インテル® Inspector の不揮発性インスペクターにより不揮発性メモリーのプログラミング・エラーを検出する方法

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「How to Detect Persistent Memory Programming Errors Using Intel® Inspector - Persistence Inspector」の日本語参考訳です。

概要

不揮発性メモリーは、アプリケーションのパフォーマンスと信頼性を向上させる大きな可能性がある、新しいメモリー・ストレージ・テクノロジーです。ただし、コードのパフォーマンスを最大限に引き出すため、開発者はいくつかのプログラミングの課題に対処する必要があります。その 1 つは、不揮発性メモリーへのストアはキャッシングにより直ちに永 売化されないことです。データは、キャッシュ階層外になり、メモリーシステムに認識された後にのみ永 売化されます。プロセッサーのアウトオブオーダー実行とキャッシングにより、永 売化の順序はストアの順序と異なる可能性があります。

現在テクノロジー・プレビューとして提供されている**インテル® Inspector の不揮発性インスペクター** (英語) は、開発者が不揮発性メモリープログラムにおいてこれらのプログラミング・エラーを検出するのに役立つ新しいランタイムツールです。キャッシュ・フラッシュ・ミスに加えて、このツールは次の問題を検出します。

- 冗長なキャッシュフラッシュとメモリーフェンス
- アウトオブオーダーの不揮発性メモリーストア
- 不揮発性メモリー開発キット (PMDK) の不正な取り消しログ

この記事では、インテル® Inspector の不揮発性インスペクターの機能を説明し、導入に役立つ情報を提供します。

背景情報

インテルと Micron* により開発された 3D XPoint™ メモリーメディアなどの新しいテクノロジーを採用した不揮発性メモリーデバイスは、直接メモリー・コントローラーに装着できます。そのようなデバイスは、通常、不揮発性デュアル・インライン・メモリー・モジュール (NVDIMM) と呼ばれます。NVDIMM にあるデータは、バイトアドレス指定可能で、システムやプログラムがクラッシュしても保持されます。NVDIMM のアクセス・レイテンシーは、DRAM に匹敵します。プログラムは、通常の CPU ロード/ストア命令を使用して NVDIMM を読み書きします。次のコード例について考えてみます。

例 1: アドレス帳を不揮発性メモリーに書き込む

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <string.h>
```

```
struct address {
    char name[64];
    char address[64];
    int valid;
}

int main()
{
    struct address *head = NULL;
    int fd;
    fd = open("addressbook.pmem", O_CREAT|O_RDWR, 0666);
    posix_fallocate(fd, 0, sizeof(struct address));

    head = (struct address *)mmap(NULL, sizeof(struct address),
PROT_READ|PROT_WRITE,
      MAP_SHARED, fd, 0);
    close(fd);

    strcpy(head->name, "Clark Kent");
    strcpy(head->address, "344 Clinton St, Metropolis, DC 95308");
    head->valid = 1;

    munmap(head, sizeof(struct address));

