て済み領域の末尾アドレスと上記ハッシュテーブルへのポインタは PMEM マップファイルの先頭領域にメタデータとして保存される．なお，PMEM マップファイル内のアドレスはすべてその先頭アドレスからのオフセットとして保存されるため，新たなプロセスに同じファイルを再度メモリマップした際は各オフセットから新たな仮想アドレスを計算できる．

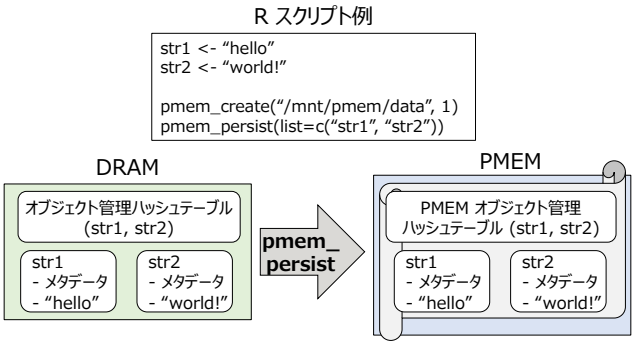
Python に関しては，数値計算用の NumPy モジュール内に R と同様の PMEM オブジェクト管理機能を実装する．そのため，本機能を使用するには NumPy モジュールをインポートする必要があり，現状では本モジュールで実装されている配列オブジェクト `ndarray` のみに対応していることに注意されたい．なお，PMEM 上のオブジェクトを管理するためのデータ構造には連結リストを使用する．

3.3 PMEM を用いたデータ永続化／復元処理 API

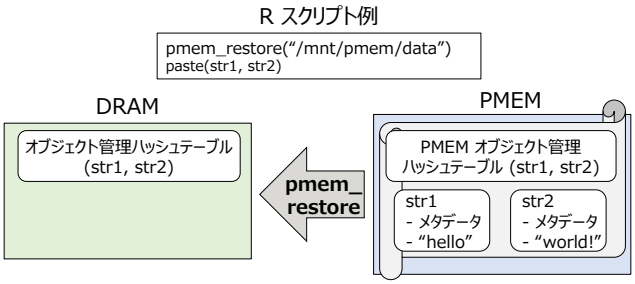
前節で説明した PMEM オブジェクト管理機能を基に，PMEM を用いた高速なデータ永続化／復元処理を容易に行うための 3 種類の API（`pmem_create`，`pmem_persist`，`pmem_restore`）を R と Python にそれぞれ実装する．これらの実装には PMDK の `libpmem` ライブラリを用いる．なお，Python では NumPy モジュール内に本 API を実装するため，NumPy モジュールをインポートした上で使用する必要がある．

`pmem_create` API は，PMEM 上のファイルの名前とファイルサイズ（GB）を引数として，ファイルを作成し R および Python の仮想アドレス空間にメモリマップする．DAX オプションを使用したファイルシステム上のファイルを指定することで，R および Python のプロセスから PMEM へ直接メモリアクセスできる．また，PMEM 上のオブジェクト管理データ構造（前述の通り R ではハッシュテーブル，Python では連結リスト）を構築し，そのオフセットアドレスをメモリマップされたファイルの先頭領域にメタデータとして記録する．図 3a の R スクリプト例では，1 GB の `/mnt/pmem/data` というファイルを PMEM 上に作成しメモリマップしている．

`pmem_persist` API は，DRAM 上のオブジェクトのリストを引数として，(1) `pmem_create` API を用いてメモリマップした PMEM 領域に指定オブジェクトをコピーし永続化，(2) それらのオブジェクトを PMEM オブジェクト管理データ構造に登録，(3) PMEM オブジェクト管理データ構造を永続化という 3 ステップの処理を行う．ステップ (1) と (3) の処理には，`libpmem` ライブラリの `pmem_memcpy_persist` 関数と `pmem_persist` 関数をそれぞれ使用する．また，ステップ (1) では，コピー対象のオブジェクトが有するデータを分割し複数スレッド（デフォルトでは 8 スレッド）を用いて `pmem_memcpy_persist` 関数を並列実行する．図 3a に示す R での例では，文字列を格納する 2 つのオブジェクト `str1` と `str2` を PMEM 上にコピーし永続化した後，そ



