UECFS: SSDとHDDを併用して高速なファイルアクセスを実現するファイルシステム

五味真幹[†1] 鵜川始陽[†2] 岩崎英哉[†2]

半導体記憶素子であるフラッシュメモリを使用した記憶装置の一種として，Solid State Drive (SSD) が，価格の低下や速度の向上に伴い，近年注目を集めている．HDD と比べ，読み書きが速い，耐衝撃性が高い，消費電力が小さいといった利点の反面，書き換え可能回数に上限がある，記憶容量当たりの単価が高いといった欠点もある．そのため，一般的なマシンにおいては，SSD と HDD を併用し，相補的に利用することが重要視されている．そこで本研究では，SSD と HDD の併用によって，ファイルアクセスを高速化するファイルシステム Union-Extended Cache File System (UECFS) を提案し，実装する．UECFS は，ファイルへのアクセス頻度に応じて，SSD か HDD のどちらかへファイルを自動配置する．ファイルの配置先は，ファイルのアクセス頻度の変化に応じて，動的に変更する．アクセス頻度の高いファイルのみを SSD に自動配置するため，SSD の使用量を抑えつつ，ファイルアクセスの高速化が可能である．また，ユーザは，SSD と HDD のどちらにファイルが配置されているか意識しなくてよい．UECFS は，UnionFS[1)] を拡張して実装した．UnionFS は，Linux 向けに実装されているファイルシステムであり，異なる複数のディレクトリを重ねてマウントし，単一のディレクトリの様に扱うことができる．UECFS は，UnionFS の機能を利用して，SSD 上のディレクトリと HDD 上のディレクトリを重ねてマウントし，独自のファイル自動配置機構により，ファイルアクセスを高速化する．UECFS を Linux Kernel 2.6.30.10 に実装し，実験を行ったところ，ファイルが SSD に自動配置されることにより，ファイルアクセスが高速化することを確認した．

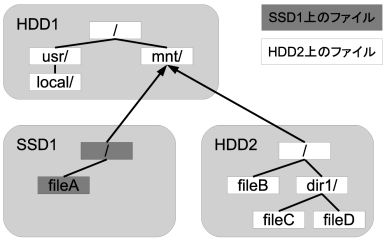
1. はじめに

半導体記憶素子であるフラッシュメモリを使用した記憶装置の一種として，Solid State Drive (SSD) が注目を集めている．SSD は，HDD と異なり，モータやアームといった可動部品を持たないため，消費電力が小さい，ランダムアクセスが速いといった利点がある．最近では，ランダムアクセス，シーケンシャルアクセスともに HDD の速度を上回る SSD も登場している．しかし，SSD は，HDD に比べて容量あたりの単価が非常に高く，書き換え可能回数に上限があるといった欠点もある．そのため，HDD の全てを SSD に替えようとすると，アクセス速度の向上によるメリットよりも，金銭的コストの増大によるデメリットの方が大きくなってしまう．そこで，SSD と HDD を併用することによって，アクセス速度の向上と，金銭的コストの低減の両方を実現することが重要視されている．

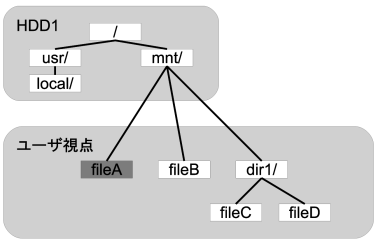
近年では，高性能な PC が低価格で入手できるため，一台の PC で，多目的のサーバを運用することが容易である．そのため，個人運用サーバや研究室向けのサーバなどでは，Web サーバ，メールサーバ，ファイルサーバなどを一台の PC で運用することも多い．多目的のサーバにおいて，SSD と HDD を併用する場合，頻繁にアクセスされるファイルを含むディレクトリのみを SSD に配置するといった方法が考えられるが，運用するサービスが多いほど作業が煩雑になってしまう．また，どのファイルやディレクトリが頻繁にアクセスされるかを，人間が判断するのは手間がかかる．

そこで本研究では，SSD と HDD を同じディレクトリに重ねてマウントし，自動的に使い分けることによってファイルアクセスを高速化するファイルシステム Union-Extended Cache File System (UECFS) を提案し，実現する．**図 1** に，UECFS を用いた場合の