    return 0;
}
```

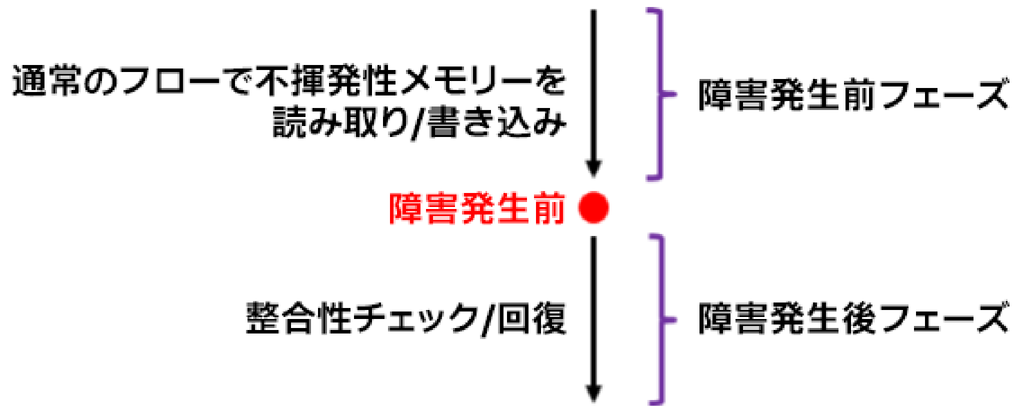
例 1 では、不揮発性メモリーはファイル `addressbook.pmem` として公開され、通常のファイルシステム API を使用してプロセスアドレス空間にマップされます。不揮発性メモリーがマップされると、プログラムは `strcpy` を呼び出して直接メモリーにアクセスします。`munmap` を呼び出す前に電力が失われた場合、不揮発性メモリーで次のいずれかのシナリオが発生します。

- `head->name`、`head->address`、および `head->valid` のいずれも、メモリーシステムに到達せず、永続化されません。
- 3 つすべてが永続化されます。
- 1 つまたは 2 つのみが永続化されます。

キャッシング効果は、不揮発性メモリー・ソフトウェア開発に課題を提示します。停電やシステムクラッシュ後にデータが回復可能で一貫性が保証されるように、開発者はいつどこでキャッシュ階層からメモリーシステムへ明示的にデータをフラッシュするか判断する必要があります。

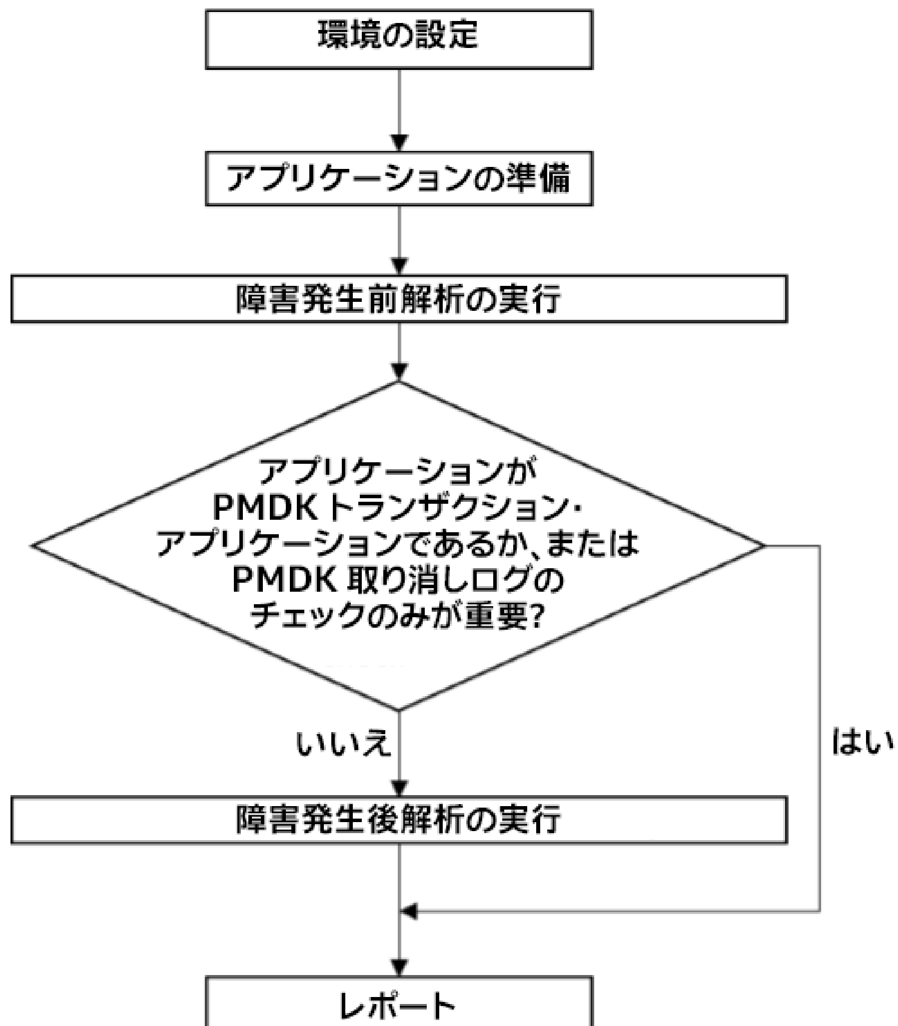
不揮発性メモリー・アプリケーションの動作

停電やシステムクラッシュなどの障害の後に再起動すると、不揮発性メモリー・アプリケーションは整合性を検証し、メモリーに格納されているデータを回復する必要があります。通常、障害によって不揮発性メモリー・アプリケーションは、障害発生前の実行と、障害発生後の実行の 2 つのフェーズに分けることができます。これにより、データが破損したり、矛盾が生じる可能性があります。障害発生前フェーズでは、アプリケーションは通常のフローを実行し、不揮発性メモリーを読み書きします。障害発生後にデータの一貫性をチェックして、アプリケーションが通常の操作を再開する前に、矛盾するデータを一貫性のある状態に回復します。



インテル® Inspector の不揮発性インスペクターの使用法

以下は、インテル® Inspector の不揮発性インスペクターの使用ワークフローです。



環境の設定

指定したディレクトリー (例えば /home/joe/pmeminsp) にツールのファイルがインストールされたら、PATH と LD_LIBRARY_PATH 環境変数にインテル® Inspector の不揮発性インスペクターのパスを追加します。以下に例を示します。

