Software PBX Performance on Intel Multi-Core Platforms - a Study of Asterisk*

White Paper

Order Number: 318862-001US
January 2008

Contents

Figures

Tables

Revision History

Date	Revision	Description
January 2008	001	Initial release.

1.0 Acronyms

Table 1. Acronyms Used in This Document

Acronym	Description
CLI	Command line Interface
codec	Compression/Decompression; Coder-decoder
DSP	Digital Signal Processing
FSB	Front Side Bus
GCC	GNU C/C++ Compiler
ICC	Intel® C/C++ Compiler
iLBC	Internet Low Bitrate Codec
PBX	Private Branch eXchange
PCM	Pulse-Code Modulation
Speex	A Variable Bitrate (VBR) codec which is able to dynamically modify its bitrate to respond to changing network conditions.
SSE	Streaming SIMD Extensions
VoIP	Voice Over Internet Protocol

2.0 Introduction

Private Branch Exchange (PBX) is a private telephone network mainly used within business organizations. In order to achieve adequate performance, the encoding, decoding and transcoding of audio streams within a PBX system has been implemented on specialized, and costly, Digital Signal Processing (DSP) devices.

As more powerful CPUs are available in the market, software-based PBX's have been developed to provide these DSP functions via general-purpose processors. In other words, the software-based PBX solutions actually eliminate the need for extra DSP devices, greatly reducing the cost of building a PBX system.

Since the widespread introduction of multi-core systems to the market, many articles have been written that discuss both the benefits and difficulties of moving the industry from a single-core to a multi-core era. The purpose of this paper is to demonstrate the use of several development tools that can help with this evolution. In particular, this paper will analyze the performance of one of the most popular open-source software PBX: Asterisk* PBX and how to use Intel® software tools to improve it.

Section 3.0 describes the results of our initial performance analysis using several application-appropriate benchmarks, including translation time, encoding time, and number of simultaneous calls. These metrics were captured during the operation of the Asterisk* application (except encoding time which is using Speex encoder); and the applications (Asterisk* and Speex) were compiled first with the GNU C compiler (GCC), and then later with the Intel® C/C++ Compiler (ICC) to create two separate executables from the same code base. We were interested in finding if there was a difference between the optimization capabilities of these two tool chains. Afterwards, both of these executables were tested on systems with a variable amount of cores in order to study the scalability of this application and to see if one tool chain provides better results. With these performance parameters in mind, we were looking to answer the following two questions:

- Does the choice of compiler make a difference in performance?

- Does the application scale well to multiple cores?

One of the most important benchmarks for a PBX system is the maximum number of concurrent calls that a PBX system can handle. If the number of calls exceeds this maximum, then all subsequent calls will be dropped, causing a disruption of services to users. Section 3.3 describes the scalability of this particular metric in the Asterisk* PBX application.

The Asterisk* PBX application is already a multi-threaded application, spawning a thread for each incoming call. But there could be potential to multi-thread the tasks within each connection in order to increase performance further. Section 4.0 describes how to use the VTune™ Performance Analyzer to look for such optimization opportunities in the Asterisk* PBX software while running on an Intel® Architecture-based platform. In addition, we were especially interested in balancing the workload of the application over the available cores, a software architecture that would yield greater thread utilization and, therefore, optimal overall performance. The Intel® Thread Profiler is a perfect tool to examine thread activity and study how they are balanced at run time as described in Section 4.2.

In short, the Intel® development tools can show us exactly what is going on in the system, from both hardware (e.g., CPU counters) and software (e.g., associating computation events with individual lines of code) points of view.

3.0 Performance Analysis

The Intel® C/C++ Compiler (ICC) is known for its high level of optimizations across many different architectures, including the latest Intel® 64, IA-32 and IA-64. Software compiled using the ICC benefits from such advanced features as auto-vectorization (SSE) and inter-procedural optimization (IPO). Indeed, Asterisk* PBX exhibits a performance improvement simply by compiling the application with Intel® C/C++ Compiler.

The following sub-sections discuss the Asterisk* performance with respect to the following criteria:

- Translation time - LESS is better
- Codec encoding time - LESS is better
- Call capacity - MORE is better

The benchmarks were performed on the Asterisk* PBX software, built first with the GNU C compiler and then with the ICC to provide a comparison of tool chain capabilities.

Note: In order to fully utilize Intel® architecture, we used "-xT" flag when building the application with the ICC. The "-xT" flag allows it to generate SSSE3, SSE3, SSE2, and SSE instructions, as appropriate, for Intel® processors; it also enables several other optimizations particular to the Intel® Core™2 Duo processor family. See Appendix B, Appendix C and Appendix D for the build switch and environment settings used for the tests.

3.1 Improving Translation Times with Intel® C/C++ Compiler

Asterisk* supports a large variety of codecs. When Asterisk* starts running, it calculates the translation time between each of its supported audio formats (transcoding time). These calculations can be displayed in the Asterisk* CLI console by typing "`core show translation`." This command generates a table of all codecs and their relative translation times, provided in ms. A higher number in the chart indicates that more work is required to translate between those formats. Of course, if the formats are the same, there is no translation needed, and Asterisk* routes the packets without any extra processing time.

Figure 1 shows the translation times required when Asterisk* is built with the GNU C compiler. It indicates that the most time-consuming codecs are Speex and iLBC. For example, it takes 11 ms to translate every 1 s of a GSM-encoded audio stream to the Speex format while it takes 10 ms for a GSM encoded audio stream to be translated into the iLBC format.

Figure 1. Translation Times - Asterisk* Built with GNU C Compiler

```
*CLI> core show translation
         Translation times between formats (in milliseconds) for one second of data
          Source Format (Rows) Destination Format (Columns)

          g723 gsm ulaw alaw g726aal2 adpcm slin lpc10 g729 speex ilbc g726 g722
     g723    -   -    -    -        -     -    -     -    -     -    -    -    -
      gsm    -   -    2    2        2     2    1     3    -    11   10    2    -
     ulaw    -   2    -    1        2     2    1     3    -    11   10    2    -
     alaw    -   2    1    -        2     2    1     3    -    11   10    2    -
 g726aal2    -   2    2    2        -     2    1     3    -    11   10    1    -
    adpcm    -   2    2    2        2     -    1     3    -    11   10    2    -
     slin    -   1    1    1        1     1    -     2    -    10    9    1    -
    lpc10    -   2    2    2        2     2    1     -    -    11   10    2    -
     g729    -   -    -    -        -     -    -     -    -     -    -    -    -
    speex    -   2    2    2        2     2    1     3    -     -   10    2    -
     ilbc    -   2    2    2        2     2    1     3    -    11    -    2    -
     g726    -   2    2    2        1     2    1     3    -    11   10    -    -
     g722    -   -    -    -        -     -    -     -    -     -    -    -    -
*CLI>
```

In contrast, by building Asterisk* with the ICC, the translation time reduces immediately - and with no code changes. Figure 2 shows that it now takes 9 ms for 1 s of a GSM audio stream to be translated into the iLBC format, a 10 percent time savings.