†1 電気通信大学電気通信学研究科
Graduate School of Electro-Communications, The University of Electro-Communications
†2 電気通信大学情報理工学研究科
Graduate School of Informatics and Engineering, The University of Electro-Communications



(a)SSD1 と HDD2 の重ねマウント



(b) ユーザから見たファイルツリー

図 1 UECFS を用いたマウント例
Fig. 1 Example of mounting UECFS

マウント例を示す．図 1(a) では，SSD1 のルートと，HDD2 のルートを，HDD1 の`/mnt` ディレクトリに重ねてマウントしている様子を表している．ユーザからは，`/mnt` ディレクトリ下のファイルやディレクトリが，図 1(b) のように見えており，SSD1 上と HDD2 上のファイルとディレクトリが同じディレクトリ内にあるように見える．UECFS は，ファイルへのアクセス回数を数え，アクセス回数の多いファイルを SSD1 に，アクセス回数の少ないファイルを HDD2 に自動的に配置する．SSD1 の空き容量が少なくなった場合，アクセス回数が少ないファイルを SSD1 から HDD2 へ配置する．配置はファイル単位で行い，同じパスを持つファイルが SSD1 上と HDD2 上に同時に存在することはない．複数のディレクトリを重ねてマウントする機能を提供する，UnionFS を拡張して UECFS を実装した．現在のところ，UECFS は，1 つの SSD と 1 つの HDD を併用することに限定している．

UECFS の特徴は次の通りである．

- SSD と HDD を併用し，アクセス回数の多いファイルのみを SSD に自動で配置することで，SSD の使用量を抑えつつ，ファイルアクセスの高速化を実現した．
- ファイルの配置先が変更されても，ファイルのパス名は変わらないので，ユーザは，SSD と HDD のどちらにファイルが配置されているか意識しなくてよい．

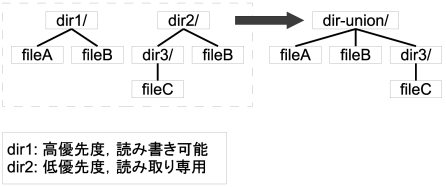


図 2 UnionFS を用いたマウント例
Fig. 2 Example of mounting UnionFS

- ユーザは，SSD と HDD のファイルシステムを，用途に応じて自由に選択できる．また，ファイルシステムに特別な変更を加える必要がなく，そのまま利用できる．

本稿の構成は，次の通りである．2 節では，UnionFS について述べる．3 節では，UECFS の設計と実装について詳細を述べる．4 節では，実装した UECFS を用いた実験の結果について述べる．5 節では，関連研究について述べる．最後に，6 節で本稿をまとめる．

2. UnionFS

UECFS は，UnionFS を拡張して実装しているため，UnionFS との共通部分が多い．本節では，UnionFS に関して，UECFS の設計と実装を理解する上で重要な内容について説明する．

2.1 概　　要

UnionFS は，Linux 向けに実装されているファイルシステムであり，複数の異なるディレクトリ（UnionFS では，ブランチと呼ぶ）を重ねてマウントし，単一のディレクトリのように扱うことができる．重ねるディレクトリは，異なるファイルシステムに属していても構わない．ブランチには優先度があり，異なるブランチに同名のファイルが存在する場合，優先度が高いブランチに属すファイルを処理する．また，ブランチには読み取り専用か，読み書き可能かをマウント時に指定する必要がある．ただし，優先度が最も高いブランチは，読み書き可能に指定しなければならない．これは，読み取り専用ブランチに属すファイルに対して書き込みが行われた場合，そのファイルを優先度が最も高いブランチにコピーしてから，コピー先のファイルに対して書き込みを行うためである．

図 2 に，UnionFS を用いたマウント例を示す．この例では，`dir1` と `dir2` を重ねて，`dir-union` にマウントしている．`dir1` の優先度を `dir2` より高くし，`dir1` を読み書き可能，`dir2` を読み取り専用と指定している．ユーザには，`dir-union` にファイルやディレクトリが存在しているように見えるが，実際には，

