

VTune(TM) Performance Analyzer for Linux

Getting Started Guide

The VTune™ Performance Analyzer provides information on the performance of your code. The VTune analyzer shows you the performance issues, enabling you to focus your tuning effort and get the best performance boost in the least amount of time.

The goal of this guide is to introduce you to the basic features of the VTune analyzer.

After completing this guide, you will be able to use the VTune analyzer to analyze your code and understand where to focus your tuning efforts to gain the most performance improvement.

This document will step you through the iterative process of tuning a sample application and step you through the stages of performance tuning:

- Locate a performance issue
- Revise the code to remove the issue
- Compare the performance of the new code with the initial code

Contents

Disclaimer and Legal Information	2
1 Build the Application.....	3
2 Analyze Your Application	3
3 Analyze Your Algorithms	10
4 Analyze Events in Your Code.....	15
5 Next Steps	19



Disclaimer and Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

MPEG is an international standard for video compression/decompression promoted by ISO. Implementations of MPEG CODECS, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

This document as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Chips, Core Inside, Dialogic, EtherExpress, ETOX, FlashFile, i386, i486, i960, iCOMP, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, MMX logo, Optimizer logo, OverDrive, Paragon, PDCharm, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, Pentium Inside, skool, Sound Mark, The Computer Inside., The Journey Inside, VTune, Xeon, Xeon Inside and Xircom are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2006-2007, Intel Corporation.



1 Build the Application

Before you start, you need to build the sample application with the settings that enable the VTune analyzer to collect the most useful data. Build the application with compiler settings for production-level optimization and symbol information. This means the code is fully optimized (as it will be when it is released) but includes symbol information.

Our example is compiled with the GNU C compiler, with ' -g -O2 ' settings (version 3.2.3).

NOTE: The results on your system may vary, based on system configuration and compiler version.

To build the application:

1. cd to the `opt/intel/vtune/samples/gsexample` directory
2. Enter: `make`
Or, if you are using the Intel® Compiler, enter:
`make CC=icc CFLAGS="-g -O2"`

2 Analyze Your Application

After building the application, you can go through the process of analyzing the performance of the code provided in the `/samples/gsexample` directory. The VTune analyzer uses data collectors to collect different types of performance data. In this step, you will use the First-use wizard to collect data, view the results and zoom into the specific problem areas of the source code. The First-use wizard collects system-wide data non-intrusively, and provides basic data on the five most active functions in your application.

2.1 Create a Benchmark

Create a *benchmark* of the original performance:

1. Run the `gsexample2a` application by entering the following command:
`#. /gsexample2a datafile.txt`
where
`datafile.txt` is the data file.



2. After the application run, you can see the elapsed time at the command line console, similar to the following image:

```
*****  
* 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  
*****
```

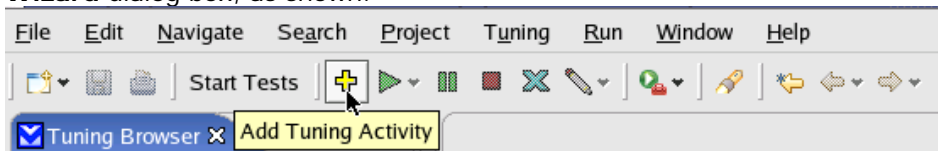
A benchmark must be measurable and reproducible so that it can be used as a basis for comparison of future revisions.

The Elapsed time is your benchmark for this phase of tuning the application.

2.2 Use the Wizard to Collect Data

In this step, you will use the First-use Wizard to create and run an *Activity* that collects performance data and saves it in a VTune analyzer project.

