Using Intel® System Studio in a Virtual Machine Environment

Whitepaper

9 December 2014

Contents
1 Introduction .................................................................................................................................................. 2
2 Intel® C++ Compiler .................................................................................................................................. 3
  2.1 Compiler Usage Principles .................................................................................................................. 3
  2.2 TCP/IP communicaton forwarding .................................................................................................... 4
3 Performance Libraries ............................................................................................................................... 6
4 Intel® VTune™ Amplifier for Systems ....................................................................................................... 6
  4.1 Dynamic binary instrumentation based data collection ....................................................................... 6
  4.2 Event-based Sampling ....................................................................................................................... 7
  4.3 Using FTrace or Perf with Intel® VTune™ Amplifier for Systems ...................................................... 8
  4.4 Example based on a virtualized Android® device ............................................................................... 8
5 Intel-enhanced GNU® GDB Application Debugger ............................................................................... 14
  5.1 Using GDB to debug applications running inside a virtual machine ............................................... 15
6 Intel® Inspector for Systems ..................................................................................................................... 18
7 SVEN Technology SDK ........................................................................................................................... 18
8 Capabilities not supported inside Virtual Machines ............................................................................ 18
9 Summary .................................................................................................................................................. 19
10 Attributions .............................................................................................................................................. 19
11 Disclaimer and Legal Information .......................................................................................................... 20
1 Introduction

Intel® System Studio and many of its components can be used for software development, analysis and debug targeting workloads running inside virtualized guest OSs. In many ways developing for a virtualized environment is only an extension of the concept of cross-development.

For compilers and libraries this implies that they can be used either in cross-build fashion or as a native compiler installed as part of your guest OS. Here as usual the expectation is that a GNU toolchain is present that the Intel® C++ Compiler can integrate with.

The Intel-enhanced GDB* application debugger can be used to debug locally inside a virtual machine or remotely using TCP-IP forwarding into the guest OS with a gdbserver debug agent running locally.

System-Visible Event Nexus (SVEN) instrumentation also has no strong dependency on hardware and thus can be used inside a Guest OS. The only dependency is access to a reliable OS timer signal.

The use of the Intel® VTune™ Amplifier for Systems poses the most complex challenge with some features available with virtualization and some having limitations or not being available. Therefore a considerable part of this whitepaper will focus on the use of the VTune™ Amplifier for Systems.

Two general limitations currently apply to Intel® System Studio and its use in a virtual environment. It does not actively support analysis and debug of workloads that are distributed across multiple guest OSs. Our Intel® System Debugger solution currently also does not support JTAG assisted debug of a guest OS running inside a virtual machine.
2 Intel® C++ Compiler

The main objective of virtualization is to obfuscate and abstract the underlying hardware of a runtime environment. The secondary objective frequently is to provide a separate container in which to isolate critical workloads that you may want to protect from interference from less critical lower priority tasks.

2.1 Compiler Usage Principles

As the compiler is largely independent of architecture details beyond the available instruction set and assumptions about cache and register set availability at run-time, it is not seriously influenced by virtualization at all.

The items I mention above are part of the core abstraction feature set of any virtual machine and thus the environment the compiler sees should not be different from the real hardware to any significant extent.

Working with the Intel® C++ Compiler targeting a guest OS is thus exactly identical to working with the compiler in any other scenario. As the guest OS frequently is feature reduced to minimize footprint and the number of processes that distract from the main purpose of a workload, using the Intel® C++ Compiler in this context usually is equivalent to using it for cross-development. The development machine being the development host and the guest OS inside the VM wrapper being the development target.

Please check out the articles on cross-development targeting Wind River* Linux*, Yocto Project* or Android* for reference below:


Redistributable compiler runtime libraries that may be required on the target platform, should you decide to use dynamic linking for your application or workload, can be found in the Intel® System Studio target package

- Windows* host:
  
  C:\Program Files (x86)\Intel\System Studio 2015.0.027\Targets\system_studio_target.tgz.
• Linux* Host:
   
   /opt/intelsystem_studio_2015.0.xxx/targets/system_studio_target.tgz

For target deployment of a codebase into a guest OS only a valid ssh connection (or adb connection in the case of Android*) with TCP-IP address forwarding into the virtual machine in question is needed. This is a standard feature for most virtual machines.

2.2 TCP/IP communication forwarding

If you require to set up IP forwarding from your development host into a guest OS,

Below is an outline of how to set it up for QEMU* and KVM* based environments

Ensure TCP/IP communication forwarding from inside the virtual machine and making the ip address of the virtual machine along with the port used for ssh/adb communication visible to the network as a whole.