Figure 2. Translation Times - Asterisk* Built with Intel® Compiler

```
*CLI> core show translation
         Translation times between formats (in milliseconds) for one second of data
          Source Format (Rows) Destination Format (Columns)

          g723 gsm ulaw alaw g726aal2 adpcm slin lpc10 g729 speex ilbc g726 g722
     g723   -    -    -    -        -     -    -     -    -     -    -    -    -
      gsm   -    -    2    2        2     2    1     2    -    11    9    2    -
     ulaw   -    2    -    1        2     2    1     2    -    11    9    2    -
     alaw   -    2    1    -        2     2    1     2    -    11    9    2    -
 g726aal2   -    2    2    2        -     2    1     2    -    11    9    1    -
    adpcm   -    2    2    2        2     -    1     2    -    11    9    2    -
     slin   -    1    1    1        1     1    -     1    -    10    8    1    -
    lpc10   -    2    2    2        2     2    1     -    -    11    9    2    -
     g729   -    -    -    -        -     -    -     -    -     -    -    -    -
    speex   -    2    2    2        2     2    1     2    -     -    9    2    -
     ilbc   -    2    2    2        2     2    1     2    -    11    -    2    -
     g726   -    2    2    2        1     2    1     2    -    11    9    -    -
     g722   -    -    -    -        -     -    -     -    -     -    -    -    -
*CLI>
```

The improvement is not consistent across the board. For example, it seems that there is no improvement for the GSM to Speex translation. But perhaps this measurement is not precise enough to identify all differences between the applications performance built with the two tool chains (i.e., since there is a maximum of 10 percent resolution). In this case, the next section will demonstrate a finer-grained set of tests for the Speex encoder.

3.2 Improving Encoding Times with Intel® C/C++ Compiler

Speex is a free software codec for speech used often in VoIP and file based compression algorithms. Out of all the codecs supported by Asterisk* PBX, we chose Speex for this test because it is complex enough to provide meaningful results and it is a software in the open domain.

After building and installing the Speex codec library and encoding/decoding tool on our test platform(s), we use the Linux* time utility to display how long it takes to perform the requested audio encoding. Specifically, the encoding time for an audio wave file can be shown by typing "`time -v speexenc <original.wav> <translated.spx>`" in the Linux* shell.

Table 3 illustrates the test results of measuring the time for Speex to encode the wave file described in Table 2 on an Intel® dual-core platform. The GCC-compiled application takes 1.49 s to encode while the ICC-compiled application takes only 1.33 s, a performance increase of 8 percent.

Table 2. Audio File for Speex Encoder

Input file for Speex Encoder	PlayTime	Format	Sampling Rate	Channels	File Size	Encoded File Size
Wave File	102 s	Wave	8 kHz/16-bit	1	1637802 B	201108 B

Table 3. Encoding Time for Speex Encoder on Intel® Core™2 Duo Mobile Processor T7400 at 2.16 GHz

# of core in Intel® Core™2 Duo Mobile processor T7400 at 2.16 GHz	1 (GCC)	1 (ICC)	2 (GCC)	2 (ICC)	Normalized to 2 GHz 1 (GCC)	Normalized to 2 GHz 1 (ICC)
Encoding time for 102 s play time	1.38 s	1.24 s	1.38 s	1.30 s	1.49 s	1.33 s

Because the Speex encoder is serialized code (like most codecs associated with the Asterisk* PBX application), it does not take full advantage of the multi-core system available to it during this test. In other words, the encoding time will remain almost the same regardless of the number of cores used.

Note: To perform this test with only one core enabled, go to your system BIOS and select the “disable” option for “core multi-processing” on the “configure advanced CPU settings” page.

3.3 Improving Call Capacity with ICC and Multi-core Systems

The following sub-sections detail the performance improvements of building the Asterisk* PBX application with the Intel® C/C++ Compiler and running it on Intel® multi-core platforms. In particular, we would like to determine if the call capacity of the application can scale with the addition of more cores to the system. And throughout this section, as in the others, we run each test twice, first with a GNU-compiled version of the executable and then with an ICC-compiled version.

3.3.1 Measuring Call Capacity with Astertest

Astertest is a Microsoft Windows* application designed to stress test systems running Asterisk* PBX software. This testing required the following hardware equipment:

1. Two Linux* systems
    a. One acting as an Asterisk* origination server
    b. One acting as an Asterisk* test server
2. One Microsoft Windows XP* system hosting the Astertest test application

When the test begins, the Astertest application will log onto both the origination server and test server through an account created in the `managers.conf` file residing on both systems under `/etc/asterisk/`. Astertest then controls the origination server to automatically make calls to the test server. At the same time, Astertest directs the test server to answer each call and playback a default GSM audio file.

For this test, we set up the system for GSM->iLBC transcoding be performed (see Appendix F), meaning that the test server will decode GSM data into linear PCM data and then encode this PCM data into iLBC data before sending it back to origination server as the playback call. This procedure will require tremendous computing power on both the origination and test servers. When the dual quad-core system used in our tests reaches its maximum capacity of 748 calls, the test server will not be able to complete any more calls as requested from the origination server (see Figure 3 below). Further, we found that to achieve this maximum number of calls - without wasting resources - the origination server and test server must be identical in their hardware and software configuration.

Figure 3. Astertest on Dual Quad-core at 2 GHz, GSM->iLBC Transcoding (ICC Compiler)

We measured the maximum call capacity and CPU load across a range of Intel® multi-core platforms, where a varying number of cores were available/enabled:

- 8-core system at 2.00 GHz
- 4-core system at 2.33 GHz
- 2-core system at 2.16 GHz
- 1-core system at 2.16 GHz

See Appendix A for the actual system configurations.

Later, we replicated the tests with the Asterisk* system setup for GSM->Speex transcoding to see if there was similar behavior across codecs. Results are shown in Table 4, Table 5, Table 6, and Table 7.

Table 4. Maximum Playback Calls on 8-core System at 2 GHz

Compiler	GSM->iLBC	GSM->Speex
GNU compiler (GCC 4.1.2)	673 calls	552 calls
Intel® compiler (ICC 9)	748 calls	590 calls

Table 5. Maximum Playback Calls on 4-core System at 2.33 GHz

Compiler	GSM->iLBC	GSM->Speex	Normalized to 2 GHz GSM->iLBC	Normalized to 2 GHz GSM->Speex
GNU compiler (GCC 4.1.2)	428 calls	345 calls	367 calls	296 calls
Intel® compiler (ICC 9)	482 calls	376 calls	413 calls	322 calls

Table 6. Maximum Playback Calls on 2-core System at 2.16 GHz

Compiler	GSM->iLBC	GSM->Speex	Normalized to 2 GHz GSM->iLBC	Normalized to 2 GHz GSM->Speex
GNU C compiler (GCC 4.1.2)	198 calls	160 calls	183 calls	148 calls
Intel® compiler (ICC 9)	228 calls	181 calls	211 calls	168 calls

Table 7. Maximum Playback Calls on 1-core at 2.16 GHz (Disable One Core in 2-core System)

Compiler	GSM->iLBC	GSM->Speex	Normalized to 2 GHz GSM->iLBC	Normalized to 2 GHz GSM->Speex
GNU C compiler (GCC 4.1.2)	97 calls	82 calls	90 calls	76 calls
Intel® compiler (ICC 9)	117 calls	89 calls	108 calls	82 calls

Each software configuration was tested using both the GNU and ICC compilers. Table 4 shows the maximum number of playback calls that can be handled by the Asterisk* server improved 11 percent for GSM->iLBC transcoding and 6.8 percent for GSM->Speex transcoding when the ICC-compiled executable was used.

Figure 4 gives another view of the data from Table 4, Table 5, Table 6, and Table 7. Specifically, it allows you to easily see that the ICC-compiled version of the application is superior, regardless of the number of cores in the platform.

