インテル® VTune™ パフォーマンス・アナライザー Linux* 版

入門ガイド

インテル® VTune™ パフォーマンス・アナライザーは、コードのパフォーマンスに関する情報を提供します。パフォーマンス問題を識別して、チューニングの焦点を置くべき箇所を示し、短時間で最高のパフォーマンスを引き出します。

本ガイドでは、VTune アナライザーの基本的な機能を紹介します。

VTune アナライザーを使用してコードを解析し、最高のパフォーマンスを実現するためにはどこに焦点を置くべきか理解することを目的としています。

本ガイドでは、サンプル・アプリケーションのチューニングをとおして、パフォーマンス・チューニングを段階ごとに説明します。

- パフォーマンス問題の特定
- 問題解決のためのコード修正
- コードの修正前と修正後のパフォーマンス比較

目次

著作権/法律に基づく表示

1 アプリケーションのビルド

最初に、VTune アナライザーが有用なデータを収集できるような設定でアプリケーションをビルドする必要があります。製品レベルの最適化とシンボル情報を有効にするコンパイラー・オプションを使用してアプリケーションをビルドすると、(リリース時のように) 完全に最適化され、シンボル情報が含まれたコードが生成されます。

本ガイドで使用する例では、GNU* C コンパイラー (バージョン 3.2.3) を `'-g -O2'` オプションで使用しています。

注意: システムの構成やコンパイラーのバージョンによって、結果は異なります。

アプリケーションのビルド:

1. cd コマンドを使用して、/opt/intel/vtune/samples/gsexample ディレクトリーに移動します。
2. `make clean` を実行し、次に `make` と入力します。
   または、インテル® コンパイラーを使用している場合は、次のコマンドを入力します。
   `make CC=icc CFLAGS="-g -O2"`

2 アプリケーションの解析

アプリケーションのビルド後、`samples/gsexample` ディレクトリーにあるコードを使用して、パフォーマンスの解析を行います。VTune アナライザーは、データコレクターを使用して異なるタイプのパフォーマンス・データを収集します。このステップでは、**[First Use Wizard (簡易ウィザード)]** を使用して、データの収集、 結果の表示、ソースコードの特定の問題箇所へのズームインを行います。このウィザードは、システム全体に渡るパフォーマンス・データを収集し、アプリケーションにおける最もアクティブな 5 つの関数の基本データを提供します。

2.1 ベンチマークの作成

オリジナル・パフォーマンスのベンチマークの作成:

1. 次のコマンドを入力して gsexample2a アプリケーションを実行します。

#. /gsexample2a datafile.txt

説明:

`datafile.txt` はデータファイルです。

2. アプリケーションを実行後、次のようにコマンドライン・コンソールで経過時間を確認できます。

```
**************************************
* Elapsed time was 10 seconds
* CRC calculated: 0x5
* File character count: 10126
* File processed 113918 times
* Total characters counted: 1152356352
*
* Average characters/sec: 1.1524e+08
* Average loop iterations/sec: 11391.80
**************************************
```

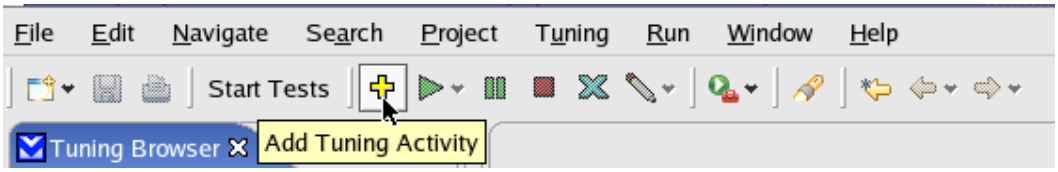
この経過時間は、アプリケーションのチューニングにおける現段階のベンチマークです。

ベンチマークは、将来修正を比較する場合の基準となるように、計測可能かつ再現可能でなければなりません。

2.2 ウィザードを使用するデータ収集

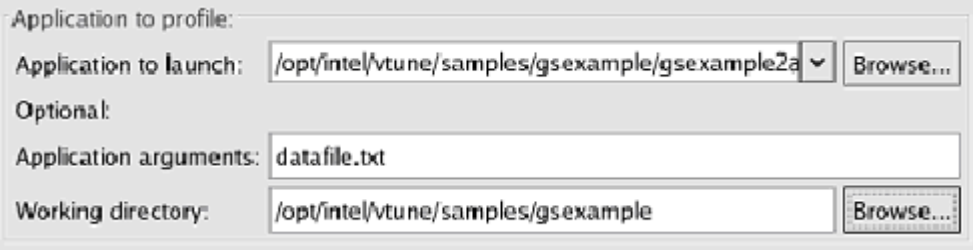
このステップでは、**[First Use Wizard (簡易ウィザード)]** を使用して、*アクティビティー*を作成し実行します。アクティビティーは、パフォーマンス・データを収集して、VTune アナライザーのプロジェクトに保存します。

1. コマンドラインで、VTune アナライザーのインストール・ディレクトリーに移動し、`vtlec` を実行して VTune アナライザーを起動します。次のコマンドを入力します。
   `# /opt/intel/vtune/bin/vtlec`
   VTune アナライザーが起動して、スタートアップ画面と **[Select a Workspace (ワークスペースの選択)]** ダイアログボックスが表示されます。
2. **[Select a Workspace (ワークスペースの選択)]** ダイアログボックスで、**[OK]** をクリックしてデフォルトの設定を使用します。
3. 次のように、VTune アナライザーのツールバーで、**[Add Tuning Activity (チューニング・アクティビティーの追加)]** をクリックして、**[Select a Wizard (ウィザードの選択)]** ダイアログボックスを開きます。



4. **[First Use Wizard (簡易ウィザード)]** () を選択して、**[Next (次へ)]** ボタンをクリックします。

5. **[Application to Launch (起動するアプリケーション)]** フィールドで **[Browse (参照)]** をクリックして、gsexample2a アプリケーションを参照します。ダイアログボックスが表示されます。
6. `/opt/intel/vtune/samples/gsexample` ディレクトリーにある gsexample2a ファイルの場所を参照して、**[OK]** をクリックします。
7. **[Application Arguments (アプリケーションの引数)]** フィールドに、データファイルの名前 (`datafile.txt`) を入力します。
8. **[Working Directory (作業ディレクトリー)]** フィールドで、**[Browse (参照)]** をクリックして `gsexample` ディレクトリーを参照します。