(a) `pmem_persist` API を用いたデータ永続化処理



(b) `pmem_restore` API を用いたデータ復元処理

図 3 R での PMEM を用いたデータ永続化／復元処理の例

れらを PMEM オブジェクト管理ハッシュテーブルに登録している．そのため，`pmem_persist` API の実行が完了した時点で両オブジェクトの永続性が保証される．

最後に，`pmem_restore` API では，`pmem_create` API を用いて作成した PMEM 上のファイルを引数として，そのファイル内の PMEM オブジェクト管理データ構造に登録された全オブジェクトを DRAM 上のオブジェクト管理データ構造に再登録する．これにより，R および Python のプロセスから PMEM 上のオブジェクトを即座に再利用できる．図 3b に示す R での例では，PMEM オブジェクト管理ハッシュテーブルに登録された 2 つのオブジェクト（`str1` と `str2`）を DRAM 上のオブジェクト管理ハッシュテーブルに再登録した後，文字列を連結する `paste` 関数でそれらを使用している．なお，Python の NumPy モジュールにおいても，`ndarray` オブジェクトに限り図 3 に示した例と同様のデータ永続化／復元処理が可能である．

4. 評価

本節では，PMEM 実製品を搭載した実機サーバにおいて今回実装した API の有効性を評価する．具体的には，3 通りのユースケースにおいてデータ永続化／復元処理に要する時間を R および Python の従来機能と PMEM 活用 API 間で比較する．

4.1 評価環境と評価方法

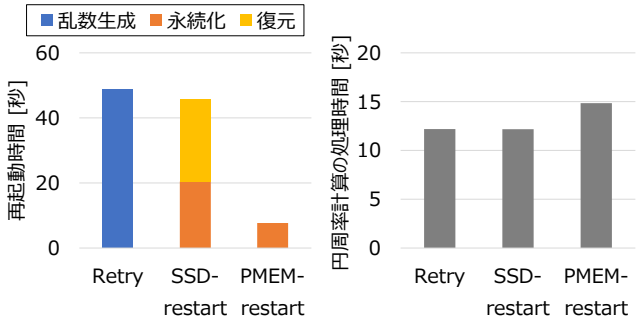
本評価では，18 コアの Xeon® Gold 6240M プロセッサ，192 GB の DRAM（32 GB DDR4 DIMM × 6），768 GB の PMEM（128 GB Intel® Optane™ DC Persistent Memory × 6），400 GB の WD Ultrastar® DC SS530 SAS SSD を搭載した PRIMERGY RX2540 M5 サーバを用いる．なお，安定した実験結果を得るためにハイパースレッディングは無効にする．PMEM は *interleaved App Direct* モードに設定し，6 枚の DIMM をインターリーブして単一の PMEM デバイス（`/dev/pmem0`）として使用する．そして，このデバイス上に EXT4 ファイルシステムを構築し，DAX オプションを用いてマウントする．Linux カーネルは 5.4.0-51-generic，R のバージョンは 3.6.1，Python のバージョンは 3.8.2 である．また，Python のモジュールとして NumPy 1.19.0 と TensorFlow 2.2.0 を用いる．

本論文にて実装した API の有効性を示すために，各ユースケースにおいて処理に必要なデータをメインメモリ上に格納し終えた時点で R および Python のプロセスを再起動する場合を想定する．そして，プロセス再起動に要する時間とデータの処理時間を *Retry*，*SSD-restart*，*PMEM-restart* の 3 通りの実行で比較する．Retry では，データの永続化／復元処理は行わず，プロセス再起動後に再度データを DRAM に格納する処理を行う．SSD-restart では，DRAM 上のデータを SSD 上のファイルに永続化し，プロセス再起動後にそのファイルからデータを復元する．このデータ永続化／復元処理には，第 3.1 節で述べた R の `save`/`load` 関数，Pickle モジュールの `dump`/`load` 関数を使用する．PMEM-restart では，SSD-restart と同様のデータ永続化／復元処理を本論文で実装した `pmem_persist`/`pmem_restore` API を用いて行う．