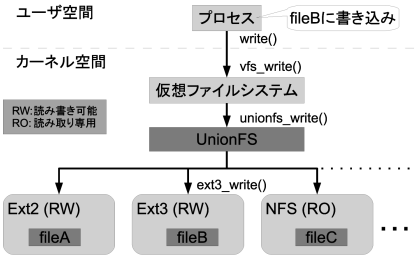


図 3 UnionFS の構成図
Fig. 3 Framework of UnionFS

```
struct uecfs_node {
  /* ファイルへのアクセス回数 */
  unsigned long access_count;

  /* ファイルの配置先を
   * 変更するかどうかのフラグ
   */
  int flag_move;
}
```

図 4 uecfs_node 構造体
Fig. 4 uecfs_node structure

dir1 や dir2 にファイルやディレクトリの実体が存在している．そのため，dir-union に属すファイルやディレクトリに対して読み書きを試みると，dir1 や dir2 に属すファイルやディレクトリに対して読み書きが行われる．ブランチの優先度により，dir-union に属す fileB に対して読み書きを試みると，dir1 に属す fileB に対して読み書きが行われる．dir2 が読み取り専用と指定されているため，dir-union に属す fileC に対して書き込みを試みると，dir1 に dir3 と fileC がコピーされ，dir1 に属す fileC に対して書き込みが行われる．

2.2 実現方法

Linux カーネルでは，プロセスがファイル操作を行う場合，読み書きなどの命令がシステムコールを介して仮想ファイルシステムへと送られ，さらに仮想ファイルシステムから，ローカルファイルシステム (Ext2, Ext3，ReiserFS，NFS など) に送られる．UnionFS は，仮想ファイルシステムとローカルファイルシステムの間に位置しており，ファイル操作に関する命令を仲介する．UnionFS が仮想ファイルシステムからファイルに対する命令を受け取ると，対象のファイルが存在するローカルファイルシステム（ブランチ）を見つけて，受け取った命令をそのまま送る．

図 3 に UnionFS の構成図を示す．プロセスが，fileB に対する書き込みのために，write システムコールを呼ぶと，その書き込み命令は仮想ファイルシステムを通して UnionFS へ送られる．そして，UnionFS は，fileB が存在する Ext3 に対して書き込み命令を送る．

3. 設計と実装

UECFS は，2 節で説明した UnionFS を拡張することで実現している．本節では，UnionFS に拡張を加えた部分を重点的に説明する．

3.1 メモリ上のデータ構造

3.1.1 uecfs_node 構造体

図 4 の uecfs_node 構造体は，ファイルの自動配置に必要な情報を管理する，ファイル毎の構造体である．ファイルへのアクセス回数 (access_count メンバ) と，ファイルを異なるブランチに移動するかどうかのフラグ (flag_move メンバ) を管理している．uecfs_node 構造体は，ブランチに属すファイルに初めてアクセスする際に生成され，ファイルを異なるブランチに移動する際に破棄される．

3.1.2 共通ファイルモデル

Linux カーネルにおいて，仮想ファイルシステムは，ローカルファイルシステムのファイル操作を抽象化するために，共通ファイルモデルを用いている．共通ファイルモデルは，次のオブジェクトから構成される．

スーパーブロックオブジェクト
　マウントされたファイルシステムに関する情報を保持するオブジェクト．

i-ノードオブジェクト
　個々のファイルに関する一般的な情報を保持するオブジェクト．

ファイルオブジェクト
　オープンされているファイルとプロセスとの間のやりとりに関する情報を保持するオブジェクト．

d-エントリオブジェクト
　ディレクトリエントリと，対応するファイルとのリンクについての情報を保持するオブジェクト．

2 節で説明したように，UnionFS は，仮想ファイルシステムとローカルファイルシステムの間に位置し，ファイル操作に関する命令を仲介する．ここでいう仲介とは， 仮想ファイルシステムから送られた，共通ファイルモデルのオブジェクトに対する命令を，ローカルファイルシステムにそのまま送ることである．UnionFS は，ファイルやディレクトリを重ねているように見せるために，異なるブランチに属すがパスが

```
struct unionfs_file_info {
  ...

  /* 各ブランチに属すファイル
   * オブジェクトへのポインタの配列
   */
  struct file **lower_files;
};

struct unionfs_inode_info {
  ...

  /* 各ブランチに属す i-ノード
   * オブジェクトへのポインタの配列
   */
  struct inode **lower_inodes;
};

struct unionfs_dentry_info {
  ...

  /* 各ブランチに属す d-エントリ
   * オブジェクトへのポインタの配列
   */
  struct path *lower_paths;
};

struct unionfs_sb_info {
  ...

  /* 各ブランチの情報を管理する
   * 構造体の配列
   */
  struct unionfs_data *data;
};

struct unionfs_data {
  ...

  /* ブランチが扱うスーパーブロック
   * オブジェクトへのポインタ
   */
  struct super_block *sb;
};
```