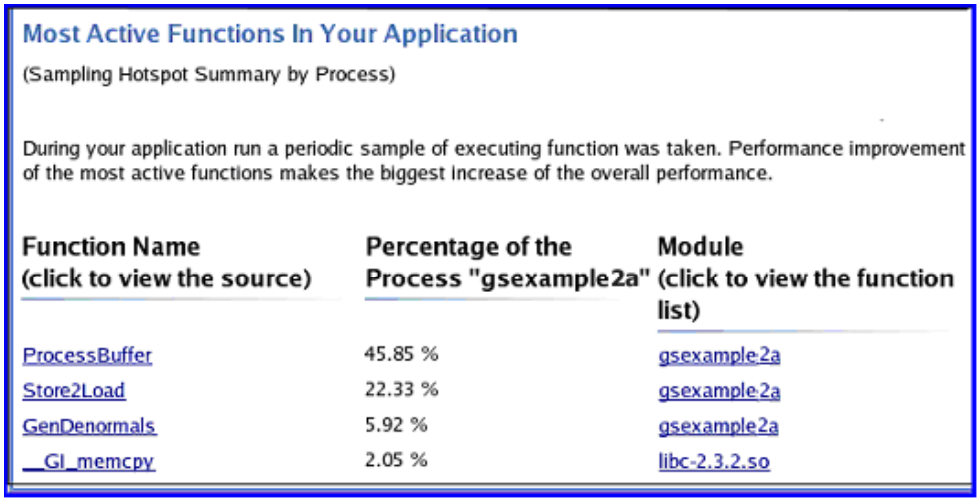


9. デフォルトの設定で gsexample2a アプリケーションを解析するアクティビティーを作成し実行するには、**[Finish (完了)]** をクリックします。

ここで、カーネルの場所を確認するメッセージが表示されることがあります。これは、アプリケーション・アクティビティーの多くがカーネルで動作し、VTune アナライザーがカーネルファイルの場所を特定できなかったためです。

カーネルの場所を確認するメッセージが表示された場合:
次回、このプロジェクトのカーネルをチューニングする場合は、**[Skip (スキップ)]** を選択します。
このプロジェクトのカーネルをチューニングしない場合は、**[Skip Always (常にスキップ)]** を選択します。

データ収集が完了すると、**[Sampling Summary (サンプリング・サマリー)]** ビューが表示されます。



Most Active Functions In Your Application

(Sampling Hotspot Summary by Process)

During your application run a periodic sample of executing function was taken. Performance improvement of the most active functions makes the biggest increase of the overall performance.

Function Name (click to view the source)	Percentage of the Process "gsexample2a"	Module (click to view the function list)
ProcessBuffer	45.85 %	gsexample2a
Store2Load	22.33 %	gsexample2a
GenDenormals	5.92 %	gsexample2a
__GI_memcpy	2.05 %	libc-2.3.2.so

[Sampling Summary (サンプリング・サマリー)] は、データ収集時にシステム内で最もアクティ

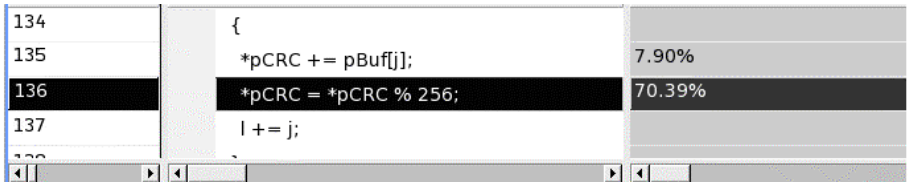
ブだった 5 つの関数に関するデータを提供します。関数をクリックして関数のソースビューを表示したり、関数が属するモジュールをクリックしてそのモジュール内の関数一覧を表示することができます。

2.3 結果の解析

サマリーの最初に表示され、最も時間を費やしているのが ProcessBuffer 関数です。
この関数に注目して、パフォーマンスを改善できるかどうか試してみましょう。

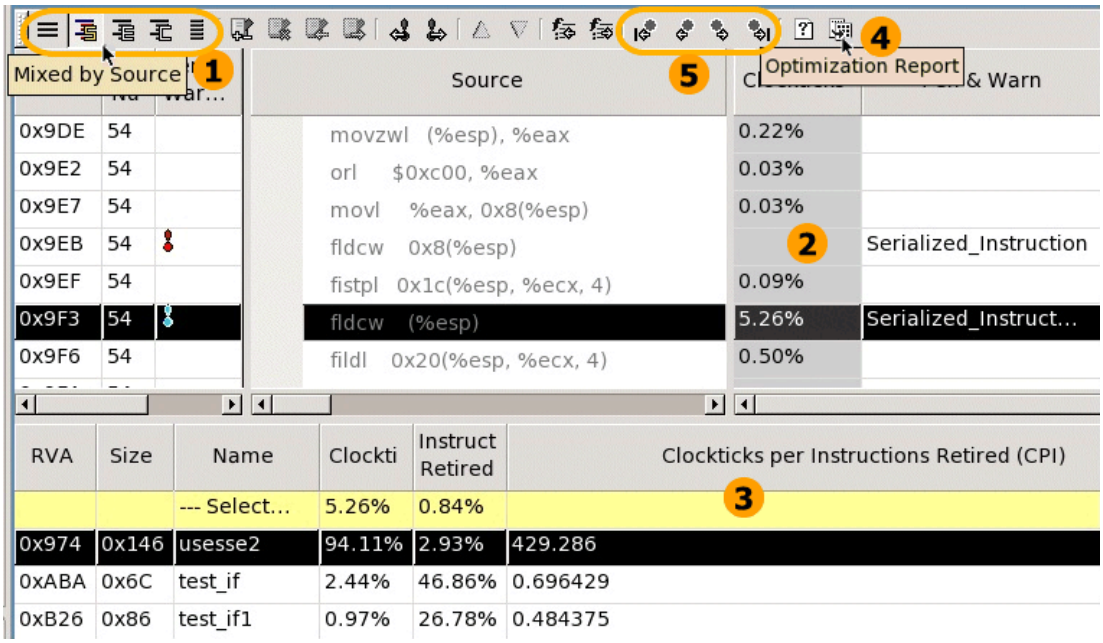
1. `ProcessBuffer` をクリックして、ソースコードを表示します。
2. **[Source (ソース)]** ビューを変更して、重要な情報を見やすくします。
   [Source (ソース)] ビューで右クリックして、ポップアップ・メニューから次を選択します。
   [View Events as (イベントを表示)] → **[% of Activity (% のアクティビティー)]**

コードをスクロールして、最も時間を費やしている行を見つけます。この例では、136 行目に対して最も高い Clockticks (Clocktick の回数) の割合が報告されています。



この機会に、**[Source (ソース)]** ビューのその他の機能について知っておくと良いでしょう。
次の図は、**[Mixed by Source (ソース行順による混合)]** を使用して表示する場合に、**[Source (ソース)]** ビューで利用可能な機能の説明です。

[First Use Wizard (簡易ウィザード)] を使用して、Clockticks (Clocktick の回数) イベントのデータを収集するアクティビティーを作成し、実行できます。Clocktick は、プロセッサーによって認識される時間の最小単位で、 プロセッサーが命令を実行するのに必要な時間のことです。