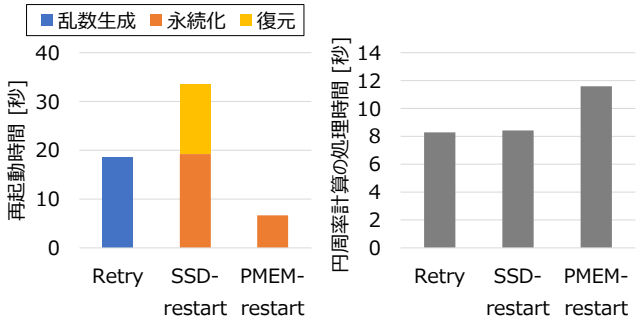
4.2 モンテカルロ法の評価結果

1 つ目のユースケースは，モンテカルロ法による円周率計算である．この実験では，10 億個の乱数を格納した 8 GB の配列を DRAM 上に 2 つ生成し，モンテカルロ法によりそれらの乱数から円周率を計算する．ここでは，2 つの配列を生成し終えた時点でプロセスを再起動する．

このユースケースの R での評価結果を図 4a に示す．Retry では，プロセス再起動後に 2 つの乱数配列を再び生成する必要があり，この処理に長い時間を要する．SSD-restart では，DRAM 上の乱数配列 2 つを SSD 上のファイルに永続化し，プロセス再起動後にそれらを DRAM 上に復元して円周率計算を行う．そのため，乱数配列を再度生成する必要はないが，永続化／復元処理に Retry と同程度の時間を要する．この永続化／復元処理に要する時間の大部分はオブジェクトのシリアライズ化／デシリアライズ化の処理時間である．これに対し，PMEM-restart では，永続化／復元処理に要する時間を SSD-restart に比べ約 1/6 に短縮している．永続化処理では DRAM から PMEM へのオブジェクトのコピーに多少の時間を要するが，復元処



(a) R での評価結果



(b) Python での評価結果

図 4 モンテカルロ法による円周率計算の評価結果

理はごくわずかな時間で完了している．また，再起動後の円周率計算の処理時間を比較すると，Retry と SSD-restart では同程度の処理時間となっている．これは，いずれの場合でも 2 つの乱数配列が DRAM に格納されるためである．これに対し，PMEM-restart では PMEM 上にコピーした乱数配列をそのまま再利用するため，PMEM の低い性能により処理時間が約 22%増加した．

また，Python での同様の評価結果を図 4b に示す．Python では R に比べ乱数生成が高速であるが，SSD-restart の永続化／復元処理に要する時間は R での結果と大きく変わらない．そのため，SSD-restart での再起動には Retry の場合の約 2 倍の時間を要する．これに対し，PMEM-restart では永続化／復元処理に要する時間を SSD-restart に比べ約 1/5 に短縮した．ただし，PMEM-restart の場合の円周率計算の処理時間は SSD-restart に比べ約 40%増加している．

4.3 R での重回帰分析の評価結果

次に，R にて重回帰分析を行う場合の評価を実施する．この実験では，ニューヨーク市の 1 ヶ月分のタクシーデータ（約 2.6 GB の CSV ファイル）[5] を使用する．`fread` 関数を用いて SSD 上の CSV ファイルを DRAM 上に読み出した後，乗客数や走行距離など 5 種類の値を説明変数，支払い額を目的変数とした重回帰分析を `lm` 関数を用いて実行する．ここでは，`fread` 関数の実行が完了した時点でプロセスを再起動する．

本ユースケースでの評価結果を図 5 に示す．Retry では，プロセス再起動後に CSV ファイルを再度読み出す必要があり，この処理に約 19 秒要している．SSD-restart では DRAM 上に読み出したデータを永続化／復元するが，これらの処理に CSV ファイルの読み出し処理以上の時間を要している．これに対し，PMEM-restart では永続化／復元処理に要する時間を SSD-restart に比べ約 1/3 に短縮し，Retry よりも高速な再起動を実現している．また，再起動後の重回帰分析の処理時間を比較すると，PMEM-restart による処理時間の増加が Retry と SSD-restart に対してわずか 5%程度であることが分かる．これは，重回帰分析の処理が CPU バウンドであり，DRAM と PMEM の性能差の影響が小さいためであると考えられる．