Details on how to do this setup can be found on Wikibooks* (http://en.wikibooks.org/wiki/QEMU/Networking)

The basic steps are as follows

1. Install QEMU, the KQEMU accelerator and bridge-utils
   
   $ su -
   
   $ yum install qemu bridge-utils

2. Creating the image for the guest OS

   For best performance, you should install your guest OS to an image file. To create one, type:

   $ qemu-img create filename size[ M | G ]

   where filename is going to be the name of your image, and size is the size of your image with the suffix 'M' (MB) or 'G' (GB) right after the number, no spaces.

   $ qemu-img create Linux.img 10G

3. Configuring network for your guest OS

   Put the following contents into /etc/qemu-ifup:

   #!/bin/sh
# script to bring up the device in QEMU in bridged mode
#
# This script bridges eth0 and tap0. First take eth0 down, then bring it up with IP 0.0.0.0
#
/sbin/ifdown eth0
/sbin/ifconfig eth0 0.0.0.0 up
#
# Bring up tap0 with IP 0.0.0.0, create bridge br0 and add interfaces eth0 and tap0
#
/sbin/ifconfig tap0 0.0.0.0 promisc up
/usr/sbin/brctl addbr br0
/usr/sbin/brctl addif br0 eth0
/usr/sbin/brctl addif br0 tap0
#
# As we have only a single bridge and loops are not possible, turn spanning tree protocol off
#
/usr/sbin/brctl stp br0 off
#
# Bring up the bridge with IP 192.168.1.2 and add the default route
#
/sbin/ifconfig br0 192.168.1.2 up
/sbin/route add default gw 192.168.1.1
#stop firewalls
/sbin/service firestarter stop
/sbin/service iptables stop

The bold values can be changed. Please change the IP's to show your setup - The first bold is a comment, so it doesn't really matter. The second bolded value is the IP identical to the one assigned to your computer (means you'll need static IPs so you can predict your IP) and the third and last is your gateway.

Now, put this into /etc/qemu-ifdown:

#!/bin/sh
#
# Script to bring down and delete bridge br0 when QEMU exits
#
# Bring down eth0 and br0
#
/sbin/ifdown eth0
/sbin/ifdown br0
/sbin/ifconfig br0 down
#
# Delete the bridge
#
/usr/sbin/brctl delbr br0
#
# bring up eth0 in "normal" mode
#
/sbin/ifup eth0
# start firewalls again
/sbin/service firestarter start
/sbin/service iptables start

Make the scripts executable so QEMU can use them:

$ su -

$ chmod +x /etc/qemu-if*

$ exit

3  Performance Libraries
The statements made in the previous chapter about the Intel® C++ Compiler not being sensitive
to the virtualization environment are exactly true for

- Intel® Integrated Performance Primitives
- Intel® Math Kernel Library
- Intel® Threading Building Blocks

When using dynamic linking for these libraries you will also again want to be aware of any
redistributable runtime libraries you need as well.
Please consult the respective library user guides for details.

4  Intel® VTune™ Amplifier for Systems
Intel® VTune™ Amplifier for Systems supports three classes of sampling methods.

1. Dynamic binary instrumentation based data collection for hotspot analysis, locks & waits
   analysis.
2. Support for FTrace and Perf based sampling data
3. Event-based Sampling and Sampling with callstacks data collection

Event-based Sampling is generally not supported with virtual environments. There are some
exceptions that we will talk about below.

4.1  Dynamic binary instrumentation based data collection
The Intel® VTune™ Amplifier basic hotspot analysis without architectural event support, lock&wait analysis as well as concurrency analysis work inside a virtual machine exactly the same way as they would outside of a virtual environment.

Either ./amplxe-cl or ./amplxe-gui can be used inside of a virtual machine for the sampling methods listed above without any problems.

However, ssh (or adb in the case of Android) based remote sampling will not work, as it relies on the use of the performance monitoring unit (PMU) on the target platform. The PMU is however generally not fully virtualized.

Please refer to the Intel® VTune™ Amplifier for Systems User Guide for additional information on binary instrumentation based hotspot, lock&wait and concurrency analysis.

4.2 Event-based Sampling

The general approach of exposing architectural performance events inside a QEMU based virtual machine is to map these events to equivalent perf events.

This however poses two problems for Event based sampling using the SEP Sampling Collector, VTune™ Amplifier for Systems or the ssh/adb based remote sampling collection.