Figure 4. Numbers of Calls with Respect to Number of Cores for GSM->iLBC and GSM->Speex

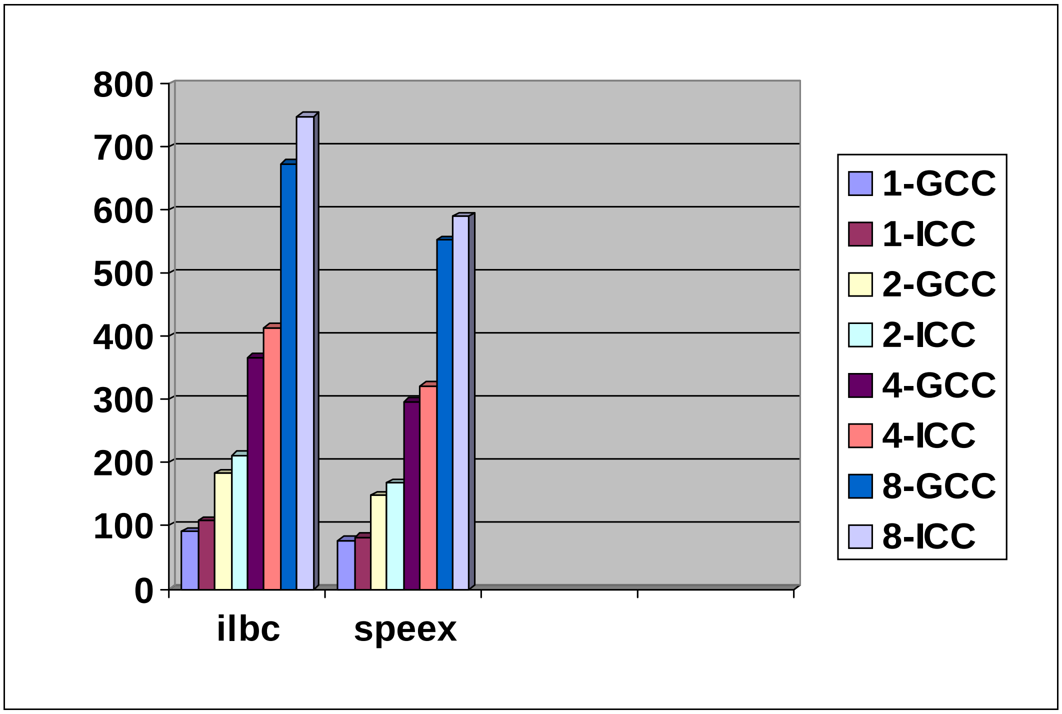


Table 8 summarizes the performance difference between the Asterisk* PBX applications compiled with the GNU versus the ICC compiler. In short, the ICC compiler seems to better utilize Intel® architecture, with performance improvements ranging from 6.8 percent to 20 percent.

Table 8. Performance Improvement Between Intel® Compiler and GNU Compiler

Number of Cores in Test	GSM->iLBC	GSM->Speex
1	20%	7.8%
2	15.3%	13.5%
4	12.5%	8.7%
8	11%	6.8%

As mentioned in the Note at the beginning of Section 3.0, we used the "-xT" flag to generate SSE instructions when compiling Asterisk* with ICC. For IA-32 architecture, SSE contains eight 128-bit registers known as XMM0 through XMM7 while the x64 extensions added eight more registers, XMM8 through XMM15. By disassembling and investigating the object file of `iCBSearch` function, we found that GCC 4.1.2 20061115 in the openSUSE* 10.2 x86-64 distribution does not utilize registers XMM8-XMM15. That is one of the reasons why ICC provides better performance than GCC. The GCC compiler does recognize that CPU is x86-64 architecture, but it does not make as good of use of the hardware features as ICC does (in our case, the XMM8–XMM15 registers).

3.3.2 Determining Application Scaling

Our last set of tests before more in-depth analysis using the Intel® VTune™ Performance Analyzer is designed to measure application scaling on a given multi-core platform as we allocate more and more of its cores to the task.

The first test machine is a dual-socket, dual-core platform with Dual-Core Intel® Xeon® processors 5140 running at 2.33 GHz. The CPU's Power Utility allows us to enable or disable each one of the four cores from executing the application at any given time. For example, to disable core 0 processor from running the Asterisk* application, we used the following command:

```
"nice -n -20 WoodcrestMaxPowerIA32e -core0"
```

By changing the parameters of this command, we can study how the number of cores enabled in the system affects the maximum call capacity. Table 9 lists the maximum number of calls that can be handled on the 4-core system.

Table 9. Maximum Calls on 4-core System

# of Core (Dual-Core Intel® Xeon® Processor 5140 at 2.33 GHz)	# of Calls (ICC GSM->iLBC)	Normalized to 2 GHz # of Calls (ICC GSM->iLBC)	Normalized Index
1 (cores 0–2 disabled)	130	111	1
2 (cores 0–1 disabled)	262	225	2.01
3 (core 0 disabled)	375	322	2.88
4 (no cores disabled)	478	410	3.67

Similarly, we use the CPU Power Utility on our second test machine, a dual-socket, quad-core platform with Quad-Core Intel® Xeon® processors E5335 running at 2.0 GHz to enable and disable each one of its eight cores, in turn. For example, we disable core 0 by typing the following command:

```
"nice -n -20 ClovertownMaxPowerIA32e -core0".
```

Table 10. Maximum Calls on 8-core System

# of Core (Quad-Core Intel® Xeon® Processor E5335 at 2 GHz)	# of Calls (ICC GSM->iLBC)	Normalized Index
1 (cores 0–6 disabled)	110	1
2 (cores 0–5 disabled)	247	2.25
3 (cores 0–4 disabled)	338	3.07
4 (cores 0–3 disabled)	444	4.03
5 (cores 0–2 disabled)	538	4.89
6 (cores 0–1 disabled)	623	5.66
7 (core 0 disabled)	693	6.3
8 (no cores disabled)	769	6.99

Comparing Table 10 to Table 9, the result of the 1-core case is basically the same. However, it is interesting to note that in the 2- to 4-core cases, there is a better scaling factor in the 8-core (dual quad-core) system compared to the 4-core (dual dual-core) system. While the FSB speeds and other components are the same on both systems, the 8-core system contains 8 MB L2 cache and the 4-core system contains 4 MB L2 cache. Apparently, the larger L2 cache makes the significant performance difference in these cases.

4.0 Analyzing Asterisk* Using Intel® Tools

Asterisk* documentation states that the software architecture is set up so that a new thread is started for each active call. Our tests maxed out at about 100 calls per core, meaning that there are about 100 corresponding threads per core. With this in mind, consider the following cases for increasing the call capacity as measured by Astertest:

1. In a single core system, we would have to improve the performance of the iLBC/Speex encoders. And since these are written as single-threaded, serialized code, we would need to actually overhaul the algorithm, an effort which is beyond the scope of this publication.
2. In a multi-core system, performance of iLBC/Speex encoders can be improved by threading. In other words, we can aim to develop a parallelized version of each encoder for a multi-core system.

Following the multi-core case, we will use both the Intel® VTune™ Performance Analyzer and the Intel® Threading Tools to see if the threading of the encoders is possible and the best way to utilize those tools. The following subsections all use the ICC-compiled version of the Asterisk* PBX application.

4.1 Intel® VTune™ Performance Analyzer

We used the Intel® VTune™ Quick Performance Analysis wizard (QPA) to perform our application analysis. More specifically, we used the wizard defaults to capture run-time data via both the "Sampling" and "Call Graph" tools (noting that "Counter Monitor" data collection is only supported when analyzing an application running in a Microsoft Windows* environment).