1. From the command-line, go to your VTune analyzer installation directory and run `vtlec` to start the VTune analyzer. Enter the following command:
`# /opt/intel/vtune/bin/vtlec`
The VTune analyzer launches and shows the start-up splash screen and the **Select Workspace** dialog box.
2. At the **Select Workspace** dialog box, click **OK** to use the default setting.
3. On the VTune analyzer toolbar click **Add Tuning Activity** to open the **Select a Wizard** dialog box, as shown:



4. Select the **First Use Wizard** and click **Next** to open the First-use wizard.
5. In the **Application to profile** field, click **Browse** to browse to the `gsexample2a` application. The dialog box opens.
6. In the dialog box, browse to the `gsexample2a` file located at `/opt/intel/vtune/samples/gsexample` and click **OK**.
7. In the **Application Arguments** field, enter the name of the data file `datafile.txt`.



- In the **Working directory** field, click **Browse** to browse for the gsexample directory, as shown:

- Click **Finish** to have the VTune analyzer create and run the Activity to analyze the gsexample2a application using the default settings.

At this point, the VTune analyzer may prompt for the kernel's location. This is because much of the application activity is processing in the kernel and the VTune analyzer did not locate an uncompressed kernel file.

When prompted for kernel location:

Select **Skip** if you intend to tune the kernel in this project, in a future session.

Select **Skip Always** if you do not intend to tune the kernel of this project file.

When data collection completes, the Sampling Summary view opens:

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

The Sampling Summary provides data on the five most active functions in the system during the data collection. Click the function to open the source view of the function, or click the module it belongs to view the full list of functions in the module.



2.3 Analyze the Results

The first function listed in the summary and the one that takes the most time, is `ProcessBuffer`. Focus on this function to see if you can find a way to improve its performance.

1. Click on the `ProcessBuffer` to view its source code.
2. Rearrange the source view to better see the important information: Right-click on a source view and in the pop-up menu select **View Events As → % of Activity**

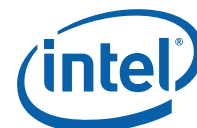
Scroll through the code to see what lines of code consumed the most processor time. For example, notice that line #136 has a large percentage of the clockticks reported against it.

134	{	
135	*pCRC += pBuf[j];	7.90%
136	*pCRC = *pCRC % 256;	70.39%
137	l += j;	

The First-use wizard helps you create and run an Activity that collects data on the clockticks event. a clocktick is the smallest unit of time recognized by the processor. It is the time required by the processor to execute an instruction.

This is a good opportunity to get to know some other features of the Source view. The screen cap below highlights and explains some of the features available in the Source view when viewing Mixed by Source mode.

RVA	Size	Name	Clockti	Instruct Retired	Clockticks per Instructions Retired (CPI)
---	---	Select...	5.26%	0.84%	
0x974	0x146	usesse2	94.11%	2.93%	429.286
0xA8A	0x6C	test_if	2.44%	46.86%	0.696429
0xB26	0x86	test_if1	0.97%	26.78%	0.484375



1	Click to change the view mode. This view is in "Mixed by Source mode."	4	Click the Optimization Report button to generate compiler optimization report for the selected code line using the Intel(R) C or Intel(R) Fortran Compiler 9.1 (or higher).
2	Percentage of events assigned to this instruction.	5	Use these buttons to navigate between code lines that took a long time to execute.
3	Summary data for the selected range of code lines		

2.4 Revise the Code

In the analysis of the data above, you identified that the function `ProcessBuffer` was consuming most of the time in the sample application and that the code line #136 shows the largest number of Clockticks. The main reasons for this are the memory accesses and pointer differences: the code reads the `pBuf` argument and loads and stores the `*pCRC` parameter on each loop iteration.

There are no dependencies between `pBuf` and `*pCRC` (actual parameters of `ProcessBuffer` – `buf` and `iCRC` do not reference the same memory), therefore you can remove the redundant stores and move the `*pCRC` variable outside of the loop at line #136. Loading it into a local variable reduces memory references; there is no need to load the pointer value and then load the memory pointed to it by the value.

Follow these steps to revise the code, or use the revised code provided in the `gsexample2c.c` file in the `/samples/gsexample/` directory.