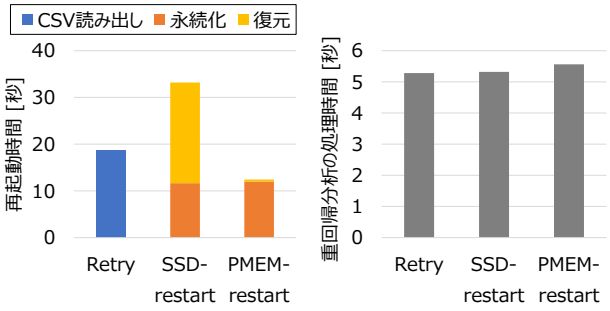


図 5 R での重回帰分析の評価結果

4.4 Python での CNN 学習処理の評価結果

3 つ目のユースケースは，Python での画像データを用いた畳み込みニューラルネットワーク（CNN）の学習処理である．この実験では，入力データとして CIFAR10 データセット [9] を使用し，TensorFlow モジュールを用いて CNN モデルの構築および学習処理を行う．CNN モデルには，Github にて公開されている CIFAR10 データセット用のサンプルモデル [3] を使用する．CIFAR10 データセットは 10 種類のカテゴリに分類された画像ファイル（JPG ファイル）6 万枚を含んでおり，これらの画像ファイルはカテゴリ毎にそれぞれ異なるディレクトリに格納されている．このデータセットの合計サイズは約 240 MB である．本実験では，6 万枚の画像ファイルを SSD から DRAM 上に読み出し，そのデータを CNN モデルの入力として使用するために NumPy の配列オブジェクト `ndarray` に変換する前処理を行う．その後，TensorFlow モジュールの `fit` 関数を用いて CNN モデルの学習処理を実行する．ここでは，DRAM 上の画像データに対する前処理が完了した時点でプロセスを再起動する．

図 6 にこの実験の結果を示す．Retry では，プロセス再起動後に SSD からの画像ファイル読み出しと前処理を再度行う必要があり，これらの処理に約 13 秒を要する．これに対し，SSD-restart では，前処理済みの画像データを永続化／復元することで再起動に要する時間を Retry に比べ約 1/4 に削減している．さらに，PMEM-restart では永

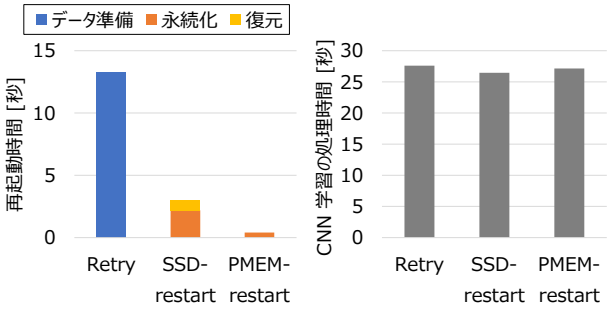


図 6 Python での CNN 学習処理の評価結果

続化／復元処理に要する時間を SSD-restart に比べ約 1/8 に削減している．また，プロセス再起動後の CNN 学習処理時間に関しては，復元したデータを DRAM に格納する SSD-restart と PMEM に格納する PMEM-restart の間でほとんど差が見られない．この結果から，このユースケースでは，本論文にて実装した API をほぼオーバーヘッドなしで使用できることが分かる．