First, the Intel® VTune™ Amplifier for Systems tries to access the PMU directly and expects responses that are not covered by the Perf mapping.

Secondly, there are many events that can be sampled and are selected by standare features like General Exploration, that are not covered by this kind of high-level PMU virtualization as implemented with QEMU.

If you have full control of your virtual machine and can implement your own PMU virtualization and event export routines you may want to consult the Performance Monitoring Unit Sharing Guide at http://software.intel.com/file/30388/ for more details on the supported architectural events.

That said, there is one exception to the strict limitations listed above and that is VMWare* Fusion 5.x as outlined in the following blog article:


In order to use this feature, you need to activate a special "Advanced Processor Option." Here is what the VMWare documentation says:

From the VMware Fusion menu bar, select Window > Virtual Machine Library.
Select a virtual machine and click Settings.
In the Settings window, in the System Settings section, select Processors & Memory.
Click to expand Advanced Options and select the check boxes for the options that you want to use...
Set Advanced Processor Options
You can enable hypervisor applications, applications that use performance counters, or both in certain virtual machines.
With VMware Fusion 5 and later virtual machines, you have the following advanced options:

[Enable code profiling applications in this virtual machine]
You can use code profiling applications such as VTune or OProfile to optimize or debug software that runs inside a virtual machine.

4.3 Using FTrace or Perf with Intel® VTune™ Amplifier for Systems

Intel® VTune™ Amplifier for Systems contains a new feature to put events on the timeline. It allows users of VTune Amplifier to draw correlations between code running on the CPU (and other data collected by VTune Amplifier), and events spawned by Linux and Android OS frameworks.

Data collected by native Linux* performance sampling methods like Ftrace and Perf that do not depend on direct PMU access can be read into the Intel® VTune™ Amplifier for Systems following the instructions in the Intel® VTune™ Amplifier User Guide in the External Data Import chapter.

All that is then required is to have FTrace enabled in the Linux* or Android* OS build of your virtualized guest OS.

4.4 Example based on a virtualized Android* device

In the image below, you can see that this application received an “activityPause” in the Tasks and Frame tab, and that the AsyncTask #1 thread for this application continued to execute code as there is significant “CPU Time” attributed to that thread. Note: On the Android device this is when the “home” button is pressed, which put the application in the background, typically applications should stop executing code when they receive a pause event, and they should save the state of their application, to be resumed once the application receives a resume event.
Now select the region of code after the activityPause until the activityResume and then click "Zoom In and Filter in by Selection" on the code that was executing in AsyncTask #1.

Now switch to the Bottom Up tab, where the code which was executing after the activityPause occurred is displayed.
In this case, there are two functions CalculatePiInJava and CalculatePiInJNI. By double clicking on those functions – you will find these are loops that continue to Calculate Pi until they succeed, and they don’t have an interrupt handler or other mechanism to detect if the activityPause event occurs, and handle it gracefully. See [http://developer.android.com/training/basics/activity-lifecycle/pausing.html](http://developer.android.com/training/basics/activity-lifecycle/pausing.html) for information on how to write code to handle the pause event.

To showcase more features of VTune Amplifier, and to see yet another issue... On the timeline in the Tasks and Frame tab you can find an activityStop that really should have been responded to, for fear the application could have been killed prematurely without properly saving the current state of the program. The following screen shot shows that you can zoom in and look at events in more detail, and that you can organize the threads in the Tasks and Frame tab by clicking on a thread in that view and dragging it up or down.
This is just one use case scenario of the Android framework events. There are many other lectures and forums on the internet that explain how to use these Android framework events. But what Intel VTune Amplifier 2015 beta for Systems gives you is a much more flexible GUI and the ability to associate the Android framework events with other data collected on the system via VTune Amplifier for Systems.

The way to get the Android Framework Events in VTune Amplifier 2015 Beta is to create a new Custom Analysis in VTune Amplifier 2015 beta for Systems. The easiest way is to copy from one of the current analysis types... image below:
And then add which type of framework events you want to see in the new Android framework events control.
This dialog accepts a comma delimited list of events (without spaces), as specified in the help associated with this control. Potential events include: gfx,input,view,webview,wm,am,audio,video,camera,hal,res,dalvik.

- gfx - Graphics
- input - Input
- view - View
- webview - WebView
- wm - Window Manager
- am - Activity Manager (The option needed for Activity Lifecycle Events)
- audio - Audio
- video - Video
You can also collect via the command Line by adding the following  
-knob ftrace-
 config=event[,event] as explained in the help on the Command Line "knob" option.