```
$ export PATH=/home/joe/pmeminsp/bin64:$PATH
```

```
$ export LD_LIBRARY_PATH=/home/joe/pmeminsp/lib64:$LD_LIBRARY_PATH
```

ツールがインストールされ、正しく設定されていることを確認するには、"pmeminsp" と入力します。

```
$ pmeminsp Type 'pmeminsp help' for usage.
```

アプリケーションの準備

前述のとおり、不揮発性メモリー・アプリケーションは通常 2 つのフェーズで構成されます。インテル® Inspector の不揮発性インスペクターを使用するには、これらの 2 つのフェーズのコードを識別する必要があります。

- **障害発生前**。ツールでチェックするコードを実行します。
- **障害発生後**。停電やシステムクラッシュの後に実行するコードを実行します。

アプリケーションが不揮発性メモリー開発キット (PMDK) (英語) を使用してトランザクションの不揮発性メモリーサポートを実装している場合、PMDK トランザクション・ランタイム・サポートがトランザクション・ブロック内のデータの一貫性と回復の責任を負います。障害発生後のコードは、PMDK トランザクション・ランタイムに配置されるため、フェーズを確認するためインテル® Inspector の不揮発性インスペクターを使用する必要はありません。

インテル® Inspector の不揮発性インスペクターに障害発生後フェーズを通知する

障害発生後フェーズを識別したら、解析時間を大幅に短縮するため、そのフェーズの開始と停止位置をツールに知らせることを強く推奨します。インテル® Inspector の不揮発性インスペクターには、アプリケーションが障害発生後フェーズ解析の開始と停止をランタイムにツールに通知するための API セットがあります。

```
#define PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT 0x2
void __pmeminsp_start(unsigned int phase);
void __pmeminsp_pause(unsigned int phase);
void __pmeminsp_resume(unsigned int phase);
void __pmeminsp_stop(unsigned int phase);
```

障害発生後フェーズの開始をツールに通知するには、フェーズの開始位置の直前に __pmeminsp_start (PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT) を呼び出します。同様に、障害発生後フェーズの停止をツールに通知するには、フェーズの終了位置の直後に __pmeminsp_stop(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT) を呼び出します。

__pmeminsp_pause(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT) と __pmeminsp_resume (PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT) 呼び出しを使用することで、解析が開始された後、停止される前に、解析の一時停止と再開を細かく制御できます。

例えば、障害発生後フェーズが関数 `recover()` 呼び出しの期間である場合、単純に関数の入口に `__pmeminsp_start(PMEMINSP_PHASE_AFTER_UNFORTUNATE_ EVENT)` を配置し、関数の出口に `__pmeminsp_stop(PMEMINSP_PHASE_AFTER_ UNFORTUNATE_EVENT)` を配置します。

例 2: インテル® Inspector の不揮発性インスペクターに障害発生後フェーズの開始と停止を通知する

```
#include "pmeminsp.h"
…
… …

void recover(void)
{
    __pmeminsp_start(PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT);
    …
    __pmeminsp_stop((PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT);
}

void main()
{
    …
    … = mmap(…..);
    __pmeminsp_stop((PMEMINSP_PHASE_AFTER_UNFORTUNATE_EVENT);
    …
    recover();
}
```

インテル® Inspector の不揮発性インスペクターの API は、`libpmeminsp.so` で定義されています。アプリケーションをビルドする際に、正しいオプションを指定します。以下に例を示します。

```
-I /home/joe/pmeminsp/include –L /home/joe/pmeminsp/lib64 –lpmeminsp.
```

障害発生前フェーズを解析する

次のコマンドは、障害発生前フェーズ解析を実行します。