3. Open the file in the editor you use for developing your application.
4. Go to the source of the `ProcessBuffer` function.
5. Introduce new local integer variable `iChkSum` initialized before the loop with value of `*pCRC` and use it for accumulating the sum.
6. After the loop, store `iChkSum` to `pCRC`. This enables the compiler to register `iChkSum` and decrease of number of memory operations. This change also enables the compiler to perform other more aggressive optimizations on this loop. Note that this change is only possible if you are sure that `pCRC` and `pBuf` are independent.
7. Modify the code as follows:



```
long ProcessBuffer(char* pBuf, long lBufLen, 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;
    }
#ifdef MYDEBUG
    printf("...lBuflen = %ld\n", lBufLen);
#endif
    *pCRC = iChkSum; //store variable value
    return l / 2;
}
```

8. Rebuild your application with the following settings:

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

2.5 Compare Performance with the Original Code

In this step, you will compare the performance results of the new code with the results of the original code.

1. Compile the revised application, using the same switches to generate a new version of the application. Or, you can use gsexample2c application provided in `/opt/intel/vtune/samples/gsexample`
2. Compare the performance against the benchmark. Run the revised application by entering the command:

```
./gsexample2c datafile.txt
```

You can see that the elapsed time decreased:

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

3. You can also see the performance improvement using the VTune analyzer. Open the First Use wizard and launch the `gsexample2c.c` application. Use the default settings to create and Activity and generate results as in section 2a. Compare the new Sampling Summary data to the data in the Sampling Summary you saw after the first Activity run.



Most Active Functions In Your Application

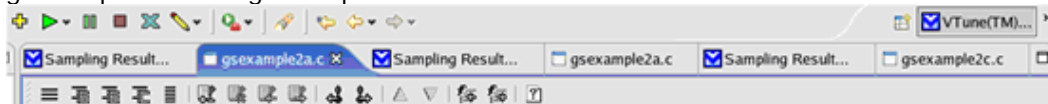
(Sampling Hotspot Summary by Process)

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

Look at the percentage of the process the `ProcessBuffer`. Note its decrease.

- Click the `ProcessBuffer` function link to open its Source View.
- Switch between the previous source view and the new one by clicking on the `gsexample2a.c` and `gsexample2c.c` tabs.



- The percentage of the clockticks for the `ProcessBuffer` function is less than before (8.26%). Application performance has improved!

Line Number	Source	Clockticks
116	<code>for (j = 0; j < lBufLen; j++)</code>	14.59%
117	<code>{</code>	
118	<code> iChkSum += pBuf[j];</code>	5.22%
119	<code> iChkSum = iChkSum % 256;</code>	8.62%
120	<code>}</code>	
121	<code>#if MYDEBUG</code>	
122	<code> printf(" lBufLen: %d\n", lBufLen);</code>	


Size	Name	Clockticks
---	Selecte...	8.62%
0x1D	GenDenor...	4.54%
0x29	Store2Load	66.29%
0x4D	ProcessBuffer	28.75%
0x13B	ProcessFile	0.31%