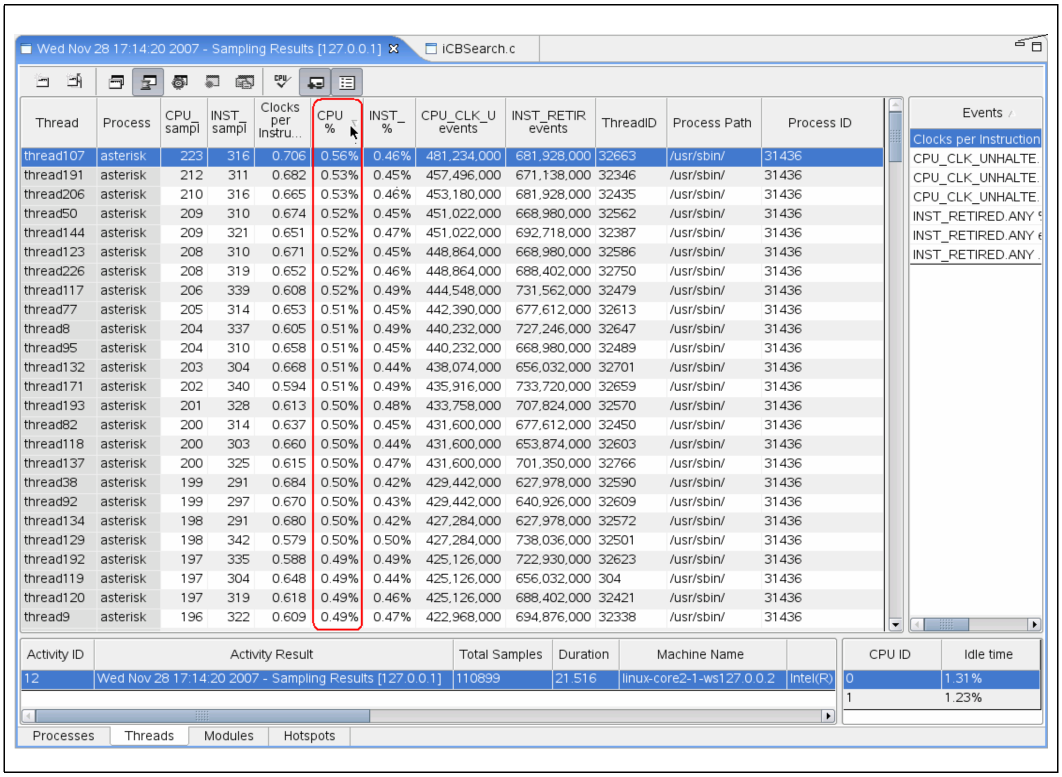
The following sections describe the data that we collected, and provide commentary with an eye towards potential multi-threading opportunities.

4.1.1 Finding Hot-spots with Sampling Tool

The Sampling Tool of the Intel® VTune™ Performance Analyzer is a good way to correlate CPU events with specific areas of application code. One of the default metrics that it captures is CPU_CLK; this metric is helpful in identifying the areas of code in which the application is spending most of its time.

On a dual-core-based test run with 200 calls, there are hundreds of threads, and Figure 5 shows that each thread takes 0.49%~0.56% of the total CPU_CLK time during this particular run. This value varies among threads because every thread is spawned in different order, and therefore, a given run of the application may yield slightly more work for one thread versus another at any given time. The sampling results can also be expanded on a function-by-function basis which shows that the `iCBSearch` function takes about 56.83% of the CPU_CLK time across all operating threads.

Figure 5. Intel® VTune™ Analyzer Sampling Results - Display Threads

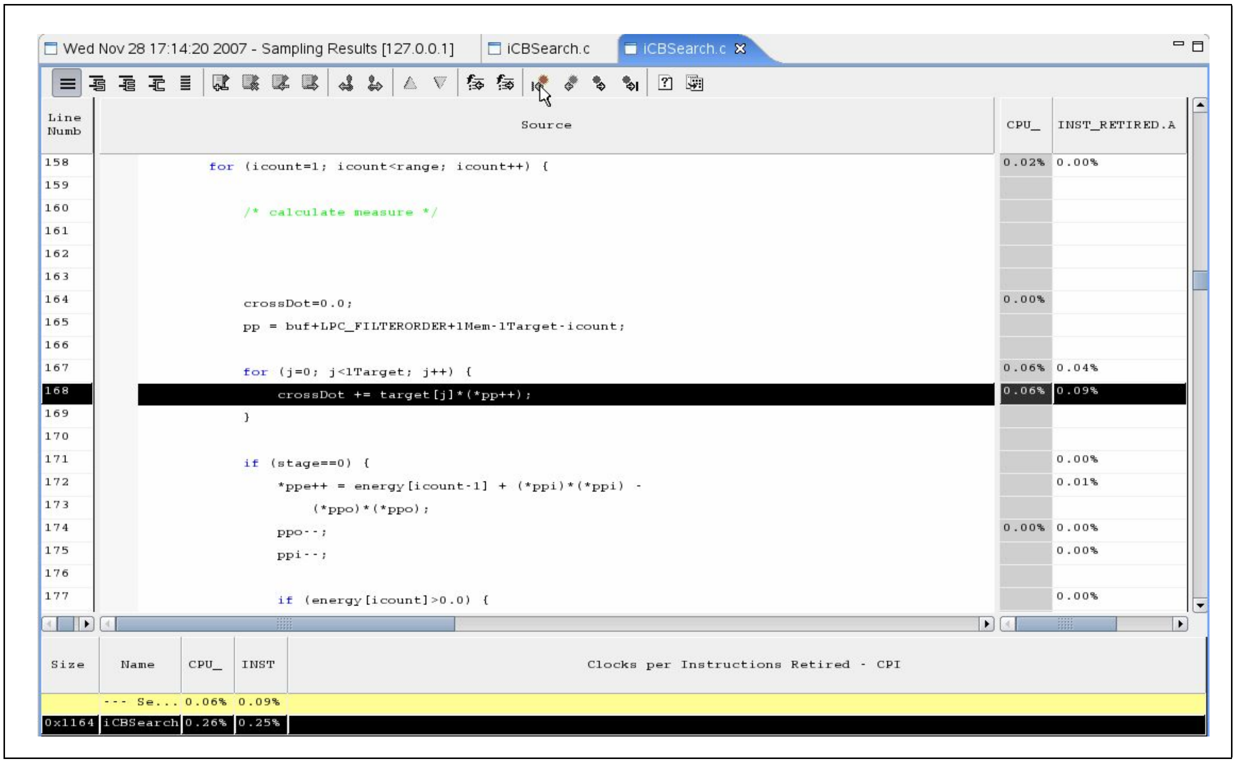


Thread	Process	CPU_ sampl	INST_ sampl	Clocks per Instru...	CPU %	INST_ %	CPU_CLK_U events	INST_RETIR events	ThreadID	Process Path	Process ID
thread107	asterisk	223	316	0.706	0.56%	0.46%	481,234,000	681,928,000	32663	/usr/sbin/	31436
thread191	asterisk	212	311	0.682	0.53%	0.45%	457,496,000	671,138,000	32346	/usr/sbin/	31436
thread206	asterisk	210	316	0.665	0.53%	0.46%	453,180,000	681,928,000	32435	/usr/sbin/	31436
thread50	asterisk	209	310	0.674	0.52%	0.45%	451,022,000	668,980,000	32562	/usr/sbin/	31436
thread144	asterisk	209	321	0.651	0.52%	0.47%	451,022,000	692,718,000	32387	/usr/sbin/	31436
thread123	asterisk	208	310	0.671	0.52%	0.45%	448,864,000	668,980,000	32586	/usr/sbin/	31436
thread226	asterisk	208	319	0.652	0.52%	0.46%	448,864,000	688,402,000	32750	/usr/sbin/	31436
thread117	asterisk	206	339	0.608	0.52%	0.49%	444,548,000	731,562,000	32479	/usr/sbin/	31436
thread77	asterisk	205	314	0.653	0.51%	0.45%	442,390,000	677,612,000	32613	/usr/sbin/	31436
thread8	asterisk	204	337	0.605	0.51%	0.49%	440,232,000	727,246,000	32647	/usr/sbin/	31436
thread95	asterisk	204	310	0.658	0.51%	0.45%	440,232,000	668,980,000	32489	/usr/sbin/	31436
thread132	asterisk	203	304	0.668	0.51%	0.44%	438,074,000	656,032,000	32701	/usr/sbin/	31436
thread171	asterisk	202	340	0.594	0.51%	0.49%	435,916,000	733,720,000	32659	/usr/sbin/	31436
thread193	asterisk	201	328	0.613	0.50%	0.48%	433,758,000	707,824,000	32570	/usr/sbin/	31436
thread82	asterisk	200	314	0.637	0.50%	0.45%	431,600,000	677,612,000	32450	/usr/sbin/	31436
thread118	asterisk	200	303	0.660	0.50%	0.44%	431,600,000	653,874,000	32603	/usr/sbin/	31436
thread137	asterisk	200	325	0.615	0.50%	0.47%	431,600,000	701,350,000	32766	/usr/sbin/	31436
thread38	asterisk	199	291	0.684	0.50%	0.42%	429,442,000	627,978,000	32590	/usr/sbin/	31436
thread92	asterisk	199	297	0.670	0.50%	0.43%	429,442,000	640,926,000	32609	/usr/sbin/	31436
thread134	asterisk	198	291	0.680	0.50%	0.42%	427,284,000	627,978,000	32572	/usr/sbin/	31436
thread129	asterisk	198	342	0.579	0.50%	0.50%	427,284,000	738,036,000	32501	/usr/sbin/	31436
thread192	asterisk	197	335	0.588	0.49%	0.49%	425,126,000	722,930,000	32623	/usr/sbin/	31436
thread119	asterisk	197	304	0.648	0.49%	0.44%	425,126,000	656,032,000	304	/usr/sbin/	31436
thread120	asterisk	197	319	0.618	0.49%	0.46%	425,126,000	688,402,000	32421	/usr/sbin/	31436
thread9	asterisk	196	322	0.609	0.49%	0.47%	422,968,000	694,876,000	32338	/usr/sbin/	31436