```
pmeminsp check-before-unfortunate-event [options] --
```

アプリケーションがシステム API を使用して永 続メモリーファイルを直接マップする場合、(アプリケーション実行前にそのファイルが存在しない場合であっても) `-pmem-file` オプションを使用してそのファイルへのパスも指定する必要があります。以下に例を示します。

```
pmeminsp check-before-unfortunate-event -pmem-file ./addressbook.pmem [options] --
```

アプリケーションが複数のファイルを作成する場合、それらのファイルが含まれるフォルダーのパスを指定することも可能です。その場所から mmap されたファイルはすべて不揮発性メモリーファイルとして解釈されます。

アプリケーションが PMDK ベースの場合、これらの 2 つのオプションは冗長です。インテル® Inspector の不揮発性インスペクターは、これらのオプションが指定されていない場合であっても、PMDK で管理されるすべての不揮発性メモリーファイルを自動的にトレースします。

障害発生後フェーズを解析する

次のコマンドは、障害発生後フェーズ解析を実行します。

```
pmeminsp check-after-unfortunate-event [options] --
```

アプリケーションが PMDK を使用して不揮発性メモリーを操作する場合を除き、不揮発性メモリーファイルの場所が指定されていることを確認します。オプション名は、`check-before-unfortunate-event` コマンドの場合と似ています。

検出された問題をレポートする

検出された不揮発性メモリーの問題のレポートを生成するには、次のコマンドを実行します。

```
pmeminsp report [option] --
```

次に、インテル® Inspector の不揮発性インスペクターによって生成されるいくつかの診断例を示します。

キャッシュフラッシュの不足

不揮発性メモリーストア (最初のストア) のキャッシュフラッシュの不足は、常に後続の不揮発性メモリーストア (2 つ目のストア) に関連しています。その潜在的な悪影響は、2 つ目のストアの後に障害が発生した場合、2 つ目のストアは永続化されますが、最初のストアは永続化されないことです。

最初のメモリーストアを以下に示します。

```
in /home/joe/pmeminsp/addressbook/writeaddressbook!main at writeaddressbook.c:24
- 0x6ED,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main, at: - 0x21F43
in /home/joe/pmeminsp/addressbook/writeaddressbook!_start at: - 0x594
```

これは、次に示す 2 つ目のメモリーストアの前にフラッシュされません。

```
in /home/joe/pmeminsp/addressbook/writeaddressbook!main at: writeaddressbook.c:26
- 0x73F,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main, at: - 0x21F43,
in /home/joe/pmeminsp/addressbook/writeaddressbook!_start at: - 0x594
```

最初のストアの場所からのメモリーロードを以下に示します。

```
in /lib/x86_64-linux-gnu/libc.so.6!strlen at: - 0x889DA
```

これは、次に示す 2 つ目のストアの場所からのメモリーロードに依存します。

```
in /home/joe/pmeminsp/addressbook/readaddressbook!main at readaddressbook.c:22 -
0x6B0
```

冗長または不要なキャッシュフラッシュ

冗長または不要なキャッシュフラッシュとは、プログラムの正当性に影響することなく、解析した実行パスから排除できるものです。冗長または不要なキャッシュフラッシュは、プログラムの正当性には影響しませんが、プログラムのパフォーマンスに影響する可能性があります。

キャッシュフラッシュを以下に示します。

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_redundant_flush at
main.cpp:134 - 0x1721,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:52
- 0x151F,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:48
- 0x14FD,
in /home/joe/pmeminsp//tests/pmemdemo/src/pemmdemo!main at main.cpp:231 - 0x1C74,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main, at: - 0x21F43,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at: - 0x12A4
```

これは、次に示すキャッシュフラッシュと冗長です。

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_redundant_flush at
main.cpp:135 - 0x1732,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:52
- 0x151F,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:48
- 0x14FD,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:231 - 0x1C7,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at: - 0x21F43,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at: - 0x12A4
```

2 つ目のキャッシュフラッシュは、次のメモリーストアに関するものです。

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_redundant_flush at
main.cpp:133 - 0x170F,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:52
- 0x151F,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!create_data_file at main.cpp:48
- 0x14FD,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:231 - 0x1C74,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at: - 0x21F43,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at: - 0x12A4
```

アウトオブオーダーの不揮発性メモリーストア

アウトオブオーダーの不揮発性メモリーストアとは、正しい永 続化の順序を強制できない 2 つのストアです。明示的なキャッシュフラッシュでも正しい順序が強制されません。
アウトオブオーダーの不揮発性メモリーストアは、'report' コマンドの -check-out-of-order-store オプションで有効になります。