1	クリックしてビューモードを変更します。このビューでは、**[Mixed by Source (ソース行順による混合)]** モードを使用しています。	4	**[Optimization Report (最適化レポート)]** ボタンをクリックして、インテル® C コンパイラー 9.1 以降またはインテル® Fortran コンパイラー 9.1 以降を使用し、選択されたコード行のコンパイラー最適化レポートを生成します。
2	この命令に対するイベントの割合です。	5	クリックして、実行に時間のかかったコード行に移動します。
3	選択した範囲のコード行のサマリーデータです。		

2.4 コードの修正

データの解析により、サンプル・アプリケーションでは `ProcessBuffer` 関数が最も多くの時間を費やしており、そのコードの 136 行目における Clockticks (Clocktick の回数) が最多であることがわかりました。この問題の主な原因は、メモリーアクセスと不要なポインター参照による遅延にあります。各反復時に `pBuf` 引数を読み込み、`*pCRC` パラメーターをロードしストアしているためです。

`pBuf` と `*pCRC` の間には依存関係がないため (実際のパラメーターである `ProcessBuffer` 関数内の `buf` と `iCRC` は同じメモリーを参照しないため)、冗長なストアを削除し、`*pCRC` 変数を 136 行目のループの外に移動することができます。*pCRC パラメーターをローカル変数にロードすることにより、ポインター値のロードとその値によって指定されたアドレスにあるメモリーのロードが不要になり、メモリー参照を削減できます。

次の手順に従ってコードを修正するか、または /samples/gsexample/ ディレクトリーにある修正済みコード (gsexample2c.c ファイル) を使用してください。

3. アプリケーション開発に使用しているエディターでファイルを開きます。
4. ProcessBuffer 関数のソースを表示します。
5. INTEGER 型の新しいローカル変数 iChkSum を作成して、ループの前に *pCRC の値で初期化し、合計を累計するのに使用します。
6. ループの後に iChkSum を pCRC へ格納します。これにより、コンパイラーは iChkSum にレジスターを割り当て、メモリー操作を減らすことができます。さらに、コンパイラーは修正後のループに対してより積極的な最適化を実行することもできるようになります。この修正は、pCRC と pBuf に依存関係がないことを前提としている点に注意してください。
7. 次のようにコードを修正します。

```
long ProcessBuffer(char* pBuf, long bluffer, int* pCRC)
{
   int j, iChkSum = *pCRC; //new integer variable
   long l = lBufLen;
   // parse buffer
   // calculating modulo 256
   for (j = 0; j < lBufLen; j++)
   {
      iChkSum += pBuf[j];
      iChkSum = iChkSum % 256;
      l += j;
   }
#if MYDEBUG
   printf("...lBuflen = %ld\n", lBufLen);
#endif
   *pCRC = iChkSum; //store variable value
   return l / 2;
}
```

8. 次の設定でアプリケーションを再ビルドします。

```
cc -g -O2 gsexample2c.c -o gsexample2c
```

2.5 コードの修正前と修正後のパフォーマンス比較

このステップでは、コードの修正前と修正後のパフォーマンスを比較します。

1. 新しいバージョンのアプリケーションを生成するために、(元のアプリケーションの生成時と) 同じスイッチを使用して、修正後のアプリケーションをコンパイルします。または、

`/opt/intel/vtune/samples/gsexample` ディレクトリーにある gsexample2c アプリケーションを使用してください。

2. ベンチマークとパフォーマンスを比較します。次のコマンドを入力して修正したアプリケーションを実行します。

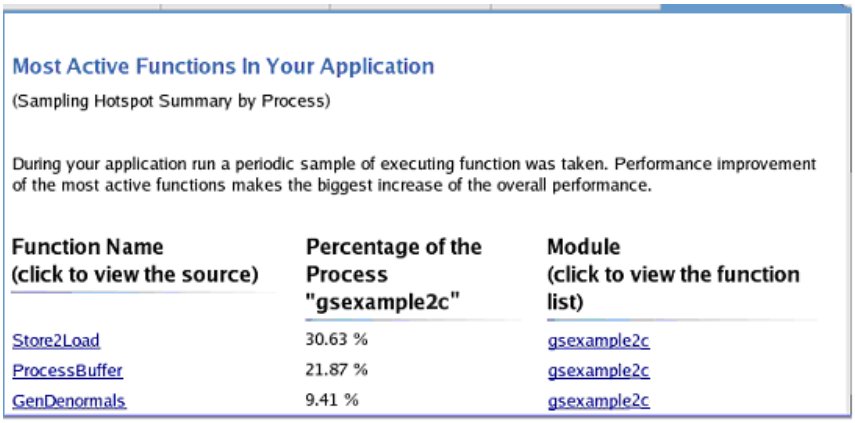
   `./gsexample2c datafile.txt`

   経過時間が著しく減少したことが確認できます。

```
*****************************************
* Elapsed time was 5 seconds
* File character count: 8653
* Total characters counted: 865300000
* CRC calculated: 0x72
*****************************************
```

3. VTune アナライザーを使用して、パフォーマンスが向上したことも確認できます。**[First Use Wizard (簡易ウィザード)]** を選択して、`gsexample2c.c` アプリケーションを起動します。デフォルトの設定を使用して、アクティビティーを作成し、結果を生成します。

   前回 (修正前のアプリケーションで) アクティビティーを実行した際に表示された **[Sampling Summary (サンプリング・サマリー)]** の内容と比較します。



Most Active Functions In Your Application

(Sampling Hotspot Summary by Process)

During your application run a periodic sample of executing function was taken. Performance improvement of the most active functions makes the biggest increase of the overall performance.

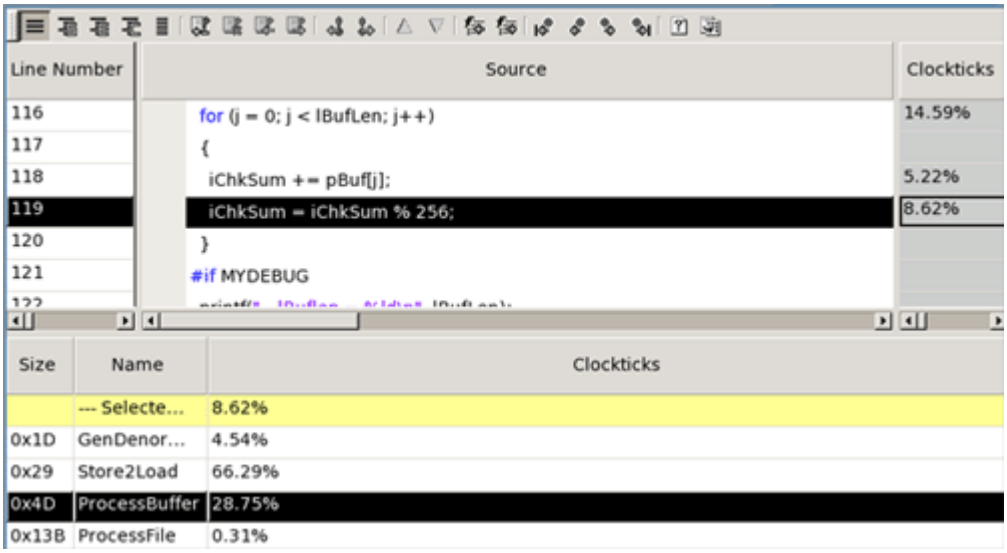
Function Name (click to view the source)	Percentage of the Process "gsexample2c"	Module (click to view the function list)
Store2Load	30.63 %	gsexample2c
ProcessBuffer	21.87 %	gsexample2c
GenDenormals	9.41 %	gsexample2c