3 Analyze Your Algorithms

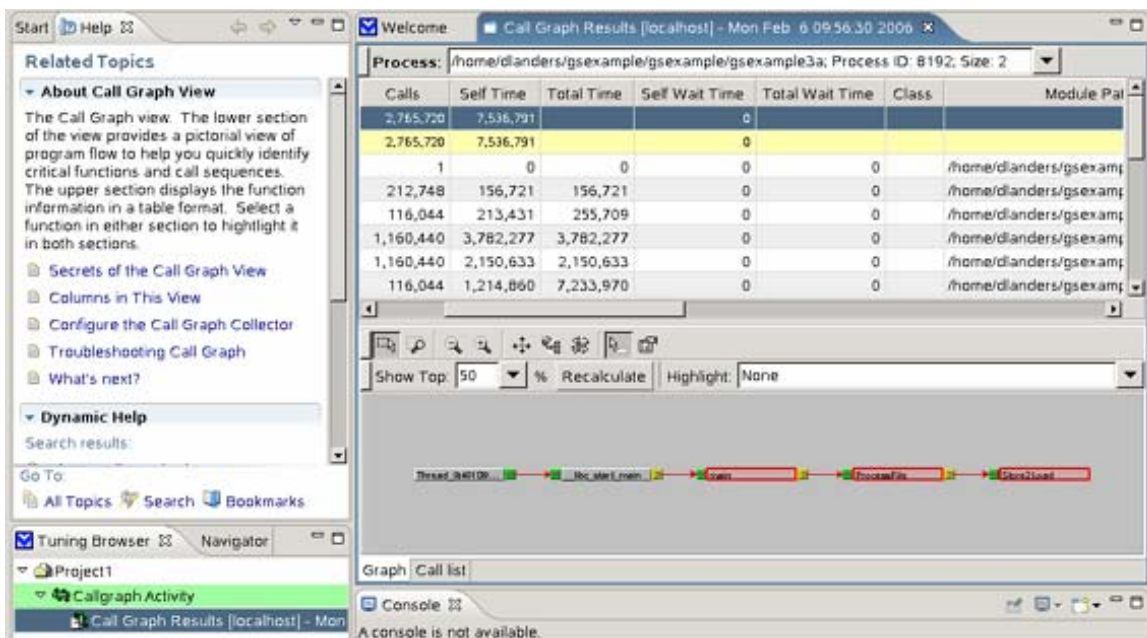
The VTune(TM) Performance Analyzer for Linux* can analyze the algorithms of your code with Call Graph.

Call Graph gathers information about how many times a function calls other functions and the amount of time each function spent executing its code versus the code of called functions. To profile your application for algorithmic tuning, use the Call Graph Wizard.

To start application profiling, do the following:

1. In the **File** menu, select **New Project**.
2. A dialog box appears.
3. In this dialog window, select  **Call Graph Wizard** and click **Next**.
4. Make sure the collection environment is set as **Linux* executable** and click **Next**.
5. In the **Application/Module Profile Settings** page, enter these settings:
 - a. Set the **Application to launch** by browsing to the `gsexample3a` file. By default it is in: `/opt/intel/vtune/samples/gsexample`
 - b. Set the **Application Arguments** as `datafile.txt`.
 - c. Set the **Working Directory** as `/opt/intel/vtune/samples/gsexample`.
6. Click **Finish** to start the data collection.

The VTune analyzer displays profiling results in the Function Summary view at the top of the right frame and the Graph and Call List view at the bottom:



Calls	Self Time	Total Time	Self Wait Time	Total Wait Time	Class	Module Path
2,765,720	7,536,791		0			
2,765,720	7,536,791		0			
1	0	0	0	0		/home/dlanders/gsexamp
212,748	156,721	156,721	0	0		/home/dlanders/gsexamp
116,044	213,431	255,709	0	0		/home/dlanders/gsexamp
1,160,440	3,782,277	3,782,277	0	0		/home/dlanders/gsexamp
1,160,440	2,150,633	2,150,633	0	0		/home/dlanders/gsexamp
116,044	1,214,860	7,233,970	0	0		/home/dlanders/gsexamp

Graph Call list

Console


A console is not available.

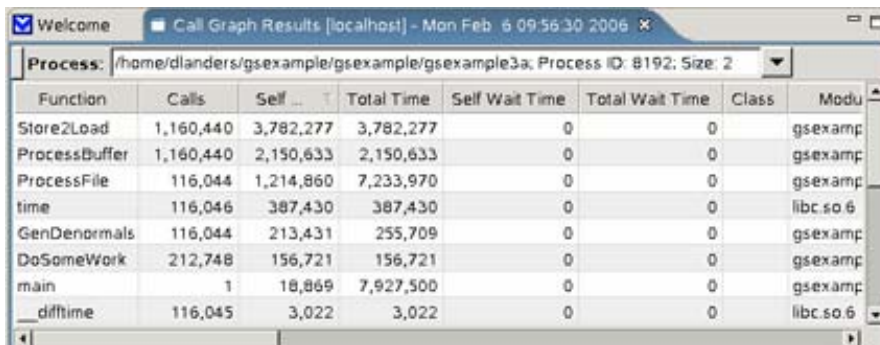


3.1 Function Summary View

Function Summary provides full information on all the application functions in table format. The view is adjustable, enabling you to view the information in different ways. It enables you to compare performance of each function in the application against one another.