Activity ID	Activity Result	Total Samples	Duration	Machine Name	
12	Wed Nov 28 17:14:20 2007 - Sampling Results [127.0.0.1]	110899	21.516	linux-core2-1-ws127.0.0.2	Intel(R)

Double-click on the `iCBSearch` function, and the GUI will display the source code as shown in Figure 6. Within the `iCBSearch` function, the hot-spot (the area of code that has the highest value of a particular metric) is at line 168 (0.06%). But this does not actually help us too much because Line 168 is a single line in a loop, and there are data dependencies in the loop that prohibit multi-threading. Besides, the CPU_CLK percentage for the line is only 0.06%, which is too small to be an effective place to improve performance.

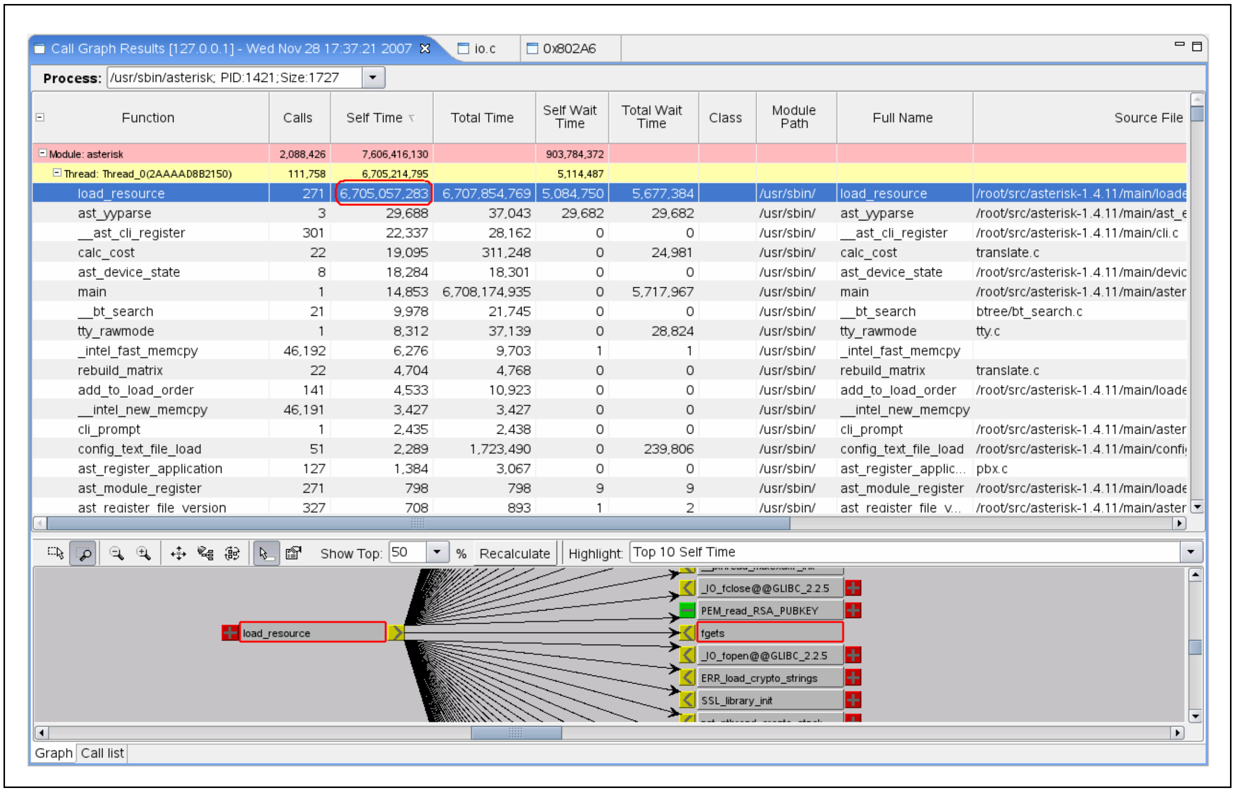
Figure 6. Hot-spot in `iCBSearch` Line 168



4.1.2 VTune™ Call Graph

In the call graph, Self Time is defined as the time spent inside a function. This includes the time spent waiting between execution activities. For the call graph of the first hot-spot in initial thread in Figure 7, the maximum “Self Time” is the function `load_resource`. Since this thread only runs once to load and call other system functions, there is no advantage to optimize this thread.

Figure 7. VTune™ Call Graph - `load_resource` Has Maximum Self Time in First Init Thread



Function	Calls	Self Time	Total Time	Self Wait Time	Total Wait Time	Class	Module Path	Full Name	Source File
Module: asterisk	2,088,426	7,606,416,130		903,784,372					
Thread: Thread_0(2AAAA08B2150)	111,758	6,705,214,795		5,114,487					
load_resource	271	6,705,057,283	6,707,854,769	5,084,750	5,677,384		/usr/sbin/	load_resource	/root/src/asterisk-1.4.11/main/loade
ast_yyparse	3	29,688	37,043	29,682	29,682		/usr/sbin/	ast_yyparse	/root/src/asterisk-1.4.11/main/ast_e
__ast_cli_register	301	22,337	28,162	0	0		/usr/sbin/	__ast_cli_register	/root/src/asterisk-1.4.11/main/cli.c
calc_cost	22	19,095	311,248	0	24,981		/usr/sbin/	calc_cost	translate.c
ast_device_state	8	18,284	18,301	0	0		/usr/sbin/	ast_device_state	/root/src/asterisk-1.4.11/main/devic
main	1	14,853	6,708,174,935	0	5,717,967		/usr/sbin/	main	/root/src/asterisk-1.4.11/main/aster
__bt_search	21	9,978	21,745	0	0		/usr/sbin/	__bt_search	btree/bt_search.c
tty_rawmode	1	8,312	37,139	0	28,824		/usr/sbin/	tty_rawmode	tty.c
_intel_fast_memcpy	46,192	6,276	9,703	1	1		/usr/sbin/	_intel_fast_memcpy	
rebuild_matrix	22	4,704	4,768	0	0		/usr/sbin/	rebuild_matrix	translate.c
add_to_load_order	141	4,533	10,923	0	0		/usr/sbin/	add_to_load_order	/root/src/asterisk-1.4.11/main/loade
__intel_new_memcpy	46,191	3,427	3,427	0	0		/usr/sbin/	__intel_new_memcpy	
cli_prompt	1	2,435	2,438	0	0		/usr/sbin/	cli_prompt	/root/src/asterisk-1.4.11/main/aster
config_text_file_load	51	2,289	1,723,490	0	239,806		/usr/sbin/	config_text_file_load	/root/src/asterisk-1.4.11/main/confi
ast_register_application	127	1,384	3,067	0	0		/usr/sbin/	ast_register_applic...	pbx.c
ast_module_register	271	798	798	9	9		/usr/sbin/	ast_module_register	/root/src/asterisk-1.4.11/main/loade
ast_register_file_version	327	708	893	1	2		/usr/sbin/	ast_register_file_v...	/root/src/asterisk-1.4.11/main/aster

The second hot-spot is the `accept_thread` function shown in Figure 8. As this function is similar to the first hot-spot that we discussed above, further optimization may not be beneficial.

