不揮発性メモリーのアウトオブオーダー解析にインテル® Inspector を使用する

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「Use Intel® Inspector for Persistent Memory Out-of-Order Store Analysis」の日本語参考訳です。

不揮発性メモリーへの直接アクセス

新しいインテル® Optane™ DC パーシステント・メモリーの最大の利点の 1 つは、直接メモリー・アクセス・モードです。プログラマーにとって、これは任意の不揮発性メモリーブロックをアプリケーションのアドレス空間にマップして、通常のバイトアドレス可能なメモリーのように使用できることを意味します。このメモリーへのアクセスは、システム I/O スタックを使用するよりも高速であると予想され、格納されるデータはすべて永続化されます。例えば、長時間実行される計算では、その領域にデータ配列を割り当てることで、中断後も以前のデータがすべて保持されているため、計算を続行することができます。

不揮発性メモリーを使用する場合、開発者はデータ構造の一貫性を考慮する必要があるため、プログラミングの課題は増えます。中断はどこでも発生する可能性があるため、プログラムは常にデータを有効な状態に保つべきです。

サンプル・アプリケーション

最初のコードは、後で読み取るため、レコードを不揮発性メモリーに継続的に格納します。構造体のサイズは固定で、特定のレコードが有効かどうかを示すフラグがあります。プログラムは、データをブロックに格納するか、後で変更するためブロックを予約します。レコード数は可変で、値を取得するためブロック全体を読み取る際に時間がかかる大きな数になる可能性があります。そのため、格納されているレコード数を含むヘッダーブロックが追加されています。

```
struct header_t
{
    uint32_t counter;
    uint8_t reserved[60];
};

struct record_t
{
    char name[63];
    char valid;
};
```

以下は '格納' コードです。

```
for (int i = 0; i < RecordsToWrite; i++)
{
    // Store number of records
    header->counter++;

    if(rand() % 2 == 0)
    {
        //Store valid record
        snprintf(records[i].name, sizeof(records[i].name),
```

```
                "record #%u", i + 1);
        _mm_clflush(records[i].name);

        records[i].valid = true;
    }
    else
    {
        // Store empty record
        records[i].valid = false;
    }
    _mm_clflush(&records[i].valid);
}
```

アプリケーションは、フォールトトレラントで、予期しない中断 (プロセス終了や電源障害など) が発生してもデータ構造を破損しないようにする必要があります。そのためには、一般に、あらゆる実行ステージでデータ構造の一貫性を損なわないようにすべきです。

2 つ目のコードは、メモリーを読み取り、データファイルからすべての有効なレコードを出力します。

```
for (uint32_t i = 0; i < header->counter; i++)
{
        // If record is valid, print it to console
        if (records[i].valid)
        {
                std::cout << "found valid record:\n";
                std::cout << "  name    = " << records[i].name << "\n";
        }
}
```

このアプリケーションを実行すると、期待通りの結果が得られます。すべて問題ないように見えますが、インテル® Inspector の不揮発性インスペクター・ツールを使用して、潜在的な不揮発性メモリーのプログラミング・エラーの有無を確認できます。

アプリケーションの正当性チェック

最初の実行では、アプリケーションがどのようにデータを不揮発性メモリーに格納しているか調べます。次のコマンドを実行します。

```
pmeminsp cb -- out_of_order write
```

次のステップでは、アプリケーションがどのようにデータを読み取っているか調べます。

```
pmeminsp ca -- out_of_order read
```

データを収集したら、データの一貫性に関するアプリケーションの正当性の解析を行います。アウトオブオーダー・ストア解析モードでレポートを生成するには、次のコマンドを実行します。

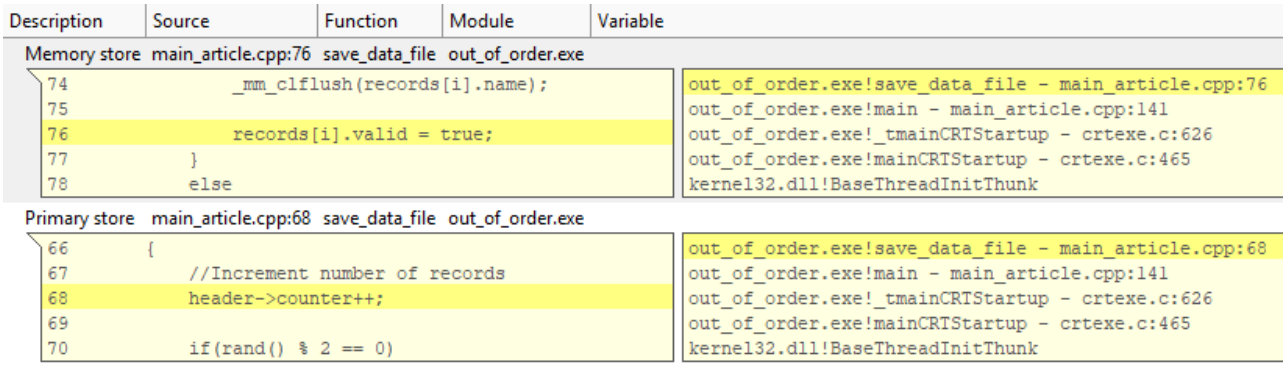
```
pmeminsp rp -check-out-of-order-store -insp -- out_of_order
```