   `ProcessBuffer` で費やされたプロセッサー時間の割合に注目すると、減少していることが確認できます。

4. `ProcessBuffer` 関数をクリックして関数のソースビューを表示します。

5. gsexample2a.c タブと gsexample2c.c タブをクリックすると、修正前の **[Source (ソース)]** ビューと修正後の **[Source (ソース)]** ビューを切り替えることができます。

6. `ProcessBuffer` 関数における Clockticks (Clocktick の回数) の割合が 70.39% から 8.26% に減少し、アプリケーションのパフォーマンスが改善されたことが確認できます。



3 アルゴリズムの解析

VTune アナライザー Linux 版は、コールグラフを使用して、コードのアルゴリズムを解析することができます。

コールグラフは、1 つの関数が他の関数を呼び出す回数、および各関数が自身のコードを実行したり、呼び出し先関数のコードを実行するのに費やす時間に関する情報を収集します。**[Call Graph Wizard (コールグラフ・ウィザード)]** を使用して、アルゴリズムをチューニングするためにアプリケーションをプロファイルします。

アプリケーションのプロファイルを開始するには、次の操作を行います。

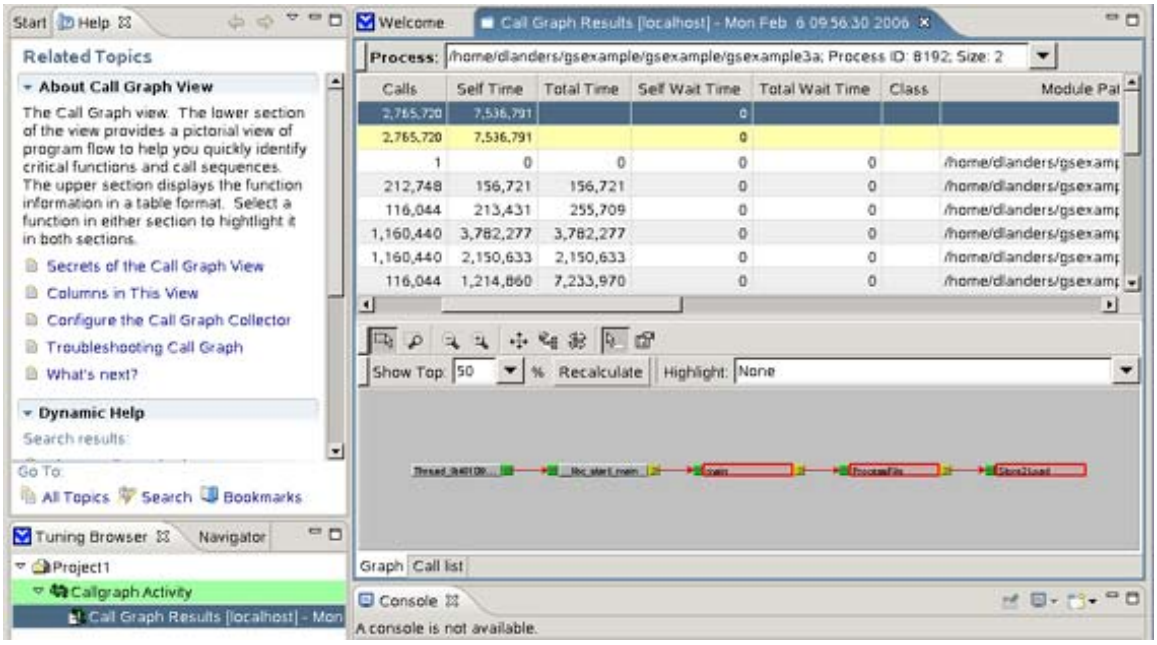
1. **[File (ファイル)]** メニューから、**[New (新規)] - [Project (プロジェクト)]** を選択します。
2. ダイアログボックスが表示されます。
3. **[Tuning Activity]** の **[Call Graph Wizard (コールグラフ・ウィザード)]** () を選択し、**[Next]** をクリックします。
4. **Linux* executable** 収集環境が選択されていることを確認して、**[Next]** をクリックします。
5. **[Application/Module Profile Configuration (アプリケーション/モジュール・プロファイルの設定)]** ページで、次の設定を入力します。
   a. **[Application to Launch (起動するアプリケーション)]** で `gsexample3a` ファイルを参照して、指定します。デフォルトでは、`/opt/intel/vtune/samples/gsexample` にあります。

b. **[Application Arguments (アプリケーションの引数)]** で `datafile.txt` を指定します。

c. **[Working Directory (作業ディレクトリー)]** で `/opt/intel/vtune/samples/gsexample` を指定します。

6. **[Finish (完了)]** をクリックして、データ収集を開始します。

プロファイリング結果は、右側のフレームの上部にある **[Function Summary (関数サマリー)]** ビュー、および下部にある **[Graph (グラフ)]** ビューと **[Call List (コールリスト)]** ビューに表示されます。



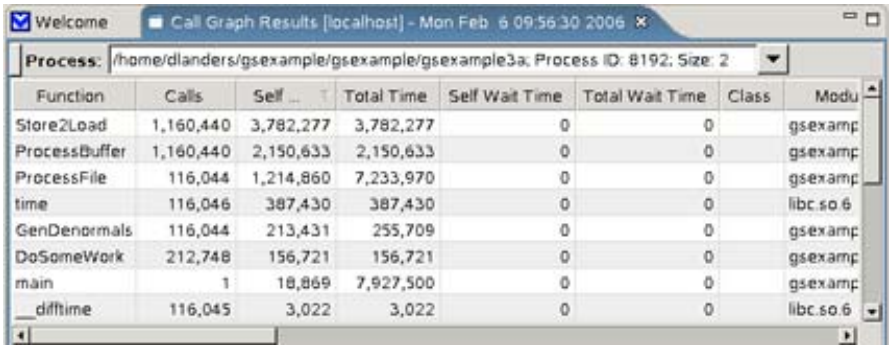
3.1 関数サマリービュー

[Function Summary (関数サマリー)] は、アプリケーションのすべての関数情報を表形式で表示します。ビューを調整してさまざまな方法で情報を表示し、アプリケーション内の各関数のパフォーマンスを比較することができます。

[Function Summary (関数サマリー)] の **[Hierarchy (階層)]** オプションは、マルチスレッド・アプリケーションを解析する際に便利です。`gsexample3a` はシングルスレッド・アプリケーションなので、**[Function Summary (関数サマリー)]** ビューを右クリックして、ポップアップ・メニューの **[Hierarchy (階層)]** をオフにします。