図 5 UnionFS の共通ファイルモデル
Fig. 5 Common file model of UnionFS

等しいファイルやディレクトリを，まとめて管理している．例えば，UnionFS のファイルオブジェクトは，**図 5** のように定義されている．unionfs_file_info 構造体の lower_files メンバは，各ブランチに属すファイルオブジェクトへのポインタの配列であり，配列の各添字はブランチと対応している．i-ノードオブジェクト（unionfs_inode_info 構造体），d-エントリオブジェクト（unionfs_dentry_info 構造体）も同様に

```
struct uecfs_data {
  ...

  struct super_block *sb;

  /* uecfs_node を管理するリスト */
  struct radix_tree_root *uecfs_list;
};
```

図 6 uecfs_data 構造体
Fig. 6 uecfs_data structure

して，各ブランチに属すオブジェクトへのポインタの配列を管理している．スーパーブロックオブジェクト（unionfs_sb_info 構造体）では，各ブランチの情報を管理する unionfs_data 構造体の配列を定義しており，unionfs_data 構造体の sb メンバは，ブランチが扱うスーパーブロックオブジェクトへのポインタである．

UECFS では，UnionFS のオブジェクトの定義をそのまま利用している（ただし構造体の名前は unionfs_ではなく uecfs_で始まるように変更している）が，各ブランチの情報を管理する uecfs_data 構造体には，uecfs_node のリストをツリー構造で管理する uecfs_list メンバを追加している．uecfs_list は，i-ノード番号をキーとして，対応する uecfs_node を管理している．

3.2 ファイルアクセス回数のカウント

仮想ファイルシステムからファイルの読み取り命令が UECFS に送られると，uecfs_read 関数（**図 7**）が呼び出される．この関数では，読み取りたいファイルが存在するブランチを探し，対象となるブランチに対して読み取り命令を送る（UnionFS も，unionfs_read 関数で同様の処理を行う）．読み取り命令を送る対象のファイルオブジェクト (target_file) は，uecfs_file_info 構造体の lower_files メンバから見つける．target_file へ読み取り命令を送った後に，target_file に対応する target_node を uecfs_list から取得し，ファイルアクセス回数のカウントを行う．ファイルアクセス回数は，カーネルスレッドにより，一定時間毎にリセットする．

3.3 ファイルの配置先変更条件

ファイルの配置先を変更する条件を満たすと，ファイル配置先を変更することを示すフラグ (uecfs_node 構造体の flag_move メンバ) を立てる．フラグが立っているファイルの配置先変更は，フラグを立ててすぐには行わず，ある特定のタイミングで行う．これにつ

```
static ssize_t uecfs_read( ... ) {
  /* 読み取り命令を送るブランチの
   * ファイルオブジェクト
   */
  struct file *target_file;

  /* target_file に対応するノード */
  struct uecfs_node *target_node;

  /* 読み取り命令を送るブランチが
   * HDD かどうかのフラグ
   */
  int flag_hdd;

  /* target_file を取得し，
   * flag_hdd を設定する
   */
  ...

  /* ブランチへ読み取り命令を送る */
  vfs_read(target_file);

  /* target_node を取得する */
  ...

  /* ファイルのアクセス回数をカウントする */
  target_node->access_count++;

  /* ファイルが HDD に配置されていて，
   * アクセス回数が閾値を超えた場合
   */
  if (flag_hdd &&
      target_node->access_count >
      threshold) {

    /* ファイルの配置先変更を
     * 示すフラグを立てる
     */
    target_node->flag_move = 1;
  }
}
```