An Example: amplxe-cl -collect-with runss -target-system=android -knob sampling-interval=2 -knob cpu-samples-mode=stack -knob ftrace-
 config=gfx,dalvik -knob chipset-event-
 config="GMCH_PARTIAL_WR_DRAM.ANY,GMCH_CORE_CLKS" --target-process com.intel.tbb.example.tachyon

5 Intel-enhanced GNU* GDB Application Debugger

Please refer to GDB: The GNU Project Debugger (http://www.gnu.org/software/gdb/) for details.

All Intel-enhanced GDB features except for BTS (Branch Trace Storage) based process specific instruction trace are available when debugging an application running inside a virtual machine as well.

For cross-development GDB comes with a remote debug agent called gdbserver. This debug agent can be installed on the debug target to launch a debuggee and attach to it remotely from the development host.

This can be useful in situations where the program needs to be run on a target host that is different from the host used for development, particularly when the target has a limited amount of resources (either CPU and/or memory).

To do so, start your program using gdbserver on the target machine. gdbserver then automatically suspends the execution of your program at its entry point, waiting for a debugger to connect to it. The following commands start an application and tells gdbserver to wait for a connection with the debugger on localhost port 2000.

$ gdbserver localhost:2000 program
Process program created; pid = 5685
Listening on port 2000
Once gdbserver has started listening, we can tell the debugger to establish a connection with this gdbserver, and then start the same debugging session as if the program was being debugged on the same host, directly under the control of GDB.

```
$ gdb program
(gdb) target remote targethost:4444
Remote debugging using targethost:4444
0x00007f29936d0af0 in ?? () from /lib64/ld-linux-x86-64.so.
(gdb) b foo.adb:3
Breakpoint 1 at 0x401f0c: file foo.adb, line 3.
(gdb) continue
Continuing.

Breakpoint 1, foo () at foo.adb:4
4       end foo;
```

It is also possible to use gdbserver to attach to an already running program, in which case the execution of that program is simply suspended until the connection between the debugger and gdbserver is established. The syntax would be

```
$ gdbserver localhost:2000 --attach 5685
```

to tell gdbserver to wait for GDB to attempt a debug connection to the running process with process ID 5685

### 5.1 Using GDB to debug applications running inside a virtual machine

Using GDB for remotely debugging an application running inside a virtual machine follows the same principle as remote debug using the gdbserver debug agent.

The only additional step is to ensure TCP/IP communication forwarding from inside the virtual machine and making the IP address of the virtual machine along with the port used for debug communication visible to the network as a whole.

Details on how to do this setup can be found on [Wikibooks*](http://en.wikibooks.org/wiki/QEMU/Networking)

The basic steps are as follows

4. Install QEMU, the KQEMU accelerator and bridge-utils
   
   ```
   $ su -
   $ yum install qemu bridge-utils
   ```
5. Creating the image for the guest OS

For best performance, you should install your guest OS to an image file. To create one, type:

```
$ qemu-img create filename size[ M | G ]
```

where `filename` is going to be the name of your image, and `size` is the size of your image with the suffix 'M' (MB) or 'G' (GB) right after the number, no spaces.

```
$ qemu-img create Linux.img 10G
```

6. Configuring network for your guest OS

Put the following contents into `/etc/qemu-ifup`:

```bash
#!/bin/sh
#
# script to bring up the device in QEMU in bridged mode
#
# This script bridges eth0 and tap0. First take eth0 down, then bring it up with IP 0.0.0.0
#
/sbin/ifdown eth0
/sbin/ifconfig eth0 0.0.0.0 up
#
# Bring up tap0 with IP 0.0.0.0, create bridge br0 and add interfaces eth0 and tap0
#
/sbin/ifconfig tap0 0.0.0.0 promisc up
/usr/sbin/brctl addbr br0
/usr/sbin/brctl addif br0 eth0
/usr/sbin/brctl addif br0 tap0
#
# As we have only a single bridge and loops are not possible, turn spanning tree protocol off
#
/usr/sbin/brctl stp br0 off
#
# Bring up the bridge with IP 192.168.1.2 and add the default route
#
/sbin/ifconfig br0 192.168.1.2 up
/sbin/route add default gw 192.168.1.1
#stop firewalls
/sbin/service firestarter stop
/sbin/service iptables stop
```

The bold values can be changed. Please change the IP's to show your setup - The first bold is a comment, so it doesn't really matter. The second bolded value is the IP identical to the one assigned to your computer (means you'll need static IPs so you can
predict your IP) and the third and last is your gateway.