Figure 8. VTune™ Call Graph - `accept_thread` has Maximum Self Time in the Second Init Thread

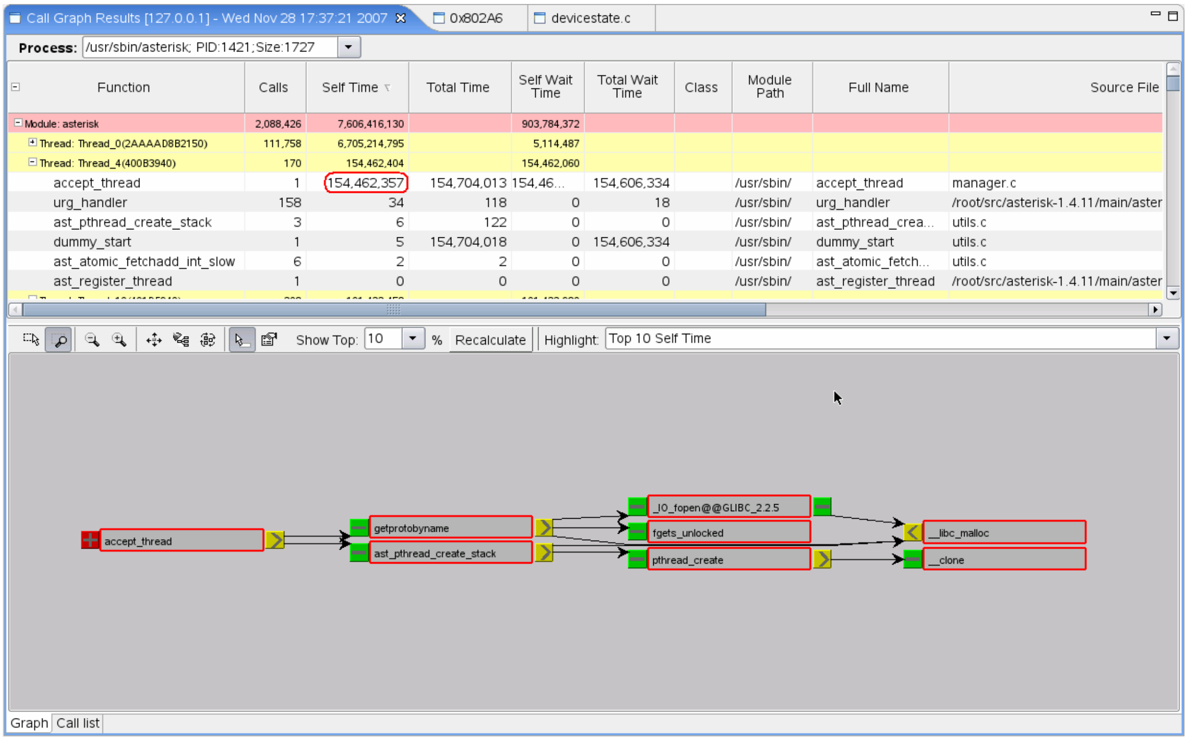
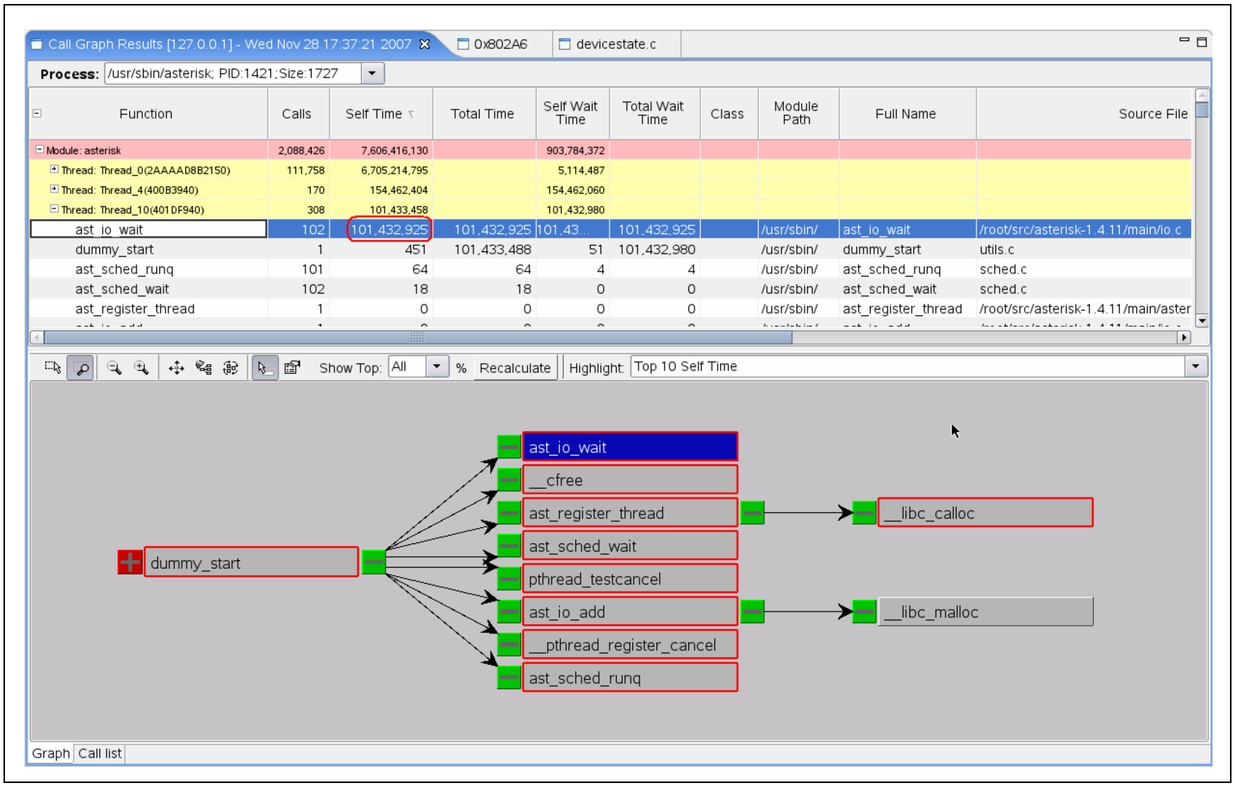


Figure 9 shows the maximum Self Time spent is in the `ast_io_wait` which is a polling function. The transcoding thread spends most of its time waiting for an input GSM audio stream. When the transcoding thread is active, it will decode the GSM audio stream and encode it to iLBC or Speex format. The encoder takes up most of the active time for a transcoding thread, leaving the rest of threads waiting to be active in a heavy load system. It does not make sense to multi-thread a function waiting for I/O.