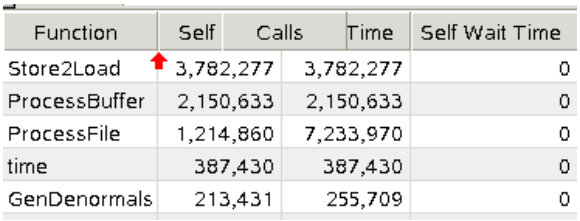
[Self Time (セルフ時間)] () カラムヘッダーをクリックして、自身のコードを実行するのに費やした時間で関数をソートします。

Self-Time (セルフ時間) - 関数内で費やされた時間 (マイクロ秒)。アクティビティーの実行で待たされた時間が含まれます。インストルメントされたその他の関数に対する呼び出しに費やされた時間は含まれません。

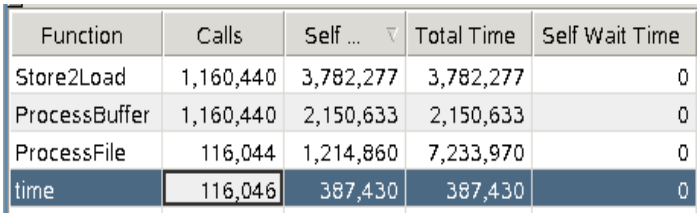


Function	Calls	Self ...	Total Time	Self Wait Time	Total Wait Time	Class	Modu
Store2Load	1,160,440	3,782,277	3,782,277	0	0		gsexamp
ProcessBuffer	1,160,440	2,150,633	2,150,633	0	0		gsexamp
ProcessFile	116,044	1,214,860	7,233,970	0	0		gsexamp
time	116,046	387,430	387,430	0	0		libc.so.6
GenDenormals	116,044	213,431	255,709	0	0		gsexamp
DoSomeWork	212,748	156,721	156,721	0	0		gsexamp
main	1	18,869	7,927,500	0	0		gsexamp
__difftime	116,045	3,022	3,022	0	0		libc.so.6

使いやすいように列の配置を変更できます。列を移動するには、カラムヘッダーをクリックして、任意の位置にドラッグします。 赤い矢印は、ドロップした際に列が表示される位置を示します。



Function	Self	Calls	Time	Self Wait Time
Store2Load	3,782,277	3,782,277		0
ProcessBuffer	2,150,633	2,150,633		0
ProcessFile	1,214,860	7,233,970		0
time	387,430	387,430		0
GenDenormals	213,431	255,709		0



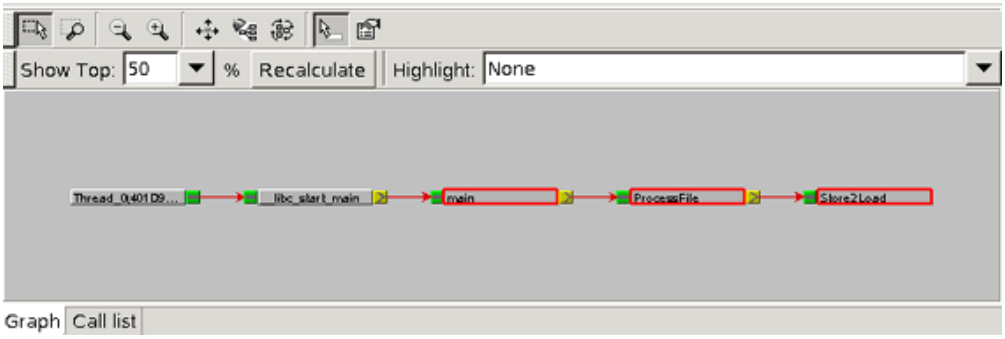
Function	Calls	Self ...	Total Time	Self Wait Time
Store2Load	1,160,440	3,782,277	3,782,277	0
ProcessBuffer	1,160,440	2,150,633	2,150,633	0
ProcessFile	116,044	1,214,860	7,233,970	0
time	116,046	387,430	387,430	0

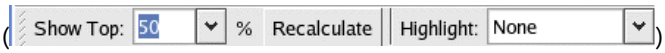
`time` 関数は、セルフ時間の値が 4 番目に大きい関数です。この関数の **Calls** 列の値は、`ProcessFile` 関数と同様に、ループごとに 1 回呼び出されます。これは不要なオーバーヘッドを発生させますが、`gsexample3b` のように次の行を追加することで、このオーバーヘッドを減らすことができます。

```
if ((fileCount % 1000) == 0) time(stop);
```

3.2 グラフビュー

[Graph (グラフ)] ビューは、アプリケーションの実行をグラフィカルに表示します。デフォルトでは、クリティカル・パス上の関数のみを表示します。この例では、スレッドの開始から `Store2Load` 関数までがクリティカル・パスです。

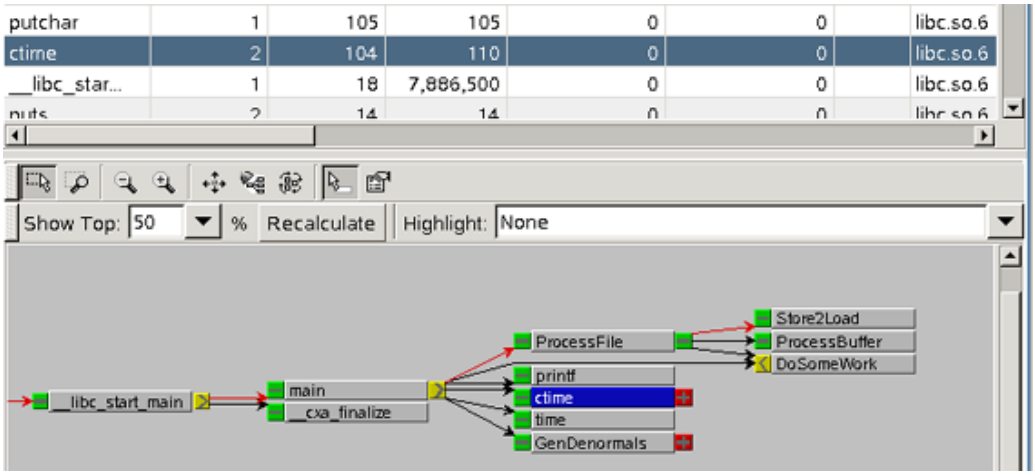


このビューの内容を制御するには、**[Graph (グラフ)]** ビューの上部にある **[Filter (フィルター)]** ツールバー (

) を使用します。このツールバーを使用して、解析結果の関数に対して、独自のフィルターを設定できます。**[Show top (始点を表示)]** ドロップダウン・メニューで、一度に表示する関数の割合 (5、10、20、50%、またはすべて) を選択できます。割合を選択して、**[Recalculate (再計算)]** ボタンをクリックすると結果が表示されます。また、**[Highlight (ハイライト)]** ドロップダウン・メニューから基準を選択して、特定の関数をハイライトすることもできます。この機能は、関数が多く含まれているアプリケーションを解析する際に便利です。そのような場合、グラフは複雑になりますが、フィルタリングを使用することで最も時間を費やす関数を特定できます。