図 7 uecfs_read 関数
Fig. 7 uecfs_read function

いては 3.4 節で説明する．

3.3.1 HDD から SSD への配置先変更条件

読み取り命令を送るファイルが HDD にあり，そのファイルへのアクセス回数が閾値を超えている場合に，配置先変更を示すフラグを立てる．閾値は，ユーザが設定できる．

3.3.2 SSD から HDD への配置先変更条件

SSD に配置されているファイルの配置先を変更する処理は，カーネルスレッドが行う．SSD に配置されているファイル数を常に監視しており，ファイル数が閾値を超えている場合に，配置先を変更するファイルを決定する処理を行う．全ファイルのアクセス回数を調べて，アクセス回数の平均値を計算し，アクセス回数が平均値を下回るファイルに対して，配置先変更を示すフラグを立てる．閾値は，ユーザが設定できる．

```
int uecfs_file_release( ... ) {
  /* 解放するファイルオブジェクトに
   * 対応するノード
   */
  struct uecfs_node *moving_node;

  /* moving_node を取得する */
  ...

  /* ファイルの配置先変更を
   * 示すフラグが立っている場合
   */
  if (moving_node->flag_move) {

    /* ファイルの配置先を変更する */
    ...
  }

  /* ファイルオブジェクトを解放する */
  ...
}
```

図 8 uecfs_file_release 関数
Fig. 8 uecfs_file_release function

3.4 ファイルの配置先を変更するタイミング

オープン中のファイルへの参照がなくなると，ファイルオブジェクトを解放するために，uecfs_release 関数（**図 8**）が呼び出される．この関数では，対象ファイルのファイルオブジェクト (uecfs_file_info 構造体の lower_files メンバ) を全て解放する (UnionFS も unionfs_release 関数で同様の処理を行う)．対象ファイルに対応する uecfs_node のフラグ (moving_node->flag_move) が立っている場合，そのファイルの配置先を変更する．

3.5 ファイルの作成

新規ファイルの作成は，SSD 上ではなく，常に HDD 上で行う．SSD には，書き込み回数の上限があり，HDD に比べて寿命が短い．新規作成ファイルは，将来多くアクセスされるかどうか分からず，SSD 上に作成しても，すぐに HDD 上へ配置される可能性があるため，最初は HDD 上に作成して SSD の書き込み回数を無駄に消費しないようにしている．

4. 実　　験

UECFS は，UnionFS 2.5.7 (for Linux Kernel 2.6.30.10) を拡張して実装した．これを用いて，ファ

表 1 I/O 性能
Table 1 I/O performance

	Size [MB]	Sequential Output			Sequential Input		Random
		Per Chr [KB/sec]	Block [KB/sec]	Rewrite [KB/sec]	Per Chr [KB/sec]	Block [KB/sec]	Seeks [B/sec]
SSD	6,560	53,160	77,258	55,966	59,300	216,870	4,915
HDD	6,560	50,264	55,729	31,765	57,430	71,969	150

イルアクセスの処理時間とオーバーヘッドを測定する実験を行った．実験環境として，Intel Core2 Quad 2.4GHz, メモリ 4GB, Intel SATA SSD 80GB, Hitachi SATA HDD 320GB を用いた．ページキャッシュの効果を減らすために，Linux のブートオプションを設定して，メモリを 256MB と認識するようにした．SSD と HDD は，両方とも Ext3 ファイルシステムでフォーマットした．SSD と HDD の I/O の基本性能を測定するために Bonnie++[2] を用いた．Bonnie++は，ファイルシステムの I/O 性能を測定するベンチマークソフトである．**表 1** に，結果を示す．Size は読み書きした総データサイズ，Per Chr はキャラクタベースの書き込み，Block はブロックベースの書き込み，Rewrite はブロックベースの再書き込みを表している．測定結果から，SSD は，HDD に比べて，I/O 性能が全体的に優れていることが分かる．特に，ブロックベース書き込みとランダムシークの性能が高く，ブロックベース書き込みは 3 倍，ランダムシークは 33 倍速い．

4.1 PostMark

実験に用いたプログラムは，PostMark[3] である．PostMark は，電子メール，ネットニュース，e-コマースなどのアプリケーションをエミュレートするために設計された，ベンチマークソフトである．PostMark が行う処理の大まかな流れは，次の通りである．

(1) 設定した数のファイルとディレクトリを作成する．
(2) 設定した数のトランザクションを実行する．1 トランザクションでは，ランダムに選んだファイルに対する読み書きを行う．設定で与えられた割合に基づいて，読み取りか書き込みのどちらかを行う．
(3) 作成したファイルとディレクトリを全て削除する．