5. 関連研究

PMEM 製品の登場により PMEM を活用したアプリケーションが数多く実装され始めている．その中でも，PMEM の不揮発性を活用するインメモリデータベースの代表として SAP HANA が挙げられる．HANA では，大容量であり更新頻度の低い Column Store Main と呼ばれるデータ構造を PMEM に格納することで，メインメモリ上で扱えるデータ量を増大しデータベースの再起動時間を大幅に削減している [1,7]．また，Renen らは，様々な条件下で実施した PMEM の基礎性能評価結果を基に，PMEM を活用するデータベースシステム向けに新たなロギング技術とページフラッシュ技術を提案しデータベースのスループット向上を達成している [10]．こうしたデータベースでは PMEM の利点を最大限活用できる一方，第 2.3 節で述べたようにそれらの実装には高度なプログラミングが必要である．

You らは，本論文と同様にスクリプト言語である JavaScript での PMEM 活用に着目し，PMEM 上のオブジェクトを管理する persistent object pool や PMEM 上のオブジェクトの作成とアクセスを容易に行うための API を JavaScript のランタイムに実装している [11]．この persistent object pool では PMEM 上でのトランザクション処理もサポートしている．本論文では，PMEM を活用した高速なオブジェクト永続化／復元処理をより簡単に実行するための API を同様のスクリプト言語である R と Python 向けに実装し，PMEM 実製品を搭載した実機サーバにおいてその有効性を示した．

6. おわりに

本論文では，煩雑な PMEM の管理機能をスクリプト言語である R と Python のバックエンドに実装し，PMEM を用いた高速なデータ永続化／復元処理を行うための API を提供した．これにより，R と Python のプログラマは PMEM の永続性を活用するプログラミングを容易に行うことができる．実機サーバにおける評価では，本 API を使用することで SSD を用いた従来のデータ永続化／復元に比べその処理時間を最大約 1/8 に短縮できることを示した．今後は，PMEM 上のオブジェクトを DRAM に再度コピーする復元処理やオブジェクト作成時にその格納先を DRAM もしくは PMEM から選択できる機能，PMEM 上でのガーベージコレクションなどを実装する予定である．

参考文献

[1] Andrei, M., Lemke, C., Radestock, G., Schulze, R., Thiel, C., Blanco, R., Meghlan, A., Sharique, M., Seifert, S., Vishnoi, S., Booss, D., Peh, T., Schreter, I., Thesing, W., Wagle, M. and Willhalm, T.: SAP HANA Adoption of Non-Volatile Memory, *Proc. VLDB Endow.*, Vol. 10, No. 12, pp. 1754–1765 (2017).

[2] Carbonnelle, P.: PYPL PopularitY of Programming Language, PYPL (online), available from ⟨http://pypl.github.io/PYPL.html⟩ (accessed November, 2020).

[3] Chollet, F.: Train a simple deep CNN on the CIFAR10 small images dataset, tensorflow.org (online), available from ⟨https://raw.githubusercontent.com/fchollet/keras/keras-2/examples/cifar10_cnn.py⟩ (accessed November, 2020).

[4] Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dulloor, S. R., Zhao, J. and Swanson, S.: Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, *CoRR*, Vol. abs/1903.05714 (2019).

[5] NYC Taxi and Limousine Commission: TLC Trip Record Data, NYC.gov (online), available from ⟨https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page⟩ (accessed November, 2020).

[6] PMDK: Persistent Memory Development Kit, pmem.io (online), available from ⟨https://pmem.io/pmdk/⟩ (accessed November, 2020).

[7] Schuster, A.: SAP HANA & Persistent Memory, SAP (online), available from ⟨https://blogs.sap.com/2018/12/03/sap-hana-persistent-memory/⟩ (accessed November, 2020).

[8] SNIA: NVM Programming Model (NPM) Version 1.2 (2017).

[9] TensorFlow: CIFAR10 dataset, tensorflow.org (online), available from ⟨https://www.tensorflow.org/datasets/catalog/cifar10⟩ (accessed November, 2020).

[10] van Renen, A., Vogel, L., Leis, V., Neumann, T. and Kemper, A.: Persistent Memory I/O Primitives, *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, ACM, pp. 12:1–12:7 (2019).

[11] You, L., Xu, H., Zhang, Q., Li, T., Li, C., Chen, Y. and Huang, L.: JDap: Supporting in-memory data persistence in javascript using Intel's PMDK, *Journal of Systems Architecture*, Vol. 101, p. 101662 (2019).