[Graph (グラフ)] ビューと **[Function Summary (関数サマリー)]** ビューは互いに同期しています。1 つのビューで関数を選択すると、もう一方のビューでも選択された関数がハイライトされます。例:

1. **[Show top (始点を表示)]** ドロップダウン・メニューで 50% を選択して、**[Recalculate (再計算)]** をクリックします。**[Graph (グラフ)]** ビューに、アプリケーション内のセルフ時間を持つ関数の上位 50% が表示されます。ただし、`ctime` 関数は関数サマリービューには表示されますが、グラフでは非表示になっていることに注意してください。
2. 関数サマリービューで `ctime` をダブルクリックすると、グラフにノードが表示されます。



3. グラフビューで `ProcessFile` ノードをクリックします。

   `ProcessFile` がフォーカス関数となり、関数サマリービューとグラフビューで自動的にハイライトされます。

3.3 コール・リスト・ビュー

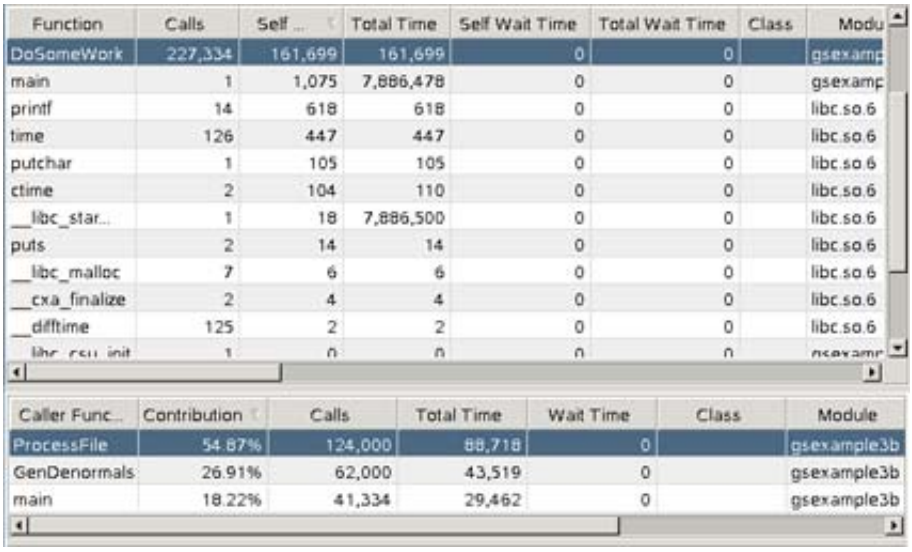
呼び出し先関数 - 現行の関数によって呼び出された子関数。
呼び出し元関数 - 現行の関数を呼び出す親関数。

[Call list (コールリスト)] ビューは、選択した (フォーカス) 関数、その呼び出し元および呼び出し先の情報を表形式で表示します。

コール・リスト・ビューを表示するには、**[Call List (コールリスト)]** タブをクリックします。

このビューでは、フォーカス関数の合計時間において、どの関数が、どれだけの割合を占めているのか確認できます。コールリストは、呼び出し元関数 (`main`) と呼び出し先関数 (`Store2Load`、`ProcessBuffer`、`DoSomeWork`) の 2 つのウィンドウに分かれています。

`DoSomeWork` をダブルクリックしてフォーカス関数にすることで、関数の相互関係を確認できます。



Function	Calls	Self ...	Total Time	Self Wait Time	Total Wait Time	Class	Modu
DoSomeWork	227,334	161,699	161,699	0	0		gsexamp
main	1	1,075	7,886,478	0	0		gsexamp
printf	14	618	618	0	0		libc.so.6
time	126	447	447	0	0		libc.so.6
putchar	1	105	105	0	0		libc.so.6
ctime	2	104	110	0	0		libc.so.6
__libc_star...	1	18	7,886,500	0	0		libc.so.6
puts	2	14	14	0	0		libc.so.6
__libc_malloc	7	6	6	0	0		libc.so.6
__cxa_finalize	2	4	4	0	0		libc.so.6
__difftime	125	2	2	0	0		libc.so.6
__libc_csu_init	1	0	0	0	0		gsexamp

Caller Func...	Contribution	Calls	Total Time	Wait Time	Class	Module
ProcessFile	54.87%	124,000	88,718	0		gsexample3b
GenDenormals	26.91%	62,000	43,519	0		gsexample3b
main	18.22%	41,334	29,462	0		gsexample3b

呼び出し元関数のリストには、DoSomeWork 関数を呼び出すすべての関数と、それぞれの関数の呼び出し総数に対する割合が表示されます。

ProcessFile 関数をダブルクリックして、フォーカス関数とします。Store2Load 関数と ProcessBuffer 関数は、ProcessFile を呼び出すたびに 10 回ずつ呼び出されるため、オーバーヘッドが発生します。このオーバーヘッドを減らす 1 つの方法は、バッファーサイズを大きめに割り当てることです。ファイル全体を保持できる位の大きなバッファーを割り当てることもできます。

3.4 コードの修正

コードの解析では、バッファーサイズが足りないために、Store2Load 関数と ProcessBuffer 関数が頻繁に呼び出されることがわかりました。次の手順に従ってコードを修正するか、または /opt/intel/vtune/samples/gsexample ディレクトリーにある修正済みのコード (gsexample3c.c) を使用してください。

1. アプリケーション開発に使用しているエディターで gsexample3c.c ファイルを開きます。
2. ProcessFile 関数のソースを開きます。
3. オーバーヘッドを減らすためには、ファイル全体のバッファーを一度に割り当てるようにコードを書き直します。

```
if (fd >= 0)
{pbuf = malloc(filelen);
   if(pbuf == NULL)
     {printf( "\n*** Error: failed to allocate enough memory for file!
Aborting. ***\n" );
                 exit(3);}
              actual = read(fd, pbuf, filelen);
              //removed if (actual > 0)
                fileCharCount += ProcessBuffer(pbuf,(long)
                actual, &iCRC);
                Store2Load(pbuf, (long) actual);
                close(fd);
                free(pbuf);
                *pCRC = iCRC;
            f = DoSomeWork();}
```

4. `ProcessFile` 関数にポインターを渡します。

```
long ProcessFile(char* cFileName, char* pbuf, long filelen, int*
pCRC)

if (fd >= 0)
         {actual = read(fd, pbuf, filelen);
     fileCharCount += ProcessBuffer(pbuf,
     (long) actual, &iCRC);
     Store2Load(pbud, (long) actual);
     close(fd);
     *pCRC = iCRC;
     f = DoSomeWork();}
```