Now, put this into /etc/qemu-ifdown:

```bash
#!/bin/sh
#
# Script to bring down and delete bridge br0 when QEMU exits
#
# Bring down eth0 and br0
#
/sbin/ifdown eth0
/sbin/ifdown br0
/sbin/ifconfig br0 down
#
# Delete the bridge
#
/usr/sbin/brctl delbr br0
#
# bring up eth0 in "normal" mode
#
/sbin/ifup eth0
#start firewalls again
/sbin/service firestarter start
/sbin/service iptables start
```

Make the scripts executable so QEMU can use them:

```bash
$ su -
$ chmod +x /etc/qemu-if*
$ exit
```

7. Installing the guest OS

Type the following to start the installation:

```bash
$ su
$ /sbin/modprobe tun
$ qemu -boot d -hda image.img -localtime -net nic -net tap -m 192 -usb -soundhw sb16 -cdrom /dev/hdc;/etc/qemu-ifdown
```

Where `image.img` was the name you gave to your image earlier. I'm also assuming `/dev/cdrom` is your CD drive - if it's not, then please change it to the correct device. After the install is complete, proceed to step 5.
8. Making the run script & running at will

The last step is to create the QEMU start script and from there on you can run your guest OS. Create this file - called *qemushort* - in the same directory as your image:

```
#!/bin/sh

su -c "/sbin/modprobe tun; qemu -boot c -hda image.img -localtime
-net nic -net tap -m 192 -usb -soundhw sb16;/etc/qemu-ifdown"
```

Where *image.img* was the name given to the image earlier.

Last step - make the startup script executable:

```
$ chmod +x /path/to/qemushort
```

6 Intel® Inspector for Systems

Intel® Inspector for Systems dynamically instruments the application to be analyzed and does not interact directly with any underlying hardware features. It is therefore possible to simply use ./inspxe-cl or ./inspxe-gui natively launched on the guest OS.

Please be aware of limitations with regards to real-time schedulers. The Intel® Inspector for Systems does not support real-time scheduler patches with Linux*.

7 SVEN Technology SDK

Drivers and applications running in your guest OS can be instrumented and the SVEN SDK kernel module that collects and logs tracing events can be compiled against and installed and launched on your Linux* guest OS just as it would on a non-virtualized host OS.

The SVEN SDK will use the OS timer provided and thus will need to rely on the timer signal provided being reasonably stable.

Please consult the SVEN SDK User's Guide for details.

8 Capabilities not supported inside Virtual Machines

Currently not supported for use with virtualized environments are:

1. Intel® Energy Profiler
2. SoC Watch for Windows* and Android*
3. Wu Watch for Android*
4. Performance Monitoring Unit based native and remote sampling for the Intel® VTune™ Amplifier for Systems
5. BTS (Branch Trace Storage) based process specific instruction trace with the Intel-enhanced GDB*
6. Intel® System Debugger

9 Summary
Using Intel® System Studio components to target guest OS based workloads inside a virtual machine requires to be aware of the limitations this will put on these components.

Some tools like the Intel® Inspector can only be run natively inside the guest OS. Some tools like the Intel® VTune™ Amplifier will have a considerable set of capabilities not available.

The majority of build tools, libraries and application debug capabilities are however not impacted at all.

10 Attributions

Portions of this software were originally based on the following:
- software copyright (c) 1999, Sun Microsystems., http://www.sun.com
- the W3C consortium (http://www.w3c.org),
- the SAX project (http://www.saxproject.org)
- voluntary contributions made by Paul Eng on behalf of the Apache Software Foundation that were originally developed at iClick, Inc., software copyright (c) 1999.

This product includes updcrc macro, Satchell Evaluations and Chuck Forsberg. Copyright (C) 1986 Stephen Satchell.

This product includes software developed by the MX4J project (http://mx4j.sourceforge.net).

This product includes ICU 1.8.1 and later. Copyright (c) 1995-2006 International Business Machines Corporation and others.

Portions copyright (c) 1997-2007 Cypress Semiconductor Corporation. All rights reserved.
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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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
Using Intel® System Studio inside a Virtual Machine Environment

responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm


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

Microsoft, Windows, Visual Studio, Visual C++, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Copyright (C) 2008–2014, Intel Corporation. All rights reserved.