サンプルコード: 不揮発性メモリーキャッシュの実装 - 簡単な検索の例

この記事は、インテル® デベロッパー・ゾーンに公開されている「Code Sample: Implement a Persistent Memory Cache-A Simple Find Example」の日本語参考訳です。

ファイル:	ダウンロード (英語)
ライセンス:	3-Clause BSD License (英語)

動作環境	
OS:	Linux* カーネル 4.3 以上
ハードウェア:	インテル® Optane™ DC パーシステント・メモリーと 第 2 世代インテル® Xeon® スケーラブル・プロセッサー エミュレート: 「DRAM を使用して不揮発性メモリーをエミュレートする方法」 (英語) を参照
ソフトウェア: (プログラミング言語、ツール、IDE、フレームワーク)	C++ コンパイラー、不揮発性メモリー開発キット (PMDK) ライブラリー
必要条件:	C++ と PMDK の使用経験

はじめに

この記事では、不揮発性メモリーをキャッシュとして利用することで、ユーザー体験を向上する方法を示します。不揮発性メモリーを使用することでデータモデルを 1 カ所にまとめ、メインメモリーとストレージの間でデータを分割する必要がなくなります。

不揮発性メモリー向けのコードには、不揮発性メモリー開発キット (PMDK) (英語) に含まれる高レベルのトランザクション・ライブラリーである、libpmemobj ライブラリーの C++ バインディング (英語) を使用します。詳細は、インテル® デベロッパー・ゾーンの不揮発性メモリー・プログラミング (英語) を参照してください。導入に必要な情報を入手できます。

この記事で紹介するコードはすべてこちらのリポジトリー (英語) にあります。手順に従ってコンパイルしてください。

揮発性バージョン

このサンプルコードは、**find-all** と呼ばれる Linux* の **find** コマンドの簡易バージョンに対応しています。上記のリポジトリーにアクセスすると、このサンプルには **create_test_tree.sh** スクリプトが用意されているのが分かります。このスクリプトは、次の 3 つのパラメーターを使用して任意のディレクトリー構造を作成します。

tree-root: ディレクトリーのルートの位置。
tree-depth: ツリーに含まれるサブディレクトリーのレベルの数。
files-per-level: 各レベルで作成されるファイルの数。

作成されるファイルは空です (**find** コマンドは名前でファイルを検索するため、ここではファイル名のみ必要です)。各レベルのサブディレクトリーの数は 3 に固定されています。

例えば、ルートが **./test_tree** で、6 つのレベルがあり、各レベルに 100 ファイルがあるディレクトリー・ツリーを作成する場合、次のコマンドを実行します。

```
$ ./create_test_tree.sh test_tree 6 100
```

ルートをリストすると、次のような出力になります。

```
[sdp@localhost find_all_pm]$ ls test_tree/
acvlhpknswalwbwtnpzggaewqnacpopv  httjlfkrsbjvnydfwkfiekfqkbqtstkq
rgwbefyncedkyvpziijfqgzxcghcbyua
agocrfeblfzeirldzyzyqeqsqecwodio  hxuimxojjfprdmuoqxkeopavhjuldomd
rhafepwdyvoxoxhhfqyybajfhkijkryh
bdpdfdfysqkpilptzfqypldeydiemcel  idvcepykqyvwuthshsopckxkpuxqmdab
...
[sdp@localhost find_all_pm]$
```

ファイル名は 32 文字で、ランダムに生成されます。ディレクトリー名もランダムに生成されますが、5 文字です。ここでは、364 サブディレクトリーと 36,400 ファイルを生成しました。

これで、任意の文字列を含むファイル名を検索する準備ができました。ファイル名に「**aazz**」という文字列を含むファイルを検索します。

```
$ ./find-all ./test_tree aazz
./test_tree/gzqtl/kwntu/fvecf/tivvl/kuzpl/klwfgiwqenljruorfdupgkmaazzydrik
./test_tree/gzqtl/kwntu/opuci/ktxmi/chisz/enlvjaazzuanuguupnnrrwxgfafekwck
./test_tree/hfibr/fifpu/wtpdt/mgrex/mbrma/sapjhqtpnxaazzkflhbmgtrhqbcxkvuj
$
```