5. 前回と同じ設定でアプリケーションを再ビルドします。1 秒あたりに処理される文字数が増加したかどうか確認します。

4 コード内のイベントの解析

[Sampling Wizard (サンプリング・ウィザード)] では、データ収集中に発生したプロセッサー・イベントに関する特定のデータを、さまざまな観点から収集します。この情報を使用して、プロセッサーを遅くするイベントの発生を減らすようにコードの配置を換えて、コードのパフォーマンスを改善します。

サンプルが収集される頻度は、サンプリング・データ収集の間にシステム上で実行中のソフトウェアがイベントを発生させる頻度によって決まります。デフォルトでは、Clockticks (Clocktick の回数) イベントが選択されています。

アプリケーションの解析を始める前に、選択可能なイベントに関する情報を入手しておくと便利です。オンラインヘルプの**「リファレンス」**にある**「プロセッサーのイベントとアドバイス (Processor Events and Advice)」**ブックの対象プロセッサーを参照してください。

4.1 [Sampling Wizard (サンプリング・ウィザード)] の使用

アプリケーションをチューニングする前に、gsexample3a を使用して、別のベンチマークを作成します。

新規アクティビティーの作成:

1. VTune アナライザーのツールバーで、**[Add Tuning Activity (チューニング・アクティビティーの追加)]** をクリックして、**[Select a Wizard (ウィザードの選択)]** ダイアログボックスを開きます。
2. **[Sampling Wizard (サンプリング・ウィザード)]** () を選択します。
3. **[Next (次へ)]** をクリックして、**[Environment and Activity Settings (環境とアクティビティー設定)]** ページに進みます。
4. **Linux* executable** 収集環境を選択し、**[Select application type (アプリケーションの種類を選択)]** ボックスで **[Application (アプリケーション)]** を選択します。
   [Next (次へ)] をクリックし、**[Application/Module Profile Settings (アプリケーション/モジュール・プロファイル設定)]** ページに進みます。
5. 次の設定を行います。
   a. **[Application to Launch (起動するアプリケーション)]** で `gsexample3a` ファイルを参照して、指定します。
   b. **[Application Arguments (アプリケーションの引数)]** で、`datafile.txt` を指定します。
      [Next (次へ)] をクリックして、**[Sampling Collection Settings (サンプリング収集設定)]** ページに進みます。
6. **[When the application terminates (before duration completes) (アプリケーションの終了時 (サンプリング間隔前))]** をオフにします。
   [Next (次へ)] をクリックして、**[Options (オプション)]** ページに進みます。
7. **[Modify Configuration (設定を変更)]** で、次の設定を行います。

イベント・ベース・サンプリング中に、サンプリング・コレクターは複数回実行されます。各実行で、サンプリング・コレクターは次のいずれかの操作を行います。
選択されたイベントのサンプリング間隔の値 (SAV) のキャリブレーションを行います。
データ収集中、データはキャリブレーションが行われた**サンプリング間隔の値 (SAV)** に基づいて収集されます。
デフォルトのサンプリング間隔の値 (SAV) を使用して、選択したイベントからデータを収集します。

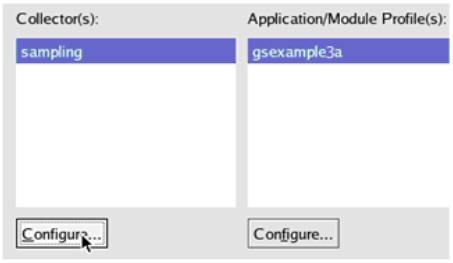
a. **[Change the Sampling Events and set advanced options such as interval, calibration, and delay (サンプリング・イベントを変更して間隔、キャリブレーション、遅延などの詳細情報を設定)]** チェックボックスをオンにします。

b. **[Finish (完了)]** をクリックした後にアクティビティーを実行するには、**[Run this Activity (アクティビティーを実行)]** チェックボックスをオンにします。

8. **[Finish (完了)]** をクリックして、アクティビティーを作成し、**[Activity Configuration (アクティビティーの設定)]** ダイアログボックスを開きます。

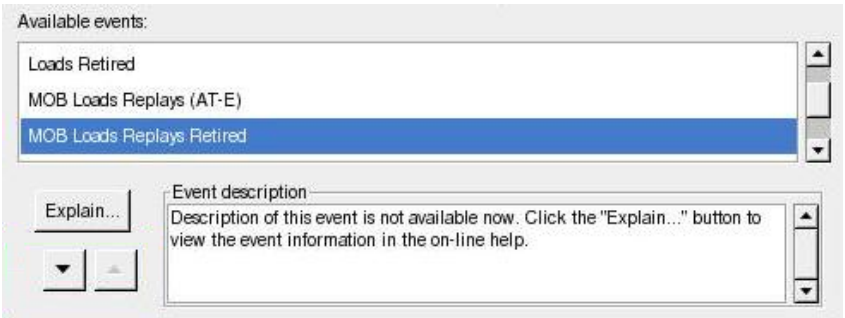
イベント・ベース・サンプリング (EBS) を使用してアプリケーションを解析します。

1. **[Activity Duration (アクティビティー継続期間)]** フィールドで **[second(s) (秒)]** ラジオボタンをクリックして、編集可能なボックスに 10 と入力します。これにより、アクティビティーを実行してデータを収集する時間が 10 秒に設定されます。
2. **[Collector(s) (コレクター)]** リストの下にある **[Configure... (設定...)]** ボタンをクリックして、サンプリング・コレクターを設定します。



[Configure Sampling (サンプリングの設定)] ダイアログボックスが表示されます。

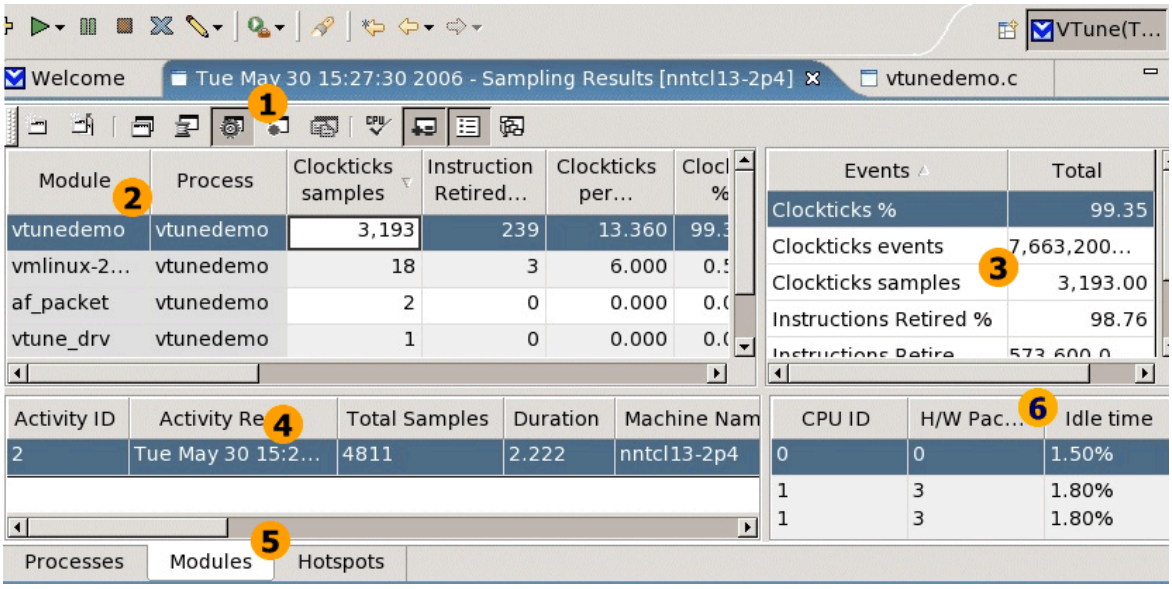
3. **[Events (イベント)]** タブを選択します。
4. **[Available Events (選択可能なイベント)]** フィールドで、MOB Load Replays Retired (リタイアした MOB ロードのリプレイ) イベントを選択します。次の図に示すように矢印をクリックして、**[Selected Events (選択されたイベント)]** リストに追加します。