Figure 9. VTune™ Call Graph - `ast_to_wait` Has Maximum Self Time in Transcoding Thread



4.2 Intel® Thread Profiler

Intel® Thread Profiler 3.1 for Microsoft Windows* provides a graphical view of thread activities during our Asterisk* testing and can, among other things give us visual confirmation that for each playback call, Asterisk* creates one thread for transcoding VoIP packets in the test server.

Figure 10 is a capture of thread behavior between 35 and 95 s of our callback test. Every 3 s, Astertest initiates a new call from the origination server to the test server. For example, at time 38.5, one thread was created for a playback call and then at time 41.5 another thread was created for the next playback call. During time 91 to 93, the test hangs up all calls and the associated transcoding threads are terminated.

Note: The Intel® Thread Profiler also needs many system resources to perform its work of recording thread activities, so we do not see the system reach the maximum call capacity that we measured earlier in Section 3.0.

Figure 10. Create/Terminate Threads for Playback Call Test

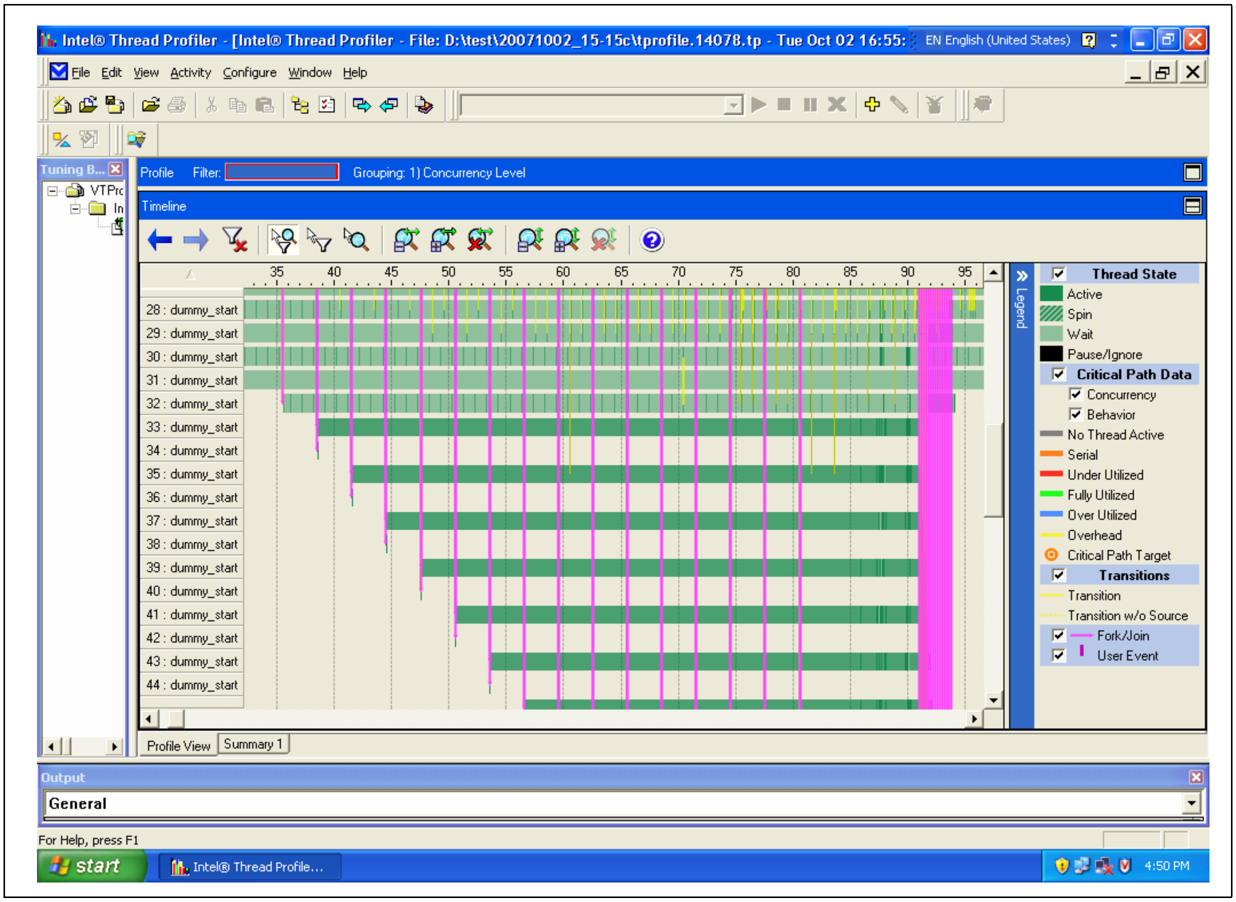


Table 2 showed us that for the Speex codec running on a single core 2 GHz machine, every transcoding thread is active for 13 ms out of every s of wall time (1.33/102). From this, we can extrapolate that every 2 GHz core in the system can support up to 77 calls (1/13x10-3), which we note is very close to the 76 calls that we measured for Table 7. This is further proof that the CPU clock cycles are almost all consumed by encoding in the Astertest testing environment, not by GSM decoding or network loading.

The CPU resource is allocated over hundreds of threads. From the perspective of Asterisk*, each encoding thread is intrinsic in the multi-user system and trying to parallelize each thread may not improve the overall system throughput. From the perspective of serialized codec in Asterisk*, threading the codec may help to increase the performance for a single input audio stream in multi-core system, but it would also increase the task-switching overhead in the case of multiple input audio streams.

5.0 Conclusion

In order to take full advantage of a multi-core system, the software running on it must partition its tasks to run on all the available computing resources. This study showed that for a multi-threaded application such as Asterisk*, which creates a new thread for every incoming call, it is easy to achieve performance increases simply by increasing the number of processor cores; the operating system will schedule the threads appropriately.

We found early on in our study that rebuilding the Asterisk* application with the Intel® C++ compiler (and running it on Intel® multi-core platforms) boosted its performance with no code changes. In other words, the ICC seems to make the best use of the hardware facilities intrinsic in Intel® architecture and thus provides a better-performing solution with minimal effort on the part of the developer. So, if you are looking for a potential performance increase of 5–10%, you should consider using the ICC in your development environment.

Using several different testing methods, we found that performance of the Asterisk* PBX application increased proportionally as more cores were made available to it. Given that the maximum call capacity on the 8-core system was almost seven times that of the single core case, we can conclude such increases in performance come without significant overhead. In fact, the increases seem to be almost linear as each multi-core configuration was tested. This indicates excellent scalability for Asterisk* and confirms that many other similarly multi-threaded software applications would benefit greatly by running on Intel® multi-core systems.

We are careful to note that not all existing applications will have such great scalability factors without going through a massive code modification effort. If you are in the position to design your software architecture from the ground up, we recommend that you consider the scalability of the algorithm to be of high importance. It is likely that multiple core machines will be available to your application at some point in the future, and some design-time up front will allow for minimal effort later on when optimizing performance for such a migration.

6.0 Related Links

Asterisk*: The Open Source Telephony Platform http://www.asterisk.org/

Astertest: Asterisk* stress testing tool http://astertest.com/downloads/

Asterisk* Performance: building your system for performance and scalability http://astertest.com/astricon_performance.ppt

Stress Testing Asterisk* with Astertest: http://www.asteriskguru.com/tutorials/astertest.html

Intel® Software Products: http://www.intel.com/software/products

Intel® Software Network: http://softwarecommunity.intel.com/isn/home/

Intel® Press: http://www.intel.com/intelpress/sum_swcb2.htm

Appendix A System Setup

To complete the test we need to have an Asterisk* PBX server running Linux* that originates the calls and another Asterisk* server to be tested. We also need a Microsoft Windows XP* system to run the Astertest.