次のように、3 つのファイルが見つかりました。検索にかかった時間を確認します。

```
$ time ./find-all ./test_tree aazz
./test_tree/gzqtl/kwntu/fvecf/tivvl/kuzpl/klwfgiwqenljruorfdupgkmaazzydrik
./test_tree/gzqtl/kwntu/opuci/ktxmi/chisz/enlvjaazzuanuguupnnrrwxgfafekwck
./test_tree/hfibr/fifpu/wtpdt/mgrex/mbrma/sapjhqtpnxaazzkflhbmgtrhqbcxkvuj

real    0m0.579s
user    0m0.517s
sys     0m0.060s
```

* パフォーマンスに関する注意事項を参照してください。

約 0.5 秒です。わずかのように見えますが、検索クエリーを頻繁に実行する必要がある場合は数秒になる可能性があります。さらに、ツリーの構造が複雑になると (以下を参照)、応答時間が数秒になり、ユーザー体験が大きく損なわれる可能性があります。

不揮発性バージョン

このバージョンでは、各パターンのディレクトリー構造 (ツリー) を模倣する永続的なデータ構造が追加されます。データ構造は、ルート・オブジェクトで始まります。ルートには、ユーザーが以前に検索したパターン のリストがあります。

```
class root {
  private:
    pobj::persistent_ptr<pattern> patterns;

  public:
    pattern *find_pattern(const char *patstr, const char *rootstr) {
        ...
    }
    pattern *create_pattern(const char *patstr, const char *rootstr) {
        ...
    }
};
```

各パターンから、すべてのサブディレクトリーを格納するツリーを作成します。ファイルは、データ構造の効率を維持するため、以前のクエリーに一致するファイル名のみ格納します。

ツリー内の各サブディレクトリーには、最後に検索が実行された時間も格納されます。ユーザーが同じパターンを検索すると、プログラムは単純にこのインメモリー・ツリー構造を反復します。ファイルシステム内の特定のサブディレクトリーの変更時間が最後の検索から変わっていない場合 (変更時間は、サブディレクトリーに新しいファイルやディレクトリーが追加または削除された場合にのみ更新されます)、そのサブディレクトリーは再スキャンする必要がありません。単純にキャッシュされている結果を出力して、次のサブディレクトリーへ再帰的に移動できます。以下は、class pattern のコードです。

```
class pattern {
  private:
    pobj::persistent_ptr<char[]> patstr;
    pobj::persistent_ptr<char[]> rootstr;
    pobj::persistent_ptr<entry> rootdir;
    pobj::persistent_ptr<pattern> next;

  public:
    pattern(const char *patstr, const char *rootstr) {
        NEW_PM_STRING(this->patstr, patstr);
        NEW_PM_STRING(this->rootstr, rootstr);
        this->rootdir = pobj::make_persistent<entry>(nullptr, rootstr, true);
        this->next = nullptr;
    }
    const char *get_patstr(void) { return this->patstr.get(); }
    const char *get_rootstr(void) { return this->rootstr.get(); }
    pobj::persistent_ptr<pattern> get_next(void) { return this->next; }
    void set_next(pobj::persistent_ptr<pattern> pat) { this->next = pat; }
    int find_all(void) {
        return rootdir->process_directory(this->patstr.get());
    }
};
```

関数 **find_all()** は、パターンでツリールートをスキャンするために呼び出されます。この関数は、特定のディレクトリー・ルート以下 (最初のエントリーは常にツリーのルートであるため) のすべてのファイルとディレクトリーをスキャンする class **entry** から再帰関数 **process_directory()** を呼び出します。