The Function Summary **Hierarchy** option is very useful, when dealing with multithreaded applications. However, since `gsexample3a` is a single-threaded application, you can turn it off by right-clicking the Function Summary view and unchecking **Hierarchy** in the pop-up menu.

Click the  **Self Time** column heading to sort the functions by the time spent executing its own code:



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	

Self-Time - time (microseconds) spent inside a function, including time spent waiting between execution activities. It does not include time spent in calls to other instrumented functions.

You may want to rearrange columns for more convenience. To move a column, click the column header and drag it to the desired position. A red arrow indicates where the column will be placed when you drop it.

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

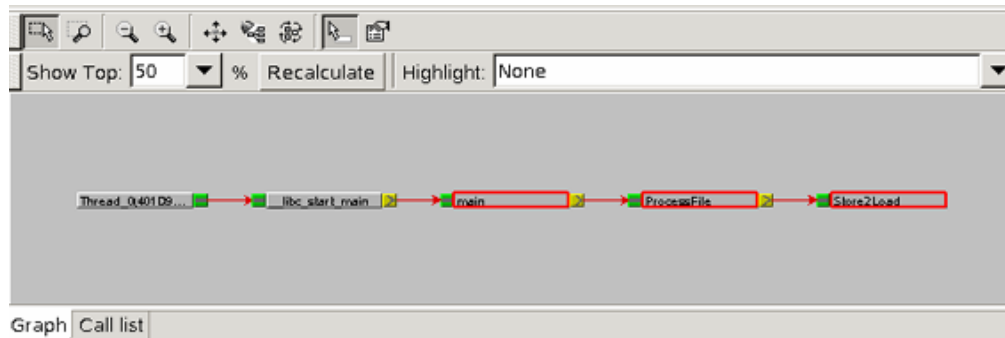
Function	Calls	Self ...	Total Time	Self Wait Time
Store2Load	1,160,440	3,782,277	3,782,277	0
ProcessBuffer	1,160,440	2,150,633	2,150,633	0
ProcessFile	116,044	1,214,860	7,233,970	0
time	116,046	387,430	387,430	0

The `time` function is the 4th highest self-time function. The **Calls** column value shows that it is called once per file loop trip, as often as the `ProcessFile` function. This causes unwanted overhead, which you can reduce by adding the following lines to `gsexample3b`:

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

3.2 Graph View

The Graph view provides a graphical presentation of the application execution. By default, it displays only the critical function path. In this case, it is the path from the start of the thread to the `Store2Load` function:



To control the content of this view, use the filter toolbar at the top of the Graph view window `Show Top: 50 % Recalculate Highlight: None`. Use this toolbar to set your individual filter settings for the analyzed functions. In the **Show Top** drop-down menu, you can select what percentage of functions (5, 10, 20, 50 % or All) you want to view at a time. After you define the percentage, click the **Recalculate** button to view the result. You can also highlight the functions of interest by choosing the criterion from the **Highlight** drop-down menu. This feature is very useful when you analyze an application with many functions. In such cases, the graph can be overwhelming and filtering allows to pinpoint the most time consuming functions.

The Graph and Function Summary views are synchronized with each other. When you select a function in one view, the other view updates to highlight the selected function also. For example:

1. In the **Show Top** drop-down menu, choose 50 (percent) and click **Recalculate**. This will expand the Graph view to display 50% of the highest self-time functions in the application. Note that `ctime` function is displayed in the function summary view, but is hidden in the graph.