Hardware:

- 2-core platform
  Arbuckle Mountain CRB with one Intel® Core™2 Duo Mobile processor T7400 at 2.16 GHz

- 4-core platform
  Intel® Server Board 5000XSL CRB with dual Dual-Core Intel® Xeon® processors 5140 at 2.33 GHz
- 8-core platform
  Intel® Server Board 5000XSL CRB with dual Quad-Core Intel® Xeon® processors E5335 at 2.00 GHz

Software:

- openSUSE* Linux* 10.2 x86-64
- Intel® C++ compiler for Linux* version 10.0.023
- Intel® Thread Profiler 3.1 for Linux*
- VTune™ Performance Analyzer 9.0 Update 1 for Linux*
- Asterisk* version 1.4.11
- Astertest
- Speex codec version 1.2beta2
- Dual-Core Intel® Xeon® processor 5140 Power Utility - Linux* Version
- Quad-Core Intel® Xeon® processor E5335 Power Utility - Linux* Version

Appendix B Environment Setting

Apply the following environment setting by source command

```
#!/bin/sh

# Linux 2.6 tuning script for Asterisk

ulimit -n 655360
```

Add the following to `/etc/sysctl.conf` to set the maximum number of sockets.

```
#ulimit -n 655360

# max open files

fs.file-max=163840

# kernel threads

kernel.threads-max=163840


# socket buffers

net.core.wmem_default=655360

net.core.wmem_max=5242880

net.core.rmem_default=655360

net.core.rmem_max=5242880


# netdev backlog

net.core.netdev_max_backlog=4096
```

```
# socket buckets

net.ipv4.tcp_max_tw_buckets=163840

# port range

net.ipv4.ip_local_port_range=10240 65000
```

The above environment setting and modification of `/etc/sysctl.conf` are needed, as the default setting will limit the number of calls not to go beyond 333. The number of open files in Operating System should be changed for several hundreds of calls in Asterisk*. Please check http://astertest.com/astricon_performance.ppt slide 27 “OS tweaks” for more detail.

Appendix C Build Asterisk* with GNU C Compiler

The following configuration is used to build Asterisk* with GNU C compiler:

```
./configure CC="gcc -O3" CFLAGS=-O3
```

The setup for modules from Astertest and modification for configuration files can be found from http://www.asteriskguru.com/tutorials/astertest.html.

Appendix D Build Asterisk* with Intel® C++ Compiler

To build Asterisk* with Intel® C++ compiler, the DEBUG definition in Make file need to be removed

```
# Include debug and macro symbols in the executables (-g) and profileing info (-pg)

## DEBUG=-g3 (removed for Intel compiler)
```

The following configuration is used to build Asterisk* with Intel® C++ compiler:

```
./configure CC="icc -O3 -xT" CXX=icpc CFLAGS="-O3 -xT"
```

Appendix E Build and Install Speex

The following configuration is for building Speex with respect to using Intel® C++ compiler and GNU Compiler:

```
./configure CFLAGS=-O3                (gcc)

./configure CC=icc CFLAGS="-O3 -xT"   (icc)
```

To build and install Speex library:

```
make

make install
```

Appendix F Launch Astertest

The following screen capture shows the procedure of launching a new playback test.

Figure 11. Astertest - Start a New Test

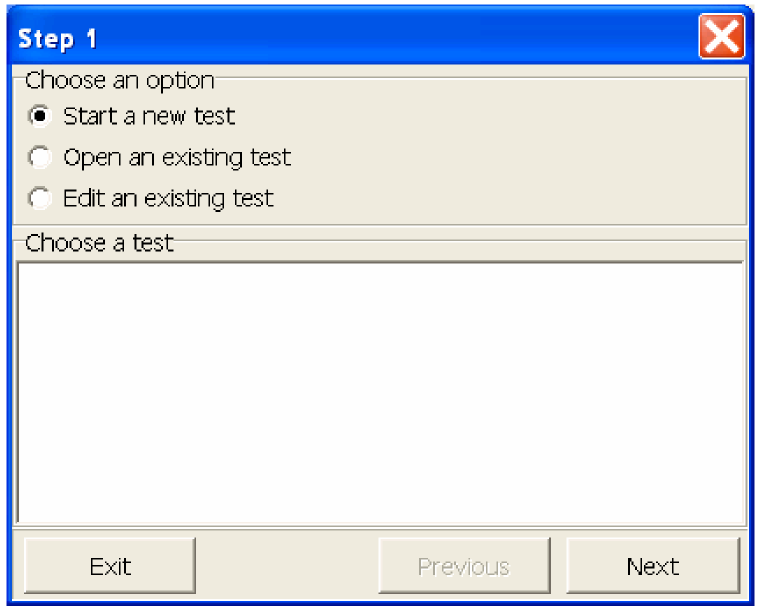


Figure 12. Astertest - Choose Playback Test

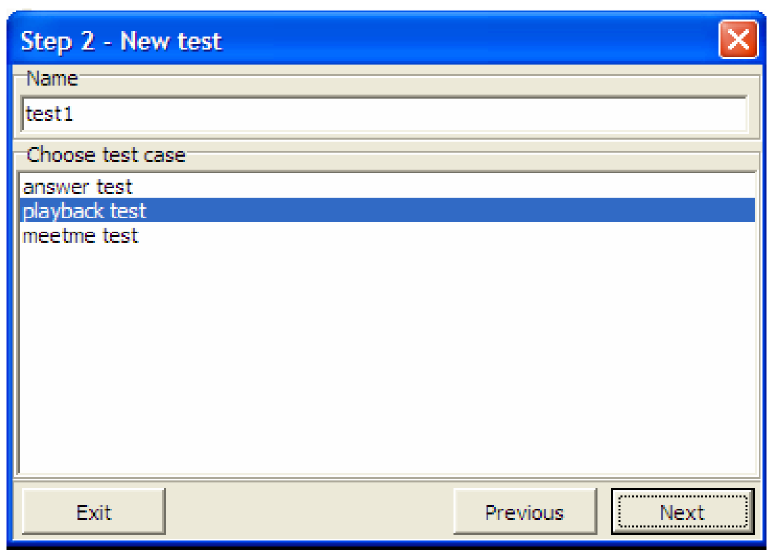


Figure 13. Astertest - Username and Password Are Both “test”

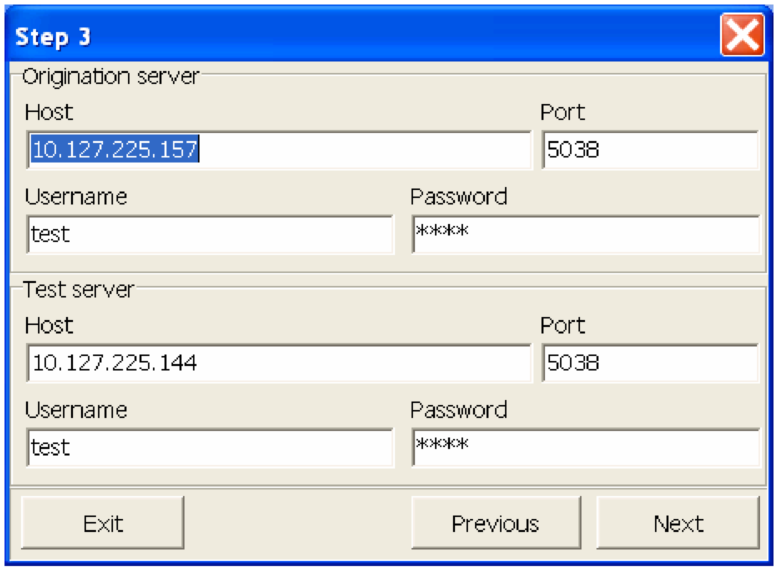


Figure 14. Astertest - Select GSM->iLBC Transcoding