エントリーは、ディレクトリーまたはファイルです。ファイルエントリー (以前の検索でキャッシュされた) は単純に結果として出力されます。一方、ディレクトリーは **process_directory()** を呼び出して再帰的に処理されます。前述のとおり、最後の検索からディレクトリーの変更時間が変わっている場合は、その内容をファイルシステムから再スキャンする必要があります。

```
        ...
        /* Let&apos;s get current &apos;last modif time&apos; */
        stat(path, &st);
        time_t new_mtime = st.st_mtime;
        if (difftime(new_mtime, this->mtime) != 0) {
            /* dir content has changed, we need
             * to re-scan it
             * */

            while ((dirp = readdir(dp)) != NULL) {
                ...
```

以下は class entry の変数です。

```
class entry {
  private:
    pobj::persistent_ptr<char[]> parent;
    pobj::persistent_ptr<char[]> name;
    pobj::p<bool> isdir;
    pobj::p<time_t> mtime;
    pobj::persistent_ptr<entry> entries;
    pobj::persistent_ptr<entry> next;
    ...
};
```

永続プログラムの名前は **find-all-pm** です。実行するには、各種パラメーターに加えて、不揮発性メモリープールの場所を指定する必要があります (プールが存在しない場合はプログラムによって作成されます)。ここでは、**/mnt/pmem/pool** にプールがあります。

```
$ time ./find-all-pm /mnt/pmem/pool ./test_tree aazz
./test_tree/hfibr/fifpu/wtpdt/mgrex/mbrma/sapjhqtpnxaazzkflhbmgtrhqbcxkvuj
./test_tree/gzqtl/kwntu/opuci/ktxmi/chisz/enlvjaazzuanuguupnnrrwxgfafekwck
./test_tree/gzqtl/kwntu/fvecf/tivvl/kuzpl/klwfgiwqenljruorfdupgkmaazzydrik

real    0m0.699s
user    0m0.020s
sys     0m0.038s
```

* パフォーマンスに関する注意事項を参照してください。

結果から、新しいパターンで初めて実行すると実行時間は長くなることが分かります。ここでは、揮発性バージョンと比較して 0.120 秒遅くなりました (0.579 秒と 0.699 秒)。しかし、再度実行すると、次のような結果になります。

```
$ time ./find-all-pm /mnt/pmem/pool ./test_tree aazz
./test_tree/hfibr/fifpu/wtpdt/mgrex/mbrma/sapjhqtpnxaazzkflhbmgtrhqbcxkvuj
./test_tree/gzqtl/kwntu/opuci/ktxmi/chisz/enlvjaazzuanuguupnnrrwxgfafekwck
./test_tree/gzqtl/kwntu/fvecf/tivvl/kuzpl/klwfgiwqenljruorfdupgkmaazzydrik

real    0m0.058s
user    0m0.020s
sys     0m0.038s
```

* パフォーマンスに関する注意事項を参照してください。

50 ミリ秒しかかかっていません。つまり、揮発性バージョンは 10 倍低速です。

大きなツリー

大きなツリーでテストしてみます。ファイルの数を 10 倍に増やしてみましょう。

```
$ ./create_test_tree.sh test_tree1 6 1000
```

揮発性バージョンを実行します。

```
$ time ./find-all test_tree1 aazz
test_tree1/npkhu/egzez/qyixsuqzfywemoaazzgdwququodezchy
test_tree1/npkhu/egzez/fpvee/sbkue/vupmb/xptjsqiqtuchcspywsjaazzxceuaokfa
test_tree1/npkhu/egzez/fpvee/zrrhb/agera/hcxhbztjmfmedzbytkgdwxeaazzygnnp
test_tree1/mgwwl/tsjzb/xuojd/wsjzw/iittr/icaazzzwzmdaevemdkjsybtegxccrjqq

...
real    0m5.531s
user    0m5.042s
sys     0m0.476s
```