2. Double-click `ctime` in the function summary view. The node becomes visible in the graph.

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
nits	?	14	14	0	0	libc.so.6

3. Click the `ProcessFile` node in the graph view. `ProcessFile` becomes the focus function and is automatically highlighted both in the function summary and the graph windows.

3.3 Call List View

Call list view provides full information on the selected (focus) function, its callers and callees in the table format.

To see the call list view, click on the **Call List** tab.

In this view, you can see which functions contributed to the focus function Total time and to what extent. Call list is divided into two windows: caller functions (`main`) and callee functions (`Store2Load`, `ProcessBuffer`, `DoSomeWork`).

To see how this view displays function interaction, double-click `DoSomeWork` to select this function as the focus one.

Callee - a child function that is called by the current function.

Caller - a parent function that calls the current function.



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
libr_rcu_init	1	0	0	0	0		ncexamp

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

The Callers list now shows all functions that call the DoSomeWork function and what percent of overall calls comes from each of them.

Double-click on the ProcessFile function to select it as the focus function. The Store2Load and ProcessBuffer functions are called 10 times for each call to ProcessFile, which causes overhead. One approach to reducing the overhead is to allocate a larger buffer. In fact, you could allocate a buffer large enough to hold the entire file.

3.4 Revise Your Code

In the analysis of your code you found that Store2Load and ProcessBuffer functions are called very often, because the buffer is not large enough. Follow these steps to revise the code, or use the revised code (gsexample3c.c) provided in the opt/intel/vtune/samples/gsexample directory.

1. Open the file gsexample3b.c in the editor you use for developing your application.
2. Open the source of ProcessFile function.



3. To reduce overhead, rewrite the code to allocate the buffer once for the whole file:

```
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. Pass the pointer into the ProcessFile function:

```
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(pbuf, (long) actual);
  close(fd);
  *pCRC = iCRC;
  f = DoSomeWork();}
```

5. Rebuild your application with the same settings you used before and rerun it. See if the number of characters processed per second increase.

4 Analyze Events in Your Code

Use the Sampling wizard to collect specific data on various aspects on the processor events that are happening during data collection. Use the information to improve code performance by reorganizing code to reduce the occurrence of events that stall the processor.

Before analyzing the application, get some more information on the available events. Go to the **VTune Performance Analyzer Reference** online help and find the processor you are interested in under the **Processor Events and Advice** book.


The frequency at which the samples are collected is determined by how often the event is caused by the software running in the system during sampling data collection. The Clockticks event is selected by default.

4.1 Use the Sampling Wizard

Before you start tuning your application, create another benchmark with gsexample3a



Create a new Activity:

1. In the VTune analyzer toolbar, click the **Add Tuning Activity** button, to open the **Select a Wizard** dialog box.
2. From the **Select a Wizard** dialog box, choose the  Sampling Wizard.
3. Click Next to go to the Environment and Activity Settings page.
4. In the **Environment and Activity Settings** page, select Linux* executable collection environment and the Application for the application type. Click **Next** to go to the **Application/Module Profile Settings** page.
5. Configure the following settings:
 - a. in the **Application parameters**, check **Working Directory** and browse to the `gsexample3a` application
 - b. in the **Application arguments**, enter `datafile.txt`.
Click **Next** to go to the **Sampling Collection Settings** page.
6. At the Sampling Collection Settings page, uncheck When the application terminates (before duration completes).
Click Next to go to the Options page.
7. In the Modify Configuration field, configure the following settings:
 - a. check the Change the Sampling Events and set advanced options such as interval, calibration, and delay check box.
 - b. check Run this Activity check box to enable the VTune analyzer run the Activity after you finish configuring it.
8. Click **Finish** to create an Activity and open the **Activity Configuration** dialog box.

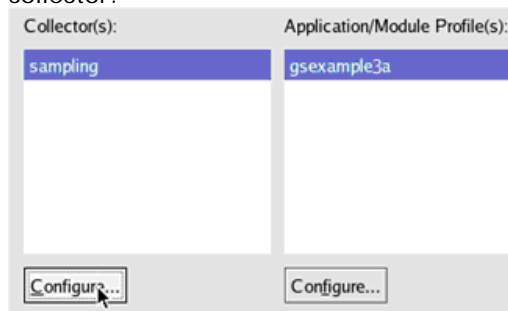
During EBS the collector may run multiple times. In each run, the sampling collector does one of the following:

Calibrates the Sample After value for the selected events. During data collection, data is collected based on the calibrated **Sample After** value

Collects data on the selected events using the default Sample After value

Analyze the application with the Event-Based Sampling (EBS):

1. In the **Activity Duration** field, click the second radio button and enter the value of 10 into the editable box. This makes the Activity run for 10 seconds during the data collection.
2. Click the **Configure** button under the **Collector(s)** window to configure the Sampling collector.

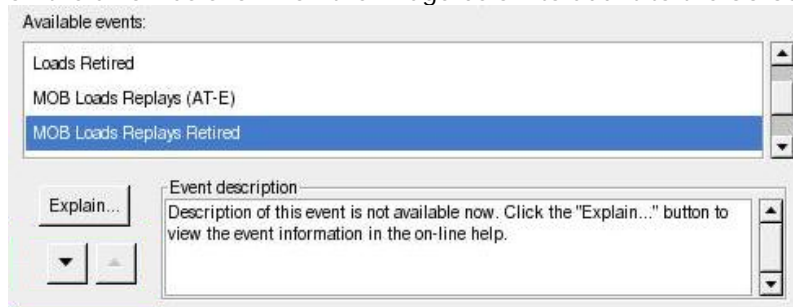


The Configure Sampling dialog box appears.

3. In the **Configure sampling** dialog box, select **Events** tab.



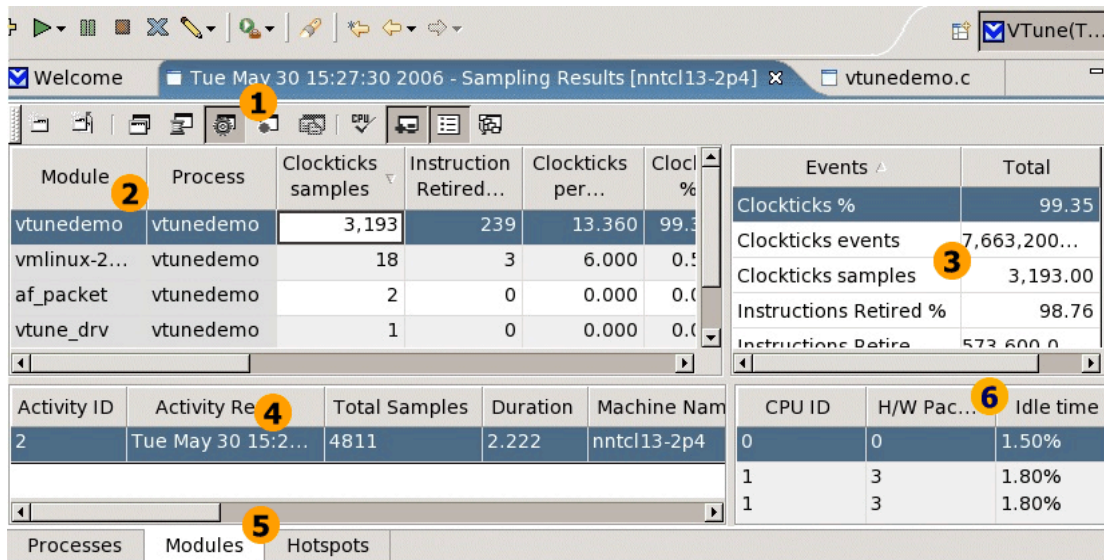
- In the **Available Events** window, select the MOB Loads Replays Retired event. Click on the arrow as shown on the image below to add it to the **Selected Events** window.




- Click **OK** to close the dialog box and start data collection. The VTune analyzer runs the Activity and displays the Sampling results upon its termination.

4.2 View the Sampling Results

Let's look at some of the sampling view features that you can configure to view more interesting data:



1	Click to change the view mode.	2	This column displays the module for which the data was collected.
3	The Selection Summary shows the total value of events and samples, as well as the percentage of samples collected for the selected module. Toggle  view with the Selection Summary button.	4	General information on the sampling data that was collected. Use this pane to see if you have collected enough, but not too many samples.
5	This is the Module view in table format. To view collected data in the horizontal bar chart format, right-click and select View as Bar Chart .	6	Activity Summary by Core. Use this view to see if there are any cores that are not fully utilized.



In this view you can see the following useful information:

- Process view – use this view to see a system-wide view of all the processes running on your system when sampling data was collected. A high number of events in a particular process indicates high CPU usage, which in turn, can indicate potential performance bottlenecks
- Module view - use the information from the Module view to drill down to the hotspots. Modules that were called frequently during sampling data collection are those displayed as ones with the highest number of events or the most CPU time.

Use the sampling results view to locate the problem in the code:

1. Look at the gsexample3a process section of the Process view that opens by default.
2. Switch to Module view by clicking the Module button.
3. Click the gsexample3a module to select it.
4. Click Hotspot button to view the most active functions of the gsexample3a module.
5. You can see that the largest number of events and MOB Loads replays retired occurred at the Store2Load function. This is the problem area of the code.
6. Click the Source view button to see the source view of this function.



4.3 Revise the Code

In the analysis of your code you found a store blocked forward issue. Follow these steps to revise the code, or use gsexample4 provided in the opt/intel/vtune/samples/gsexample directory.

1. Open the file in the editor you use for developing your application.
2. Open the source of Store2Load function.

The code lines of the problem area:

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

The problem is that at the first line we clear the byte within the 32-bit value and then load the 32-bit value. The processor must wait for the write of the byte to complete, before it can load the 32-bit value containing the byte. This causes a blocked store-forward issue.

3. To eliminate the blocked store-forward, you need to write and read data of the same size. Change the code so that the 32-bit value is read, then the first byte of this value is cleared in the 32-bit integers and stored as the 32-bit value.

The revised code section looks as follows:

```
clr = *(pBits + i);  
clr &= 0xffff00ff;  
*(pBits + i) = clr;
```

4. Rebuild your application with the same settings you used before.




4.4 Compare Performance With the Previous Code

To compare the results of the code revised in this step, create another Sampling Activity by repeating the procedure described in step 3a. Use the optimized code or the gsexample3c application provided in /opt/intel/vtune/samples/gsexample directory. After creating and running the Activity look at the results:

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

The results show that the number of the MOB Loads Replays Retired decrease and you can also note the performance increase of the application.

5 Next Steps

You can use the VTune analyzer to filter the compiler optimization information. These reports often contain excellent tuning information. In source view, select the code lines you are interested in and click Compiler Optimization Report  to go directly to the compiler advice for your selected code lines. This feature is supported by the Intel(R) Compiler 9.1, or higher, but it utilizes a standard format open to other compilers. For more information, search the online help for "Compiler Optimization". The training exercise in <install_dir>/vtune/samples is self-contained and pre-compiled so you can try it even if you don't have a compatible compiler installed.

You can use the VTune analyzer for further analysis of your application with the help of data collectors. See the web-based tutorial located at <install_dir>/vtune/training/gv_vtl/index.htm to better understand the product functionality and features.

You can learn about other Intel software development products through the Intel web site at: <http://www.intel.com/software/products/>.

See the product Release Notes document for information about Technical Support and any Limitations that apply to the product.