本実験では，**表 2** のようにパラメータを設定した．

4.2 ファイルアクセスの処理時間

PostMark のソースコードを一部修正し，全トランザクションを均等な 10 区間に分けて，区間毎にかかった処理時間を計測した．UECFS を用いない SSD のみの場合，UECFS を用いない HDD のみの場合，UECFS を用いて SSD と HDD を重ねてマウントする場合の 3 通りについて計測した．UECFS を用いた実験では，HDD から SSD へファイルの配置先を変更する条件となる，ファイルアクセス回数の閾値を 4 回に設定した．また，SSD から HDD へファイルの配置先を変更する条件となる，SSD 上に存在するファイル数の閾値を 20,000 に設定した．

表 2 PostMark の設定内容
Table 2 Parameter settings for PostMark

ファイル数	40,000
ファイルサイズ [KB]	20
トランザクション数	400,000
読み取り命令数：書き込み命令数	5:1
ディレクトリ数	200

図 9 に結果を示す．区間毎の平均処理時間は，SSD のみの場合では 10 秒，HDD のみの場合では 264 秒となり，ともに区間毎で処理時間のばらつきは少ない．UECFS を用いた場合では，第 1 区間から第 4 区間にかけて，区間毎の処理時間が短くなっている．アクセス回数が閾値を超えた HDD 上のファイルが，SSD 上へ配置されることにより，ファイルアクセス速度が向上しているためであると考えられる．第 1 区間では，HDD のみの場合より，UECFS を用いた場合の処理時間が長いが，これは HDD から SSD へのファイル配置が起こることよって処理時間が増えているためであると考えられる．第 4 区間以降，処理時間が一定となっているのは，SSD 上に存在するファイル数が閾値に到達して，SSD から HDD へファイルが配置されることで，ファイルアクセス速度が向上しなくなったためであると考えられる．ファイルの総移動回数は，HDD から SSD へ 58,622 回，SSD から HDD へ 26,363 回であった．

4.3 オーバーヘッド

UECFS を用いない HDD のみの場合と，UECFS を用いて 2 つの HDD（HDD1 と HDD2）を重ねてマウントする場合で，全トランザクションにかかる処理時間を測定した．UECFS を用いた場合では，HDD1 から HDD2 へファイルの配置先を変更する条件とし

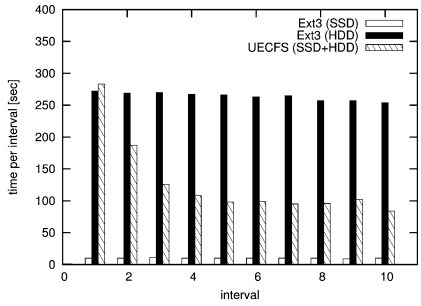


図 9 区間あたりの処理時間
Fig. 9 Processing time per interval

表 3 処理時間
Table 3 Execution times

ファイルシステム	閾値	処理時間 [秒]	ファイル移動数
Ext3	—	2,640	—
UECFS	4	3,348	39,731
UECFS	100	2,723	0

て，ファイルアクセス回数の閾値を変えて測定した．また，HDD2 上のファイル数が 40,000 になると，アクセス回数の少ないファイルを HDD2 から HDD1 へ配置するように設定した．

表 3 に結果を示す．UECFS（閾値 = 4）の場合，ほとんどのファイルが配置先を変更されており，ファイルコピーやファイル削除によるオーバーヘッドが 26.8%と大きくなった．UECFS（閾値 = 100）の場合，配置先が変更されるファイルがないため，オーバーヘッドが小さい．後者の結果から，ファイルアクセスにかかるオーバーヘッドが 3.1%と小さいことが分かる．

5. 関 連 研 究

FlashCache[4] と eNVy[5] は，金銭的コストを抑えつつ，システムのパフォーマンスを高めるために，SSD を DRAM と HDD の間に配置して，SSD をディスクキャッシュとして使う方法を提案している．UECFS では，SSD をディスクキャッシュとして利用せず，SSD のコントローラを扱う必要がないため，利用する SSD を限定しない点で，これらとは異なっている．ZFS[6] は，複数のディスクを仮想的に一つとして扱うことができるファイルシステムであり，一部のディスクをキャッシュとして利用可能であるため，SSD をキャッシュとして利用することができる．UECFS は，SSD と HDD に別々のファイルシステムをマウントできる点で異なっている．近年，JFFS2[7] や YAFFS[8] など，様々なフラッシュファイルシステムが登場しているが，UECFS は，これらのファイルシステムを，用途に合わせて自由に選んで SSD にマウントできる点で優れている．