収集されたデータは、現在のディレクトリー以下の '.pmeminspdata' サブフォルダーに格納されます。

インテル® Inspector のグラフィカル・ユーザー・インターフェイス (UI) でデータを表示するには、次のコマンドを実行します。

```
inspxe-gui ./pmeminspdata
```

いくつかの問題が報告されています。



インテル® Inspector は、レコードの 'valid' フラグと合計レコード数カウンターの格納順序が正しくないことを報告しています。カウンターをインクリメントしてから 'valid' フラグを設定するまでの間にアプリケーションが終了した場合、これは問題となります。有効なデータがレコードに格納される前にカウンターがインクリメントされるため、カウンターはインクリメントされても、新しいレコードは不揮発性メモリーで初期化されません。以降のそのブロックの読み取りは未定義の動作になります。

インテル® Inspector の UI は、同様の問題を同じグループにまとめて、“header->counter” の格納順序が 'name' メンバーの異なる部分で正しくないことを示しています。しかし、'name' メンバーは、現在のレコードが初期化された後に設定される 'valid' フラグに依存するため、'name' メンバーに関する順序は無視できます。

一貫性のないデータの修正

この問題を修正するには、オリジナルのソースを変更する必要があります。

```
for (int i = 0; i < RecordsToWrite; i++)
{
    if(rand() % 2 == 0)
    {
        //Store valid record
        snprintf(records[i].name, sizeof(records[i].name),
                 "record #%u", i + 1);
        _mm_clflush(records[i].name);

        records[i].valid = true;
    }
    else
    {
        //Store empty record
        records[i].valid = false;
    }
    _mm_clflush(&records[i].valid);

    //Increment number of records
    header->counter++;
}
```

最初にレコードの内容を格納して、それが確実に不揮発性メモリーに格納されるようにフラッシュします。その後でのみ、合計レコード数カウンターを安全にインクリメントできます。この順序を使用することで、初期化されていないレコードや部分的に初期化されたレコードが読み取りコードによって読み取られるのを防ぎます。

次に、このアプリケーションを再コンパイルして、ほかの問題がないか確認するためメモリーストア解析を再度実行します。次のコマンドを再度実行します。

```
pmeminsp cb -- out_of_order write
```

そして、ここでは問題レポートをコンソールに出力します。

```
pmeminsp rp -check-out-of-order-store -- out_of_order
```

アプリケーションの問題は 2 つだけになりました。1 つ目の問題は、'counter' メンバーの最初の初期化に関するものです。もう 1 つは、'name' メンバーへのシンボルの格納順序を逆にすることを提案しています。試してみる価値があるかもしれませんが、'name' データブロック全体が 'valid' フラグに依存しているため、安全に無視できます。

```
#=============================================================================
# Diagnostic # 1: Out-of-order stores
#-------------------
  Memory store
    of size 4 at address 0x20AA9230000 (offset 0x0 in d:\testapp\outoforder.dat)
    in d:\testapp\out_of_order.exe!save_data_file_fixed at main_article.cpp:110 - 0x13D8
    in d:\testapp\out_of_order.exe!main at main_article.cpp:190 - 0x1886

  is out of order with respect to

  memory store
    of size 1 at address 0x20AA923007F (offset 0x7F in d:\testapp\outoforder.dat)
    in d:\testapp\out_of_order.exe!save_data_file_fixed at main_article.cpp:126 - 0x14A0
    in d:\testapp\out_of_order.exe!main at main_article.cpp:190 - 0x1886


#=============================================================================
# Diagnostic # 2: Out-of-order stores
#-------------------
  Memory store
    of size 1 at address 0x20AA9230080 (offset 0x80 in d:\testapp\outoforder.dat)
    in c:\windows\system32\msvcr120d.dll!mbctombb_l at <unknown_file>:<unknown_line> -
0xC4F20
    in c:\windows\system32\msvcr120d.dll!mbctombb_l at <unknown_file>:<unknown_line> -
0xC3C93
    in c:\windows\system32\msvcr120d.dll!snprintf at <unknown_file>:<unknown_line> -
0x2D05B
    in d:\testapp\out_of_order.exe!save_data_file_fixed at main_article.cpp:118 - 0x144D
    in d:\testapp\out_of_order.exe!main at main_article.cpp:190 - 0x1886

  is out of order with respect to

  memory store
    of size 8 at address 0x20AA9230088 (offset 0x88 in d:\testapp\outoforder.dat)
    in c:\windows\system32\msvcr120d.dll!mbctombb_l at <unknown_file>:<unknown_line> -
0xC4F20
    in c:\windows\system32\msvcr120d.dll!mbctombb_l at <unknown_file>:<unknown_line> -
0xC5083
    in c:\windows\system32\msvcr120d.dll!mbctombb_l at <unknown_file>:<unknown_line> -
0xC4C38
    in c:\windows\system32\msvcr120d.dll!snprintf at <unknown_file>:<unknown_line> -
0x2D05B
    in d:\testapp\out_of_order.exe!save_data_file_fixed at main_article.cpp:118 - 0x144D
    in d:\testapp\out_of_order.exe!main at main_article.cpp:190 - 0x1886
```

まとめ

不揮発性メモリーは、さまざまなクラスのアプリケーションを次のレベルへ引き上げることができる非常に強力なテクノロジーですが、この新しいテクロジーは新たな課題をもたらし、コードの正当性にさらなる注意が必要です。ここで紹介した例は、インテル® Inspector の不揮発性インスペクター・ツールを使用してアプリケーションを確認し、散発的な不具合を引き起こす問題を修正する方法を示しました。

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。