5. **[OK]** をクリックして、ダイアログボックスを閉じます。そして、**[OK]** をクリックして、データ収集を開始します。VTune アナライザーは、アクティビティーを実行して、終了時にサンプリング結果を表示します。

4.2 サンプリング結果の表示

データをフレキシブルに表示できるサンプリング・ビューの機能について次に示します。



1	クリックしてビューモードを変更します。	2	この列はデータが収集されたモジュールを表示します。
3	選択サマリーは、モジュールで収集されたイベント、サンプル、およびサンプルの割合の合計を表示します。**[Selection Summary (選択サマリー)]** ( ) ボタンを使用してビューを切り替えます。	4	収集したサンプリング・データに関する一般的な情報です。十分かつ適切な量のサンプルが収集されたかどうかを確認するのに使用します。
5	これは表形式の **[Module (モジュール)]** ビューです。収集したデータを水平バーチャート形式で表示するには、右クリックして **[View as Bar Chart (バーチャートで表示)]** を選択します。	6	コアごとのアクティビティー・サマリーです。完全に活用されていないコアがあるかどうかを確認するのに使用します。

ビューには、次のような有益な情報が表示されます。

- **[Process (プロセス)]** ビュー – サンプリング・データが収集されたときにシステム上で実行されていたすべてのプロセスを表示します。特定プロセス中の高いイベント数は、潜在的なパフォーマンス・ボトルネックになる可能性のある、高い CPU 使用率を示します。
- **[Module (モジュール)]** ビュー – これらの情報を使用して、**[Module (モジュール)]** ビューから **[Hotspot (ホットスポット)]** ビューにドリルダウンします。サンプリング・データ収集中に頻繁に呼び出されたモジュールは、最大イベント数または最大 CPU 時間のモジュールとして表示されます。

[Sampling (サンプリング)] ビューを使用して、問題のあるコード部分を特定します。

1. デフォルトで表示される **[Process (プロセス)]** ビューの gsexample3a プロセスのセクションを確認します。

2.  をクリックして、**[Module (モジュール)]** ビューに切り替えます。
3. gsexample3a モジュールをクリックして選択します。
4.  をクリックして、gsexample3a モジュールで最もアクティブな関数を表示します。
5. Store2Load 関数で、最も多くのイベントと MOB Load Replays Retired (リタイアした MOB ロードのリプレイ) が発生していることが確認できます。ここが、このコードの問題箇所です。
6.  をクリックして、この関数の **[Source (ソース)]** ビューを表示します。

4.3 コードの修正

コードの解析では、ブロックされたストアフォワード問題が見つかりました。次の手順に従ってコードを修正するか、または /opt/intel/vtune/samples/gsexample ディレクトリーにある修正済みのコード (gsexample4) を使用してください。

1. アプリケーション開発に使用しているエディターでファイルを開きます。
2. Store2Load 関数のソースを開きます。

   コードの問題箇所を次に示します。

   ```
   *(((char*) (pBits + i)) + 1) = 0;
   clr = *(pBits + i);
   ```

   ここで問題となっているのは、最初の行で 32 ビットの値の中の 1 バイトをクリアした後に、その 32 ビットの値をロードしていることです。プロセッサーはバイトの書き込みが終わるまで、問題のバイトを含む 32 ビットの値をロードすることができないため、ブロックされたストアフォワードが発生します。
3. この問題を回避するには、同じサイズのデータを読み取り、書き出す必要があります。32 ビットの値を読み取り、この値の中の 1 バイトを 32 ビットの整数でクリアし、32 ビットの値として格納するように、コードを変更します。

   修正後のコードを次に示します。

   ```
   clr = *(pBits + i);
   clr &= 0xffff00ff;
   *(pBits + i) = clr;
   ```
4. 前回と同じ設定でアプリケーションを再ビルドします。

4.4 コードの修正前と修正後のパフォーマンス比較

修正したコードの結果を比較するには、別のサンプリング・アクティビティーを作成します。最適化されたコードを使用するか、または /opt/intel/vtune/samples/gsexample ディレクトリーにある gsexample4c アプリケーションを使用してください。アクティビティーを作成し実行して、結果を確認します。

Name	Instructions Retired ...	Clockticks samples ▽	MOB Loads Replays Retired s
ProcessBuffer	22,376	29,380	16
GenDenormals	130	4,782	0
Store2Load	2,011	1,099	2

MOB Load Replays Retired (リタイアした MOB ロードのリプレイ) の数が減少し、アプリケーションのパフォーマンスが向上したことが確認できます。

5 次のステップ

VTune アナライザーを使用して、コンパイラーによる最適化情報をフィルターすることができます。通常、これらのレポートには、チューニングに関する優れた情報が含まれています。

[Source (ソース)] ビューでコード行を選択し、**[Optimization Report (最適化レポート)]** () をクリックして、選択した行のコンパイラー・アドバイスを表示します。この機能は、インテル・コンパイラー 9.1 以降でサポートされていますが、その他のコンパイラーでも利用可能な標準形式を使用しています。詳細は、オンラインヘルプで「Compiler Optimization (コンパイラーによる最適化)」を検索してください。<installdir>/vtune/samples にあるコンパイル済みのファイルは、互換性のあるコンパイラーをインストールしなくても試すことができます。

VTune アナライザーでは、データコレクターを使用してアプリケーションのさらなる解析を行うことができます。製品の機能を理解する上で役立つ Web ベースのチュートリアルが <installdir>/vtune/training/gs_vtl/index.htm にあります。ご活用ください。

インテル® ソフトウェア開発製品の詳しい情報については、次のインテル Web サイトを参照してください。
http://www.intel.co.jp/jp/software/products/

テクニカルサポートおよび製品の制限事項については、製品のリリースノートを参照してください。