メモリーストアを以下に示します。

```
in /home/joe/pmemcheck/mytest/writename6!writename at writename6.c:13 - 0x6E0,
in /home/joe/pmemcheck/mytest/writename6!main at writename6.c:21 - 0x72D,
in /home/joe/pmemcheck/mytest/writename6!main at writename6.c:20 - 0x721,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at: - 0x21F43,
in /home/joe/pmemcheck/mytest/writename6!_start at: - 0x594
```

これは、次に示すメモリーストアの後に配置されるべきです。

```
in /home/joe/pmemcheck/mytest/writename6!writename at writename6.c:14 - 0x6EB,
in /home/joe/pmemcheck/mytest/writename6!main at writename6.c:21 - 0x72D,
in /home/joe/pmemcheck/mytest/writename6!main at writename6.c:20 - 0x721,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at: - 0x21F43,
in /home/joe/pmemcheck/mytest/writename6!_start at: - 0x594
```

取り消しログがない更新

PMDK トランザクションでメモリー位置が更新される前にそのメモリー位置の取り消しログを作成するのは開発者の責任です。通常、開発者は PMDK 関数の `pmemobj_tx_add_range()` または `pmemobj_tx_add_range_direct()` を呼び出すか、`TX_ADD()` マクロを使用してメモリー位置の取り消しログを作成します。メモリー位置が更新される前に取り消しログを作成しないと、トランザクションがコミットされなかった場合に PMDK はメモリー位置に対する変更のロールバックに失敗します。

取り消しログがない更新の問題は、メモリーストアとメモリーが更新されるトランザクションに関連してレポートされます。

メモリーストアを以下に示します。

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_tx_without_undo at
main.cpp:190 - 0x1963,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:175 -
0x1877,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:163 -
0x180D,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:245 - 0x1CF4,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at: - 0x21F43,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at: - 0x12A4
```

これは、次のトランザクションで取り消しログが作成されません。

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_tx_without_undo at
main.cpp:185 - 0x1921,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:175 -
0x1877,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:163 -
0x180D,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:245 - 0x1CF4,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at: - 0x21F43,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at: - 0x12
```

更新がない取り消しログ

PMDK トランザクションでメモリー位置が更新される前にそのメモリー位置の取り消しログを作成するのは開発者の責任です。通常、開発者は PMDK 関数の `pmemobj_tx_add_range()` または `pmemobj_tx_add_range_direct()` を呼び出すか、`TX_ADD()` マクロを使用してメモリー位置の取り消しログを作成します。メモリー位置の取り消しログがあっても、PMDK トランザクション内で更新されない場合、パフォーマンスの低下を招いたり、トランザクションがコミットされない場合にメモリーが `dirty/uncommitted/stale` 値にロールバックされる可能性があります。

更新がない取り消しログの問題は、PMDK の取り消しログ呼び出しとメモリーの取り消しログが作成されるトランザクションに関連してレポートされます。

メモリー領域には以下の取り消しログがあります。

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!test_tx_without_update  at
main.cpp:190 - 0x1963,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:175 -
0x1877,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:163 -
0x180D,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:245 - 0x1CF4,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at: - 0x21F43,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at: - 0x12A4
```

しかし、トランザクションで更新されません。

```
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo! test_tx_without_update at
main.cpp:185 - 0x1921,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:175 -
0x1877,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!run_tx_test at main.cpp:163 -
0x180D,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!main at main.cpp:245 - 0x1CF4,
in /lib/x86_64-linux-gnu/libc.so.6!__libc_start_main at: - 0x21F43,
in /home/joe/pmeminsp/tests/pmemdemo/src/pemmdemo!_start at: - 0x12A4.
```

まとめ

不揮発性メモリーは、魅力的な新しいテクノロジーです。この記事で説明したように、このテクノロジーにはいくつかのプログラミングの課題があります。これらの課題は、プログラム・ライフサイクルの早期に問題を発見し、非常に大きな投資対効果が得られる、インテル® Inspector の不揮発性インスペクターなどのツールを使用することで緩和できます。

次のステップ

ベータ版登録サイト (英語) からインテル® Inspector の不揮発性インスペクターのベータプログラムに参加して、テクノロジー・プレビューをダウンロードできます。

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。