SSD は，データの上書きができず，一度内容を消去してから書き込む必要があり，書き込み性能は，デバイスドライバやファイルシステムに大きく影響されることがある．また，データの書き込み回数が限られており，HDD に比べて寿命が短い．これらの特性を考慮して，SSD への書き込みを抑えつつ，SSD と HDD の両方を活かすシステムが提案されている．Koltsidas らの研究[9] では，読み取りの多いファイルを SSD に配置し，書き込みの多いファイルを HDD に配置することで，ファイルアクセスの高速化を実現している．Soundararajan らの研究[10] では，HDD を SSD のライトキャッシュとして利用することで，SSD の寿命を延ばしつつ，ファイルアクセス速度を向上する方法を提案している．UECFS では，アクセス回数の多いファイルのみを SSD に配置することで，SSD への書き込みを抑えている．Koltsidas らの研究は，ページ単位でディスクにデータを格納しているが，UECFS は，ファイル単位でディスクにデータを格納している点で異なっている．Soundararajan らの研究では，HDD への書き込み速度を向上するために，シーケンシャルライトが SSD 程度に速い HDD を利用し，HDD にログ構造でデータを格納することに限定している．UECFS では，高性能な HDD を用意する必要はなく，HDD のファイルシステムを自由に選べるため，ユーザの選択肢が多い．

6. お わ り に

本稿では，SSD と HDD を併用し，ファイルを自動配置することで，ファイルアクセスの高速化を実現するファイルシステム UECFS について述べた．UECFS により，SSD 上のディレクトリと，HDD 上のディレクトリを重ねてマウントすることで，SSD 上と HDD 上のファイルやディレクトリを，同じディレクトリに属しているように見せることができる．また，アクセス回数の多いファイルを SSD 上に，アクセス回数の少ないファイルを HDD 上に自動で配置することで，ファイルアクセスの高速化が可能である．UECFS は，UnionFS 2.5.7 (for Linux Kernel 2.6.30.10) を拡張することで実装した．PostMark を用いた実験を行った結果，アクセス回数の多いファイルが SSD へ自動配置されることで，平均的なファイルアクセス時間が速くなることを確認した．

実験から，HDD から SSD へのファイル配置によ

るオーバーヘッドが大きく，ファイルアクセスのオーバーヘッドが小さいことが分かった．現在は，HDD から SSD へファイルの配置先を変更する際に，ファイルをコピーしてすぐに HDD 上のファイルを削除しているが，ファイル削除を遅延することで，負荷を分散する方法などを検討する必要がある．

参 考 文 献

1) Quigley, D. P., Sipek, J., Wright, C. P. and Zadok, E.: UnionFS: User- and Community-oriented Development of a Unification Filesystem, *Proceedings of the 2006 Linux Symposium*, Vol.2, Ottawa, Canada, pp.349–362 (2006).
2) Coker, R.: Bonnie++, http://www.coker.com.au/bonnie++.
3) Katcher., J.: Postmark: a new file system benchmark., Technical report, Network Appliance, Inc. (1997).
4) Kgil, T. and Mudge, T.: FlashCache: a NAND flash memory file cache for low power web servers, *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ACM, p. 112 (2006).
5) Wu, M. and Zwaenepoel, W.: eNVy: a Non-Volatile main memory storage system, *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pp.116 –118 (1993).
6) Watanabe, S.: Solaris 10 ZFS Essentials (2010).
7) Woodhouse, D.: JFFS: The Journalling Flash File System, Ottawa Linux Symposium (2001).
8) One, A.: YAFFS: Yet another flash filing system, http://www.yaffs.net (2002).
9) Koltsidas, I. and Viglas, S.: Flashing up the storage layer, *Proceedings of the VLDB Endowment*, Vol.1, No.1, pp.514–525 (2008).
10) Soundararajan, G., Prabhakaran, V., Balakrishnan, M. and Wobber, T.: Extending SSD lifetimes with disk-based write caches, *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, Berkeley, CA, USA, USENIX Association, pp.8–8 (2010).