* パフォーマンスに関する注意事項を参照してください。

5.5 秒で結果が見つかりました。不揮発性バージョンを実行します。

```
$ time ./find-all-pm /mnt/pmem/pool test_tree1 aazz
test_tree1/mgwwl/tsjzb/xuojd/wsjzw/iittr/icaazzzwzmdaevemdkjsybtegxccrjqq
test_tree1/mgwwl/tsjzb/xuojd/isplz/jlqje/gwalshqfqlopanbutlcduuaazznziwle
test_tree1/mgwwl/tsjzb/xuojd/isplz/fpcpaectvcipoaazzuhvfltrcrxrqvnz
test_tree1/mgwwl/tsjzb/thybd/elaga/uczjzwoywatubaazzcktnsmlfvgbxoal
...
real    0m5.568s
user    0m5.001s
sys     0m0.531s
```

* パフォーマンスに関する注意事項を参照してください。

揮発性バージョンとほぼ同じで、5.7 秒かかりました。これは、このプログラムの主なボトルネックが不揮発性メモリーへの書き込みではないことを示しています。むしろ、ファイルシステムからのファイルのメタデータの読み取りとパターンマッチングの可能性があります。

同じパターンを再度検索します。

```
$ time ./find-all-pm /mnt/pmem/pool test_tree1 aazz
test_tree1/mgwwl/tsjzb/xuojd/wsjzw/iittr/icaazzzwzmdaevemdkjsybtegxccrjqq
test_tree1/mgwwl/tsjzb/xuojd/isplz/jlqje/gwalshqfqlopanbutlcduuaazznziwle
test_tree1/mgwwl/tsjzb/xuojd/isplz/fpcpaectvcipoaazzuhvfltrcrxrqvnz
test_tree1/mgwwl/tsjzb/thybd/elaga/uczjzwoywatubaazzcktnsmlfvgbxoal
...
real    0m0.058s
user    0m0.022s
sys     0m0.036s
```

* パフォーマンスに関する注意事項を参照してください。

以前と同様に 58 ミリ秒になりました。相対的には、揮発性バージョンはおよそ 100 倍低速です。一般に、ツリーに含まれるファイルの数が多くなるにつれて、不揮発性メモリーキャッシュの使用によってもたらされる利点は大きくなります。

まとめ

この記事では、不揮発性メモリーをキャッシュとして利用することで、ユーザー体験を向上する方法を紹介しました。具体的には、不揮発性メモリーキャッシュを使用して、Linux* の **find** コマンドの簡易バージョンに対応する小さなサンプルコードのパフォーマンスを 36,400 ファイルでは 10 倍、364,000 ファイルでは 100 倍向上しました。この記事で紹介したコードはすべてこちらのリポジトリー (英語) にから入手できます。

法務上の注意書き

* パフォーマンス結果は 2019 年 3 月 1 日時点のテスト結果に基づいたものであり、公開されている利用可能なすべてのセキュリティー・アップデートが適用されていない可能性があります。詳細については、構成の開示を参照してください。絶対的なセキュリティーを提供できる製品はありません。性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。

システム構成: 2019 年 5 月 10 日時点のインテルによるテスト。1 ノード、2x インテル® Xeon® Platinum 8260 プロセッサー、Wolfpass プラットフォーム、合計メモリー 192GB (12 スロット/16GB/2667MT/秒 DDR4 RDIMM)、合計不揮発性メモリー 1.5TB (12 スロット/128GB/2667MT/秒) インテル® Optane™ DC パーシステント・メモリー・モジュール (DCPMM)、インテル® ハイパースレッディング・テクノロジー有効、ストレージ (ブート): 1x TB P4500、ucode: 0x400001C、OS: CentOS* Linux* 7.6、カーネル: 3.10.0-957.12.2.el7.x86_64

次の脆弱性に対するセキュリティー緩和策: CVE-2017-5753、CVE-2017-5715、CVE-2017-5754、CVE-2018-3640、CVE-2018-3639、CVE-2018-3615、CVE-2018-3620、CVE-2018-3646、CVE-2018-12126、CVE-2018-12130、CVE-2018-12127、CVE-2019-11091

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。